Christian Zhuang-Qing Nielsen, 201504624

ISU 5: **Message Distribution System**

Indtil videre har vi arbejdet med en tråd der har adgang til en message queue, hvor andre tråde passerer meddelelser til denne MQ. Problemet ved dette er at andre tråde bliver nødt til at have adgang til dette MQ-objekt. De skal også vide på forhånd hvilken slags data de skal sende til MQ'en. Dette giver adgang til tovejskommunikation, men på bekostning af høj kopling.

MDS - hvorfor og hvordan?

Hvorfor: Vi har brug for et system med lav kopling, hvor afsenderen ikke behøver at antage modtagerens struktur for at kunne sende beskeder. Vi har brug for et system for flere modtagere kan få den samme besked (hvis nogen). Styringen af hvilke slags beskeder modtagerne skal modtage og afsenderen skal sende er ikke håndteret af dem selv, men derimod et MessageSystem.

Hvordan: Dette gøres ved at benytte **statusinformation** frem for requests/confirms. Mere specifikt til dette formål kan vi bruge det såkaldte **broadcasting**-design. Her har vi et MDS med `MessageSystem` og `SubscriberId`. Derudover har vi Publishers, der står for at sende beskeder, og vi har Subscribers, der ønsker at modtage beskeder. Dette design kræver at vi i forvejen har en kørende MDS, at vi bruger singletons (så vi kun har én MDS som alle har nem adgang til), at meddelelser er ***globalt*** identificérbare vha. stenge. Med denne metode er det kun muligt at have ***envejskommunikation***

Subscribers abbonerer på en bestemt navngivet meddelelse (i form af en `string`) de ønsker at få statusopdateringer på. Dette gøres ved at alle subscribers giver en message queue pointer (der peger mod deres MQ) til MDS, så den ved hvor den skal sende hen. Hver subscriber har også en lokal ID som de skal bruge til at finde ud af om en besked er rettet mod dem.

Publisheren notifier alle subscribers (hvis der er nogen), hvorefter hver af dem vil modtage meddelelsen destribueret med deres eget ID. Den gør dette ved at passere en meddelelse og den associerede globale ID, hvorefter MDS selv notifier alle med samme ID.

PostOffice-design - hvorfor og hvordan?

Hvorfor: Vi kan benytte os af PostOffice-designet når vi har en masse kompleks kommunikation mellem mange parter, som vi ønsker at forsimple. PostOffice gør brug af mediator pattern (se afsnittet om denne sidst). PostOffice gør det muligt at sende en besked til én specifik modtager, hvilket ofte kan være smart. Vi får lav kopling, singletons (så vi kun har ét PostOffice som alle har nem adgang til) og tovejskommunikation.
Hvordan: Vi skaber et centralt "Postkontor", hvor alle meddelelser fra alle trådene skal igennem for at blive "distribueret" til den rigtige modtager. Dette gøres ved at navngive modtageren i `string`-format eller have en handler. Vi kan benytte vores forhenværende message queue system, så længe vi har en fungerende PostOffice-klasse.

Decoupling fra disse metoder

PostOffice giver decoupling da afsenderen ikke behøver at kende modtageren. Det simplificerer kommunikationen mellem trådene en masse, og gør også koden nemmere at læse for udviklere.

MDS giver lav kopling fordi publishers ikke behøver have kendskab til nogen subscribers. De behøver faktisk ikke engang vide om der er nogle subscribers til at modtage deres meddelelser. Fordi de ikke er koplet sammen, så kan der kommunikeres på tværs af abstraktionslag, uden at tage større forbehold for hvordan meddelelsesstrukturen skal passeres rundt.

Design - overvejelser og implementation

MDS

I MDS abbonnerer **subscriberen** igennem MDS ved metoden `MessageDistributionSystem::getInstance().subscribe()`. Den har også en `handleMsg()`, som er en switch der håndterer forskellig funktionalitet afhængig af hvilken ID der bliver sendt med rundt, f.eks. ved at notify andre subscribers. **Publisheren** har en metode `MessageDistributionSystem::getInstance().notify()`, som den bruger til at notify subscribers der har abboneret på en given ID.

De forskellige meddelelsesstrukturer og deklarationer af globale string ID'er (som bruges til at finde subscribers) er erklæret inde i `Common header file(s)`. Den egentlige implementering af metoderne er i kildekodefilerne.

PostOffice

I PostOffice kalder en tråd `send()`, nogle parametre (heriblandt modtagerID og selve meddelelsen). PostOffice finder modtageren ud fra dens ID, og sender beskeden videre til denne. Når modtageren har fået beskeden sender den en acknowledgement/bekræftelse at den har modtaget beskeden. PostOffice sender derefter denne bekræftelse videre til den originale afsender.

Igen findes identifikationen i seperate header files. I afsenderen kan den vælge hvilke beskeder den vil sende med send metoden, og hos modtageren har den en række `handle_X_Message()`, som den bruger til at udføre en funktion på baggrund af hvad der stod i beskeden.

Design Patterns (i sig selv og ift. MDS/PostOffice)

Observer Pattern

Observer pattern kan bruges når vi gerne vil have en notifikation omkring et event, men ikke har lyst til at polle dataen. Det er en One-to-many løsning, der kun har envejskommunikation. Eksempler på brug kunne være en handling i en GUI eller en sensor der har fået en ny værdi som andre skal have kendskab til.

Vi har brugt observer pattern i vores MDS. Dette er temmelig tydeligt da vi har en lang række observers der i dette tilfælde er vores subscribers, og et subject der er vores publisher/MDS.

Pros/Cons:

- 😃 Lav kopling.
- 😃 Mange kan modtage beskeder på samme tid.
- 😦 Det koster at lave opdateringer (alle skal have kendskab til dette).
- 😦 Subscribere kan ofte være meget langsomme til at handle deres meddelelse, hvilket påvirker publisheren (åbenbart).

Mediator Pattern

Man bruger mediator pattern når ens program har alt for høj kopling, og man bliver nødt til at fjerne noget af kendskaben til hinanden. Bliver også nødt til at fjerne brugen af f.eks. MsgQueues kendskab til hinanden. Vi bruger mediator når vi har veldefineret kompleksitet der giver en masse afhængighed i koden, hvilket gør den ustruktureret. Vi bruger det også når det er svært at genbruge et objekt fordi indeholder for mange referencer (og kommunikerer) til alt for mange andre objekter.

Vi har brugt mediator pattern i både MDS og PostOffice (en central enhed der styrer kommunikationen mellem parterne). I det ene tilfælde var det afsendere/modtagere, i det andet tilfælde var det kommunikationen mellem publishers og subscribers. I virkeligheden kan det bruge i grafisk software, f.eks. når en grafisk update skal spredes ud til alle interesserede parter.

Pros/Cons:

- 😃/😦 Centraliceret kontrol
- 😃 Fokuserer på hvordan objekter interagerer og ikke hvordan de opfører sig.
- 😃 Entiteter behøver ikke have kendskab til hinanden.

Singleton Pattern

Man bruger singleton pattern når man vil sikre sig at en klasse kun har én instans, og at der global adgang til den. Det er ofte system-wide adgang til et givent objekt, hvilket betyder er der en masse pointers og referencer som skal passeres rundt. Eksempler på brug er f.eks. konfigurationsservice, loggingservice, og enhver applikation-wide service generelt.

Vi har benyttet os af singletons i både MDS og PostOffice til at sikre os at der kun har været én mediator-klasse, som alle andre har haft nem global adgang til.

Pros/Cons:

- 😦 Global variabel-ish, hvilket betyder at flere ikke kan tilgå på samme tid --> seriel access
- 😦 Hvem skal skabe variablen og hvem skal delete den når den ikke bliver brugt mere?
    - I vores MDS har vi brugt *static block initilization approach*:

- 😃 Første gang den bliver tilgået bliver den skabt.
- 😃 Super nemt at kode og forstå
- 😃 Ingen låse (i denne løsning)
- 😞 Første tilgang til variablen kræver at den bliver skabt --> kan medføre mutlithreading udfordringer.

Man skal ikke bruge double-checked locking idiomet med Singletons, da det ikke virker korrekt.