Christian Zhuang-Qing Nielsen, 201504624

ISU 4: OS API

- ISU 4: OS API
  - Designfilosofien
    - Hvorfor Objektorienteret?
    - Hvorfor OS API?
  - Udfordringer og Design
    - PIMPL / Cheshire Cat idiom
    - CPU- / OS-arkitektur
  - Effect on design/implementation
    - Message Queues - pthread vs OS API
    - RAII i brug
  - UML Diagrammer til implementering (Klasse- og Sekvensdiagrammer). Hvordan?

Designfilosofien

En API er et interface der hjælper lave applikationer. I alle tilfælde vil en API gøre det mulige at skjule dele af systemet (gøre det simplere for programmører). Ydermere gør en API det muligt for API'ens udviklere kun at vise de metoder, som rent faktisk bør bruges af andre udviklere, hvilket øger sikkerheden i systemet. Til sidst kan en API forhindre folk i at benytte sig af dele af systemet som i ikke må.

Hvorfor Objektorienteret?

- Det er nemmere hvis man er vant til objekt-orienteret programmering.
- Renere kode
- Det er nemmere at implementere ud fra fra diagrammer (f.eks. klasse- og sekvens-).

Hvorfor OS API?

I dette tilfælde er det operativsystemets API vi taler om. Denne API lægger sig som et abstraktionslag mellem det konkrete operativsystem og applikationerne man udvikler til dem, hvilket gør man ikke behøver at lave applikationer meget om. Med sådan en API er det muligt at tilgå operativsystemets ressourcer og metoder på simpel vis. Det giver også en fælles platform at udvikle metoder til flere forskellige operativsystemer, hvor kun en brøkdel af koden skal ændres/tilpasses for at metoderne vil kunne fungere på det andet OS. Ved at bruge den samme API igen og igen vil det også være langt nemmere for udviklerne at lave nye funktioner i fremtiden.

Med disse OS API'er er det også muligt at udøve **Cross-development**, hvilket betyder man kan udvikle systemet på sin host platform og så debugge til der ikke er flere fejl. De ting som er hardware bestemte (f.eks. tiden, gyroskop, osv) kan man bare bruge stubbe som implementeres når man udvikler systemet til et helt nyt system under denne abstrakte API.

Udfordringer og Design

PIMPL / Cheshire Cat idiom

PIMPL bruges til at skjule implementeringen af en klasse ved at wrappe denne inde i en struct, som ikke kan tilgås fra APIen/biblioteket. Det er en af metoderne der bruges til at skjule dele af systemet for udviklerne.

CPU- / OS-arkitektur

I vores OS API er windows-implementeringen og linux-implementeringen opdelt i forskellige mapper. Koden minder meget om hinanden og headerfilerne er næsten ens, men alligevel er der ændringer som skal laves for at få det til at fungere. Disse er bygget videre ud fra det fælles "abstrakte API lag" som ligger mellem de konkrete OS-API'er og applikationen.

Det er også relevant hvilken type CPU man bruger, da filerne skal compiles til den rigtige target-arkitektur. Makroer, definitioner, mapper osv. varierer også af de forskellige operativsystemet.

Effect on design/implementation

Message Queues - pthread vs OS API

I vores OS API er tråd-klassen blevet implementeret af os. Den overordnede `Thread`-klasse har en privat variabel som er en `ThreadFunctor`. denne *TF er selve tråden* (dvs. det er i denne at `run()`-metoden findes. `Thread`-klassen er bare den kontrollerende enhed, som håndterer trådens start, dens prioritet, og at man skal vente på at den er færdig: `join()`-metoden).

I API'en bruges også `pthread_create()` til at initialisere tråden ligesom tidligere. Denne gang er Message-klassen beskrevet af osapi'en. Tråden indeholder selv handleren, som den bruger til at bearbejde meddelelsen og vælge ud fra denne hvad der skal ske. Dette gøres ved at switche på de events den handler.

RAII i brug

Vi husker at implementere delete i de klasser vi laver i OS-api'en for at sikre garbage collection når variablerne ikke bliver brugt længere / er out of scope. Hukommelsen til disse variabler og objekter bliver først allokeret når de bliver skabt, hvilket også er en fordel frem for at allokere på forhånd.

UML Diagrammer til implementering (Klasse- og Sekvensdiagrammer). Hvordan?

Det smate ved objektorienteret programmering er at det såkaldte ***representational gap*** er lille. Dette betyder at det er nemt at overføre noget fra et UML-diagram til reel implentering af kode. I et klasse-diagram kan man f.eks. tegne alle de klasser som arbejder sammen, samt hvilke metoder de har, og i et sekvensdiagram kan man se hvilke metoder der skal bruges hvornår. Kombinationen af disse to diagrammer letter implementeringsbyrden gevaldigt.