Lab 1 : Raamwerken

19 februari 2016

Cedric De Boom, Hendrik Moens
(e-mail: {voornaam.achternaam}@intec.UGent.be)

1 Inleiding

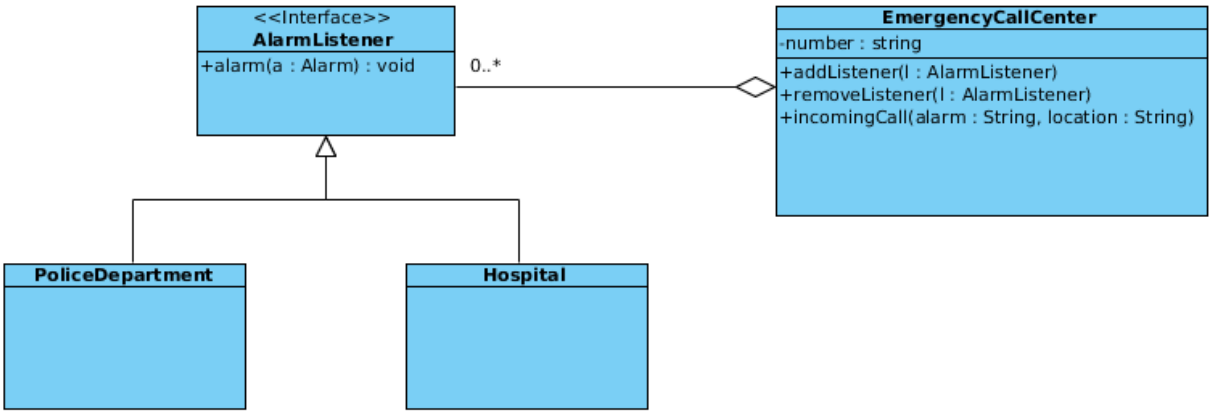
In de labsessies van het vak Software-ontwikkeling II gaan we dieper in op geavanceerdere programmeerconcepten zoals event-driven programming, sockets, multithreading, GUIs, software modules, unit testing enzovoort. Daarnaast komen ook allerlei aspecten van objectgeoriënteerd ontwerp en programmeren aan bod, met nadruk op het beperken van afhankelijkheden tussen verschillende subsystemen, hetgeen de herbruikbaarheid ten goede komt. In het kader daarvan zullen de labsessies steeds op elkaar volgen, waarbij telkens verder gebouwd wordt op code van vorige sessie(s). Het is dus zaak om elke sessie actief mee te werken en indien nodig de opdracht thuis verder af te werken. Hou ook steeds de code bij, aangezien ze later nog van pas zal komen.

De labsessies zullen gequoteerd worden en tellen mee voor 20% van de punten voor het vak Software-ontwikkeling II. Hiertoe dienen jullie op het einde van de labsessie jullie code te uploaden op het Indianoplatform. Let erop dat je steeds dezelfde package en klassenamen gebruikt zoals beschreven in de opgave.

In deze eerste labsessie bekijken we enkele veel voorkomende patronen voor communicatie tussen subsystemen: observer, event broker en whiteboard.

2 Observer

Een eerste eenvoudig en veelgebruikt patroon voor communicatie is het Observerpatroon. Dit wordt gebruikt wanneer een of meerdere klassen (de observers) geïnteresseerd zijn in een andere klasse (het subject). Het Observerpatroon wordt geïllustreerd in een voorbeeldapplicatie weergegeven in Figuur 1.



Figuur 1: Een noodoproep applicatie die het Observerpatroon illustreert. Verschillende instanties (bv. ziekenhuizen, politie, enz.) willen op de hoogte gehouden worden van binnenkomende berichten op een of meerdere noodoproep nummers (bv. 112, 101, enz.).

De applicatie stelt een eenvoudig programma voor dat noodoproepen op verschillende gereserveerde telefoonnummers (bv. 112, 101, enz.) beheert. Elk gereserveerd telefoonnummer wordt beheerd door een `EmergencyCallCenter` object. Deze klasse heeft de methode `incomingCall()` die opgeroepen wordt wanneer op het telefoonnummer een oproep binnenkomt, en krijgt als argumenten het type van de oproep mee en de locatie vanwaar de oproep komt.

Daarnaast zijn er ook een aantal objecten die verschillende instanties voorstellen die geïnteresseerd zijn in deze oproepen, en die allen de interface `AlarmListener` implementeren. Als een `AlarmListener` meldingen wil krijgen van oproepen op een bepaald noodnummer, dan registreert deze zich bij het desbetreffende `EmergencyCallCenter` door de methode `addListener`. Eens geregistreerd, zal de `alarm()` methode opgeroepen worden wanneer een oproep binnenkomt bij het `EmergencyCallCenter`. De broncode van deze applicatie is terug te vinden op Indianio.

Opgave 1

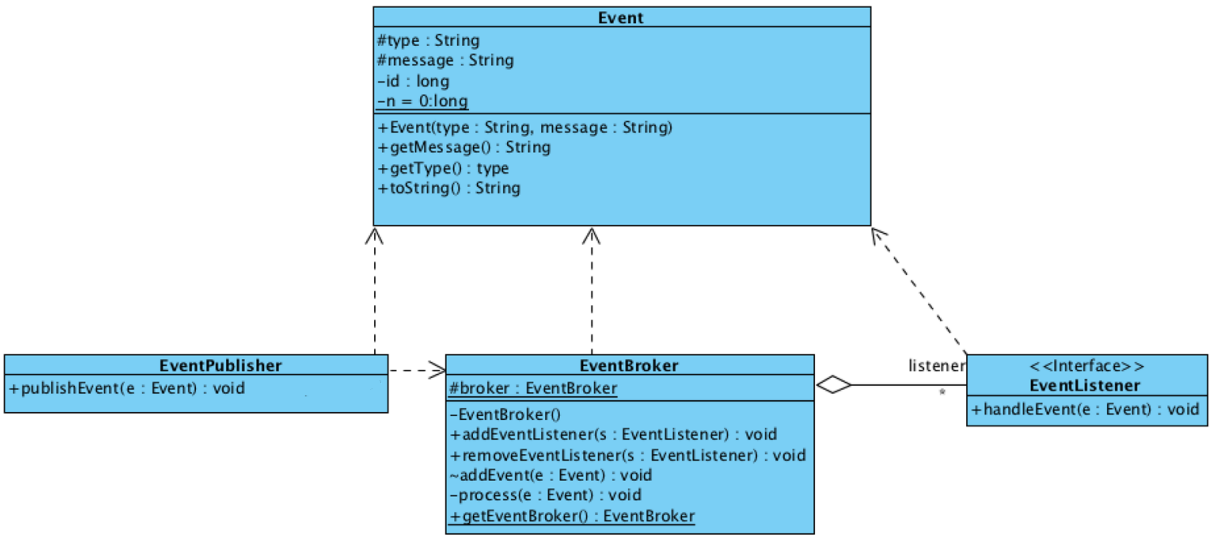
Download de broncode van Indianio. De klassen van de voorbeeldapplicatie bevinden zich in de package `alarm` en zijn voorzien van een uitvoerbare `Main` klasse om de applicatie te testen.

Breidt deze applicatie uit door een extra ziekenhuis toe te voegen (bv. "AZ"), dat luistert naar het 112 noodnummer en een extra noodnummer (bv. 101), waar enkel de politie in geïnteresseerd is. Implementeer daarnaast ook een nieuwe klasse `FireDepartment` die luistert naar het 112 noodnummer. Voeg in de `Main` klasse enkele extra binnenkomende oproepen toe ten einde je applicatie te testen. Let hierbij op het aantal regels "glue code" die je moet toevoegen om de observers aan de subjects te koppelen. Dit is meteen een van de nadelen van het Observerpatroon.

3 Event Broker

Een nadeel van het Observerpatroon is dat de observers steeds zelf moeten registreren bij alle subjects die ze willen volgen. Om dit probleem op te lossen en de observers los te koppelen van de subjects, kan een EventBroker gebruikt worden als Mediator.

Een UML diagram van een eenvoudige `EventBroker` is weergegeven in Figuur 2. In dit geval implementeren de observers de `EventListener` interface en registreren zij zichzelf bij de `EventBroker`. De subjects zijn extensies van `EventPublisher` en pushen hun `Event`s naar de `EventBroker`, die ze doorgeeft aan alle listeners. Op die manier zijn de `EventListener`s volledig losgekoppeld van de `EventPublisher`s. Aangezien er slechts één `EventBroker` is in het systeem waar zowel de publishers als listeners mee communiceren, is het nuttig om hier het singletonpatroon toe te passen.



Figuur 2: UML klasse diagram van het Event Broker patroon.

Opgave 2

Vorm de noodoproepapplicatie om zodat de communicatie gebeurt door gebruik te maken van de `EventBroker`. Plaats de nieuwe klassen in een aparte package `alarmevent`. Allereerst dienen `Event`s gedefiniëerd te worden die gebruikt worden voor de communicatie.

1. Maak de klasse `alarmevent.AlarmEvent`, die overerft van `Event`, en die naast de reeds bestaande `type` en `message` attributen, ook een nieuw `location` attribuut bevat. Voorzie de `AlarmEvent` klasse van een constructor die twee `String`s meekrijgt: de eerste voor het `type` alarm en de tweede voor de `location`. Als `message` vul je een *human-readable* boodschap in die het alarm event beschrijft (deze wordt gebruikt als je het event object uitprint), bv. "ALARM! crash at Plateaustraat".
2. Implementeer vervolgens een nieuwe klasse `alarmevent.EmergencyCallCenter` als `EventPublisher`. Deze implementeert opnieuw de methode `incomingCall` waarbij nu `AlarmEvent`s worden doorgegeven aan de `EventBroker` via de `publishEvent()` methode.
3. Maak ook nieuwe klasses `alarmevent.PoliceDepartment`, `alarmevent.Hospital` en `alarmevent.FireDepartment`, die nu de `EventListener` interface implementeren. Deze dienen zich ook te registreren bij de `EventBroker`.
4. Creëer ten slotte een nieuwe main klasse `alarmevent.Main`, die hetzelfde gedrag test als de oorspronkelijke `Main` klasse. Merk op dat er – buiten constructie van alle objecten – geen "glue code" meer nodig is om de communicatie op te zetten.

Wanneer je de applicatie test, zal je merken dat alle event listeners van alle events op de hoogte worden gebracht. Dit is niet gewenst in onze alarm applicatie. Daarom passen we in opgave 3 de event broker aan zodat er selectief geluisterd kan worden.

Opgave 3

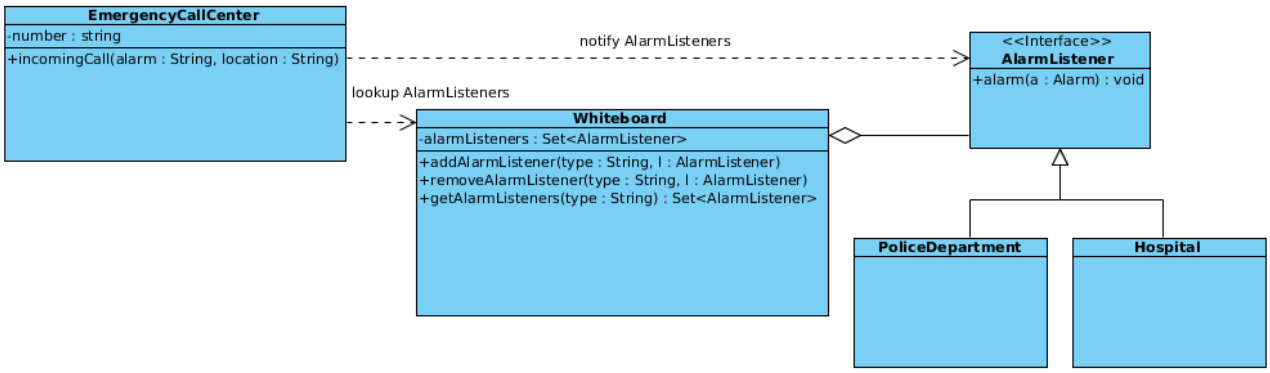
Het voordeel van de EventBroker – ontkoppeling tussen eventpublishers en event listeners – is tevens zijn nadeel: de eventpublisher heeft geen enkele controle over welke eventlisteners het event gaan afhandelen, en of het event wel afgehandeld wordt. Om daar (deels) aan tegemoet te komen kan de `EventBroker` ook optreden als filter, waarbij `Event`s van een bepaald type enkel naar bepaalde `EventListener`s gestuurd worden.

1. Voeg aan de klasse `EventBroker` de methode `addEventListener(String type, EventListener s)` toe, die `EventListener`s toelaat zich enkel te registreren voor `Event`s van het opgegeven type. Let op: het moet nog steeds mogelijk zijn om de oorspronkelijke `addEventListener` en `removeEventListener` methodes te gebruiken om generiek te luisteren.
2. Pas ook de `process()` methode aan zodat de binnekomende events daadwerkelijk gefilterd worden.
3. Wijzig tot slotte ook de registraties van de verschillende `EventListener`s. De brandweer is enkel geïnteresseerd in `Event`s van het type "fire". Een ziekenhuis wil enkel gecontacteerd worden in het geval van een "fire" of een "crash". De politie stuurt een eenheid bij elke noodoproep.

De applicatie kan getest worden met dezelfde `Main` klasse als Opgave 2. Controleer of de events nu inderdaad correct gefilterd worden.

4 Whiteboard

In sommige gevallen is het niet voldoende om `Event`s te filteren in de `EventBroker`, maar wil de `EventPublisher` volledige controle over wie de `Event`s bereiken zoals in het Observerpatroon. Aan de andere kant willen we wel de losse koppeling tussen publishers en listeners behouden. Om dit te bereiken kan gebruik gemaakt worden van een andere aanpak: het Whiteboard patroon, weergegeven op Figuur 3. Deze aanpak wordt ook gerefereerd als "*Service Locator*".



Figuur 3: UML klasse diagram van het Whiteboard patroon.

Bij het Whiteboard patroon registreren alle eventlisteners zich opnieuw bij een centrale entiteit, de klasse `Whiteboard`. In tegenstelling tot het EventBrokerpatroon, zullen de eventpublishers nu de events niet doorgeven aan deze centrale entiteit, maar enkel de verzameling van listeners opvragen, in ons voorbeeld met de methode `getAlarmListeners()`. Eens de collectie van listeners verkregen is, kan de eventpublishers zelf de gewenste listeners op de hoogte brengen van het event.

Net als bij de `EventBroker` kan het singletonpatroon toegepast worden om de `Whiteboard` globaal beschikbaar te maken. Ook kan het `Whiteboard` net als de `EventBroker` de listeners al filteren op het type van `Event`s ze wensen te ontvangen.

Opgave 4

Zoals je wellicht hebt opgemerkt bij het testen van de applicatie met de `EventBroker`, wordt bij elk oproep van type “crash” of “fire” een ziekenwagen gestuurd van elk ziekenhuis. In deze opgave willen we dat bij het `EmergencyCallCenter` binnenkomende oproepen gericht aan ziekenhuizen slechts aan één van de beschikbare ziekenhuizen gemeld worden, en er steeds wordt afgewisseld tussen de verschillende ziekenhuizen. Dit om het werk te verdelen.

1. Implementeer een nieuwe versie van de noodoproep applicatie, dit keer gebruik makend van het Whiteboard patroon zoals weergegeven in Figuur 3. Maak hiervoor een nieuwe package `alarmwhiteboard`. Implementeer de klasse `alarmwhiteboard.Whiteboard`. Gebruik hierbij het singletonpatroon en de mogelijkheid tot filteren zoals bij de `EventBroker`.

2. Implementeer daarnaast ook de klasse `alarmwhiteboard.EmergencyCallCenter`, die gebruik maakt van `Whiteboard` om de referenties te krijgen naar alle `AlarmListener`s. Wanneer de oproep van type “crash” of “fire” is, mag dit slechts aan één van de beschikbare ziekenhuizen gemeld worden en wordt er telkens afgewisseld van ziekenhuis. De politie en de brandweer blijven ook de gewenste oproepen krijgen zoals in de voorgaande opgave.

3. Om je applicatie te testen maak je de klasse `alarmwhiteboard.Main`, die nu gebruik maakt van het `Whiteboard`. Als `AlarmListener`s kunnen de klassen `Hospital`, `PoliceDepartment` en `FireDepartment` uit Opgave 1 gebruikt worden, waarbij je de `AlarmListener`s in de `main` methode registreert. Merk op dat je hier net als bij de event broker deze glue code zou kunnen vermijden door de `AlarmListener`s zelf te laten instaan voor hun registratie, maar dit is hier vermeden om de klassen uit Opgave 1 te kunnen hergebruiken.

5 Tot slot: uploaden

Per opgave wordt een afzonderlijke package verwacht, waarbinnen elk bestand slechts één klassedefinitie of interfacedefinitie bevat. Concreet verwachten we:

- Opgave 1: package `alarm`
  - `Alarm.java`

  - `AlarmListener.java`
  - `EmergencyCallCenter.java`
  - `FireDepartment.java`
  - `Hospital.java`
  - `PoliceDepartment.java`
  - `Main.java`
- Opgave 2: package `alarmevent`
  - `AlarmEvent.java`
  - `EmergencyCallCenter.java`
  - `FireDepartment.java`
  - `Hospital.java`
  - `PoliceDepartment.java`
  - `Main.java`
- Opgave 3: package `eventbroker`
  - `Event.java`
  - `EventBroker.java`
  - `EventListener.java`
  - `EventPublisher.java`
- Opgave 4: package `alarmwhiteboard`
  - `EmergencyCallCenter.java`
  - `Whiteboard.java`
  - `Main.java`

Je submit naar Indianio een zip-bestand, met als naam Groep_Nr_Lab1.zip. Dit bestand bevat dus mappen met dezelfde namen als de Java-packages zoals hierboven aangegeven. Alle bovenvermelde bestanden moeten aanwezig zijn (desnoods leeg), anders wordt je submissie niet aanvaard door Indianio.