**Ubåtscentralen**

|  |  |  |
| --- | --- | --- |
| **Namn** |  | Simon Roos |
| **Utbildning** |  | Pythonutvecklare med inriktning AI |
| **Datum** |  | 2024-09-27 |

**Sammanfattning**

Denna rapport beskriver utvecklingen av ett system för övervakning och hantering av ubåtar. Systemet bearbetar rörelserapporter och sensordata, och sensorfel loggas samt kategoriseras baserat på inläst data. Kärnvapenaktivering, torpedavfyrningar, samt kollisioner mellan ubåtar hanteras lika så. *Unittesting* har genomförts, men fokuserar främst på ovanliga scenarier. En modulbaserad struktur för lättare underhåll, samt prestandaförbättringar genom integration med ett lågnivåspråk, såsom C++, för att effektivisera systemets tyngre processer, avsågs som viktiga lärdomar.

Innehåll

[1. Projektbeskrivning 4](#_Toc178362314)

[1.1 Bakgrund 4](#_Toc178362315)

[1.2 Syfte 4](#_Toc178362316)

[1.3 Mål 4](#_Toc178362317)

[1.4 Avgränsningar och fokus 5](#_Toc178362318)

[1.5 Tidplan/aktiviteter 5](#_Toc178362319)

[2. Teknisk bakgrund och val 6](#_Toc178362320)

[2.1 Övergripande 6](#_Toc178362321)

[2.2 *SubmarineSystem*-klassen 6](#_Toc178362322)

[2.3 *Type hinting* 6](#_Toc178362323)

[2.4 Sökning 7](#_Toc178362324)

[2.5 Inläsning och loggning av *MovementReports* 7](#_Toc178362325)

[2.6 Loggning av *Sensordata* 7](#_Toc178362326)

[2.7 Nukes 8](#_Toc178362327)

[2.8 Torpeder 8](#_Toc178362328)

[2.9 Felhantering 9](#_Toc178362329)

[2.91 Testning 9](#_Toc178362330)

[2.92 Demonstration 10](#_Toc178362331)

[3. Resultat 10](#_Toc178362332)

[4. Avslutning och slutsatser 11](#_Toc178362333)

[5. Referenslista 11](#_Toc178362334)

[Bilagor 12](#_Toc178362335)

[Bilaga 1 12](#_Toc178362336)

[Bilaga 2 12](#_Toc178362337)

[Bilaga 3 13](#_Toc178362338)

# 1. Projektbeskrivning

## 1.1 Bakgrund

När målet är att skapa ett relativt avancerat system som kräver effektivitet i form av datorprestanda, men även människo-läsbarhet, gäller det att rätt verktyg används för jobbet. När det involverar ett avancerat system som till exempel ett centralsystem för ubåtar, är det i regel godtyckligt att applicera objekt-orienterad kod. Objektorienterad programmering uppstod cirka 1960 då Ole-Johan Dahl, tillsammans med Kristen Nygaard utvecklade SIMULA språken på Norwegian Computing Center (NCC).[[1]](#footnote-1) Objektorienterad programmering är ett sätt att strukturera och organisera programmet så att det ska vara lättare att första hur olika delar kommunicerar med varandra.

## 1.2 Syfte

Syftet med arbetet var i första hand att demonstrera färdigheter i att kunna arbeta enligt en kunds specifikationer, att kunna skriva en läsbar och övergripande effektiv objekt-orienterad Pythonkod, samt bevisa kunskap i att dokumentera och genomföra ett programmeringsprojekt. I andra hand var syftet att utvecklare ny expertis i alla nämnda områden.

## 1.3 Mål

Målet var att skapa en applikation, eller ett API som tillfredsställer kundens krav. Mer specifikt: ett centralsystem som har hand om tusentals ubåtar till Lindas lustfyllda rederi. Fokus låg på att skapa en robust, välutvecklad kod med genomtänkt användning av objekt-orienterad programmering, som är både tidseffektiv och minnesnål, men också läsbar och lätt att underhålla.

## 1.4 Avgränsningar och fokus

Att designa ett ubåtssystem som även hanterar kärnvapen skulle kräva omfattande säkerhetslogik och liknande åtgärder. Därför väljs det att bortse från dessa mer realistiska aspekter av projektet, såsom säkerhetshantering, eftersom målet enbart är att demonstrera kompetens inom Pythonprogrammering och projektgenomföring med mera.

## 1.5 Tidplan/aktiviteter

**Vecka 34**

*Mål: Få en uppfattning av vad uppgiften kräver. Börja skriva en Submarine-klass.*

*Resultat: En Submarine-klass skrevs, med proper felhantering för angivelsen av serienummer.*

**Vecka 35**

*Mål: Expandera Submarine-klassen, färdiggör efterfrågade attribut och metoder.*

*Resultat: Submarine-klassen vidareutvecklas, position och sensordata hanteras.*

**Vecka 36**

*Mål: Framställa strukturen för programmet.*

*Resultat: SubmarineSystem-klassen införs. Allting angående ubåtarna hanteras av ett SubmarineSystem objekt som användaren skapar en instans av.*

**Vecka 37**

*Mål: Alla specifikationer ska vara uppfyllda.*

*Resultat: Alla efterfrågade funktioner förutom “nukes” är implementerade.*

**Vecka 38**

*Mål: ~~Programmet ska vara så optimerat och användarvänligt som möjligt. Klasserna ska vara bra kapsylerade.~~ Skapa nuke systemet och inför en unittester.*

*Resultat: Programmet kan rent praktiskt presenteras då alla specifikationer har implementerats. Nu saknas endast dokumentation.*

**Vecka 39**

*Mål: Projektet ska kunna presenteras fullständigt.*

*Resultat: Projektet presenteras och dokumentationen skrivs färdigt.*

# 2. Teknisk bakgrund och val

## 2.1 Övergripande

När programmet körs så presenteras användaren endast av en demonstration av systemet. Programmet är tänkt att användas som en modul där användaren endast importerar *SubmarineSystem*-klassen (och de nya typerna för type hinting, om så önskas) från *submarine\_system*.

from submarine\_system import SubmarineSystem

Användaren skapar sedan ett *SubmarineSystem*-objekt som sedan interageras med under programmets gång. Programmet saknar ett UI eftersom enbart *backend* har tagits i beaktning.

## 2.2 *SubmarineSystem*-klassen

Hanteringen av ubåtar sker endast genom *SubmarineSystem*-klassen. Valet att endast ha en klass som användaren interagerar med är gjort med tanke på användarvänlighet och enkelhet. Det är ej avsett att användaren ska interagera med ubåtsobjekten, då dessa är en privat del av klassen för systemet. Användaren kan enbart inhämta information från ubåtarna och sköta dem via ett *SubmarineSystem*-objekt.

## 2.3 *Type hinting*

Koden använder sig i stor utsträckning av type hinting för att tydliggöra vilka typer av returvärden som funktioner och metoder ger samt vilka argument de förväntar sig. Genom att använda type hinting blir koden lättare att förstå för den som granskar den, vilket är särskilt värdefullt för nya utvecklare som ska fortsätta utveckla systemet. Även variabler, framför allt de som representerar mer komplexa datastrukturer, såsom ett *dictionary* som innehåller andra *dictionaries*, type hintas ibland.

I vissa fall används även *NewType*, vilket ger utvecklaren möjlighet att definiera nya typer utan att behöva skapa en ny klass. Exempelvis finns en typ kallad *SerialNumber* i systemet, som representerar en sträng i ett specifikt format (XXXXXXXX-XX). Systemet utnyttjar dessutom *NewType* för att definiera typer av avancerade datastrukturer. Till exempel *MovementLog* är en *deque* som innehåller en ny typ *MovementLogEntry*, som i sin tur är en *tuple* som innehåller en gammal position, en riktining, ett avstånd, och en ny position*.*

## 2.4 Sökning

Systemet hanterar information om ubåtar genom att använda ett *dictionary*, där varje ubåt kan identifieras och sökas upp med sitt unika serienummer. I detta sammanhang prioriteras tidseffektiviteten vid uppslagningar, då *dictionaries* erbjuder snabbare åtkomsttider jämfört med andra datastrukturer. Även om ett *dictionary* inte är lika minnesekonomiskt som exempelvis en lista, anses minnesförbrukningen vara försumbar med tanke på att det endast rör sig om ett tusental ubåtar, särskilt i relation till andra operationer som systemet utför.

## 2.5 Inläsning och loggning av *MovementReports*

Vid inläsning av rörelsefiler behandlas varje fil sekventiellt, rad för rad. Varje rörelse, som omfattar riktning och avstånd, loggas och lagras för respektive ubåt i en deque-struktur. Denna metod bidrar till ökad minneseffektivitet, då endast de senaste rörelserna (för närvarande 50 stycken) bevaras, istället för att alla loggar sparas. Att logga varje rörelse för samtliga ubåtar medför en betydande tidskostnad. För att optimera denna process lagras all relevant information—inklusive gammal position, riktning, avstånd och ny position—i tuppler, som utgör en av Pythons mest tidseffektiva datastrukturer.

Kollisioner mellan ubåtar loggas även. För att avgöra om två ubåtar kolliderat, mappar systemet varje ubåts slutposition i ett *dictionary*. Sedan kollar systemet för varje ny ubåt om dess slutposition redan är mappad i dictionaryt, för att slutligen bedöma att en kollision har skett.

## 2.6 Loggning av *Sensordata*

Varje ubåt registrerar information om sensorfel baserat på dess sensordata. Vid inläsning av sensordata bearbetar systemet fil för fil, rad för rad, för att identifiera alla uppkomna sensorproblem. Om en rad innehåller minst en nolla klassificeras detta som ett sensorfel. Systemet kategoriserar alla rader utifrån olika typer av sensorfel. Om samma sensorer på samma positioner visar en nolla i flera olika rader, sammanräknas dessa rader som en typ av sensorfel.

I slutändan sorterar systemet alla olika typer av sensorfel samt antalet sensorer som varit felaktiga vid respektive incident. För att beräkna antalet fel som inträffade på en rad används metoden *string.count*, vilket föredras framför en loop på grund av dess tidseffektivitet, då den är implementerad i C.

## 2.7 Nukes

Systemet hanterar även kärnvapen som kan aktiveras med en hemlig aktiveringskod och nyckel. Dessa koder och nycklar lagras inte i klartext i systemet, utan läses in som en SHA256-hash. Detta är av stor vikt för att förhindra läckage av denna känsliga information.

Klassansvaret är betydande i detta sammanhang; klassen *SubmarineSystem* ansvarar för att ta emot användarens autentiseringsträng (bestående av datum, hemlig nyckel och aktiveringskod) samt serienumret för den aktuella ubåten. Denna sträng omvandlas till en hash-sträng, som systemet sedan skickar till det interna ubåtsobjektet med det respektive serienumret. Aktiveringslogiken hanteras inom ubåtsobjektet, som först kontrollerar att den angivna hashen är korrekt och sedan returnerar om kärnvapnet har aktiverats eller ej.

## 2.8 Torpeder

Användaren kan beställa en torped genom klassen *SubmarineSystem*. När en torped beställs anropas en metod för beställning som enbart verifierar att den aktuella ubåten existerar innan torpeden avfyras. Denna metod är dekorerad med en annan metod, *\_prevent\_friendly\_fire*. Denna dekorator innehåller logik som möjliggör för systemet att bedöma om en torped avfyrad från den angivna ubåtens position, i den angivna riktningen, skulle träffa en annan ubåt hanterad av systemet.

Valet att implementera en dekorator för att kontrollera friendly fire, tillsammans med en separat metod för beställning, grundar sig på en tydlig uppdelning av ansvar mellan de olika metoderna. Detta skapar en mer konsekvent kodstruktur, där namnen på metoderna tydligt förmedlar deras avsedda funktioner.

## 2.9 Felhantering

Ett stort fokus läggs på felhantering i programmet. Programmet kastar *Exceptions* i stor utsträckning, allra oftast när användaren försöker referera till en ubåt som inte finns, eller läsa in filer som inte finns. Programmet kastar inte alltid *exceptions*, ibland när oväsentliga fel uppstår, som när en rörelserapport innehåller en icke-existerande riktning, promptas användaren endast med en varning. Felhantering är smidigt i Python eftersom man kan förtydliga för andra utvecklare som använder koden vad exakt som är fel.

Programmet använder sig mycket av *property* som är en inbyggd dekorator i Python som i kort gör det möjligt att köra ytterligare logik innan ett attribut hämtas från en klass, vilket i detta fall används för att hantera fel.

## 2.91 Testning

I filen *test.py* finns en enhetstestare, *TestCase*, som heter *SubSysTest*. Här testas många av de metoder som klassen *SubmarineSystem* tillhandahåller för att säkerställa att de fungerar som avsett. Tester fokuserar särskilt på ovanliga fall, såsom när systemet försöker referera till en icke-existerande ubåt eller när man försöker registrera en ubåt med ett felaktigt serienummerformat. Anledningen till att fokus lades på just dessa ovanliga fall är att det är i dessa situationer som systemets beteende förutses vara som mest oförutsägbart.

Testcaset instansierar ett nytt *SubmarineSystem* i metoden *setUp*, men registrerar inga ubåtar. När tester utförs som kräver en ubåt, används en dekorator som skapar en ny ubåt innan testet körs och förstör den efter att testet har genomförts.

## 2.92 Demonstration

När programmet körs direkt presenteras en demonstration av systemets funktionalitet för användaren. Demonstrationen inleds med att all sensordata läses in, följt av inläsningen av rörelserapporter. När dessa två processer är slutförda, visas de mest centrala delarna av systemet upp. Först presenteras en rapport över sensorfel samt en rörelserapport för en specifik ubåt (se bilaga 1). Därefter visas vilka ubåtar som har orsakat kollisioner med andra ubåtar, samt vilken ubåt som befinner sig längst bort och längst ner, respektive närmast och högst upp (se bilaga 2).

Slutligen avfyrar alla ubåtar torpeder i slumpmässiga riktningar, och systemet loggar om det sker någon incident av "friendly fire". Dessutom testas aktiveringen av ett kärnvapen för en av ubåtarna (se bilaga 2 och 3).

# 3. Resultat

Alla efterfrågade specifikationer har implementerats och fungerar som förväntat. Den mest tidskrävande processen är att flytta alla ubåtar enligt rörelserapporterna, följt av inläsningen av sensordata. Rörelserapporterna tar betydligt längre tid än hanteringen av sensordatan. Sammanlagt tar det cirka 15 minuter att flytta och utföra sensorfelsökning för cirka 6000 ubåtar. En stor del av denna tid spenderas på loggning. Att logga fullständiga rapporter för de cirka 6000 ubåtarna använder strax över 100 MB RAM.

Koden är välstrukturerad och består av endast två klasser, varav en är en medlem av den andra. Metoderna är organiserade på ett logiskt och sammanhängande sätt, så att de är rimligt relaterade till varandra. De mest centrala metoderna som användaren förväntas använda är försedda med *docstrings* som förklarar deras funktion.

# 4. Avslutning och slutsatser

Eftersom all kod, förutom testningen, är skriven i en enda fil kan det vara svårt att navigera och hitta specifika delar av koden. En uppdelning av funktioner och komponenter i moduler samt en strukturering av projektet som ett Pythonpaket hade varit fördelaktigt. Detta skulle bidra till ökad tydlighet kring vilka delar som ansvarar för vilka funktioner, samt underlätta sökandet efter specifika komponenter.

Ytterligare testning hade varit nödvändig, då testningen för närvarande endast omfattar ovanliga fall. Det är av stor vikt att även testa vanliga användningsfall, då det ofta är i dessa situationer som det upptäcks att koden inte fungerar på det sätt den förväntades göra.

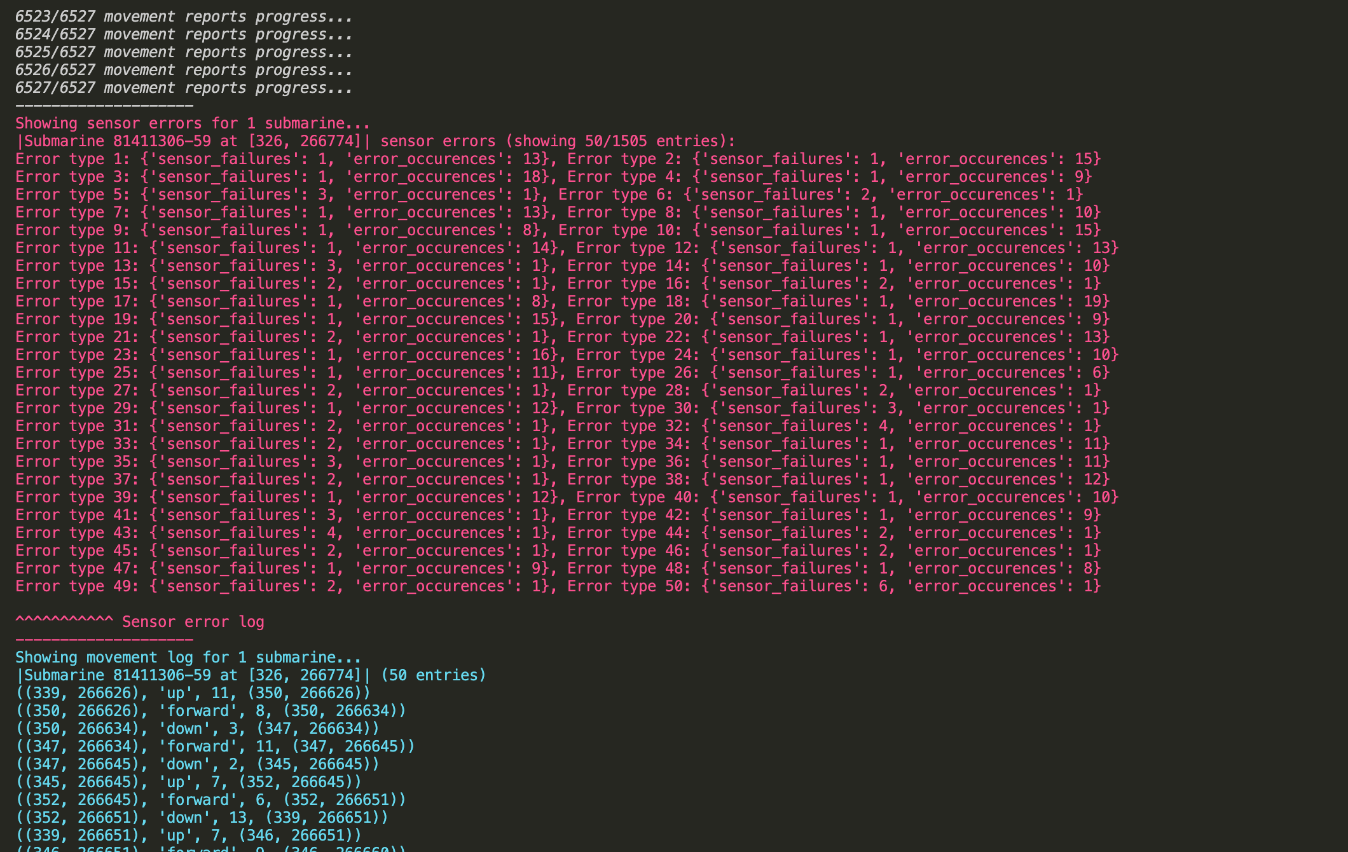
En annan insikt från detta projekt är att Python inte är det mest tidseffektiva språket. En mer effektiv lösning för hanteringen av rörelserapporter och sensorfelsökning kunde ha varit att implementera en pipeline till ett lågnivåspråk, såsom C++, för att förbättra prestandan.

# 5. Referenslista

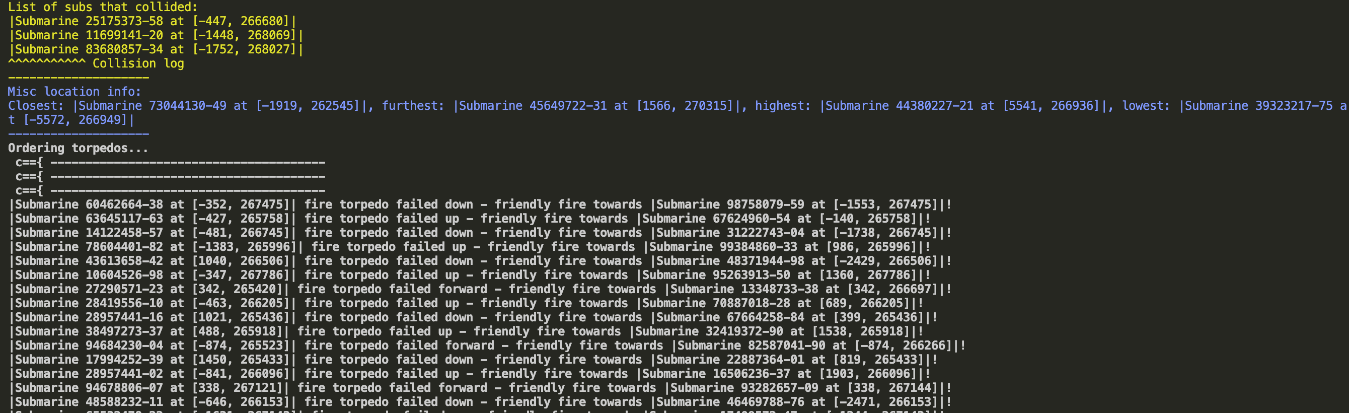
Andrew P. Black, “Ole-Johan Dahl A.M. Turing Award Laureate”, A.M. Turing, 2001, <https://amturing.acm.org/award_winners/dahl_6917600.cfm> (åtkomstdatum 24 september 2024)

# Bilagor

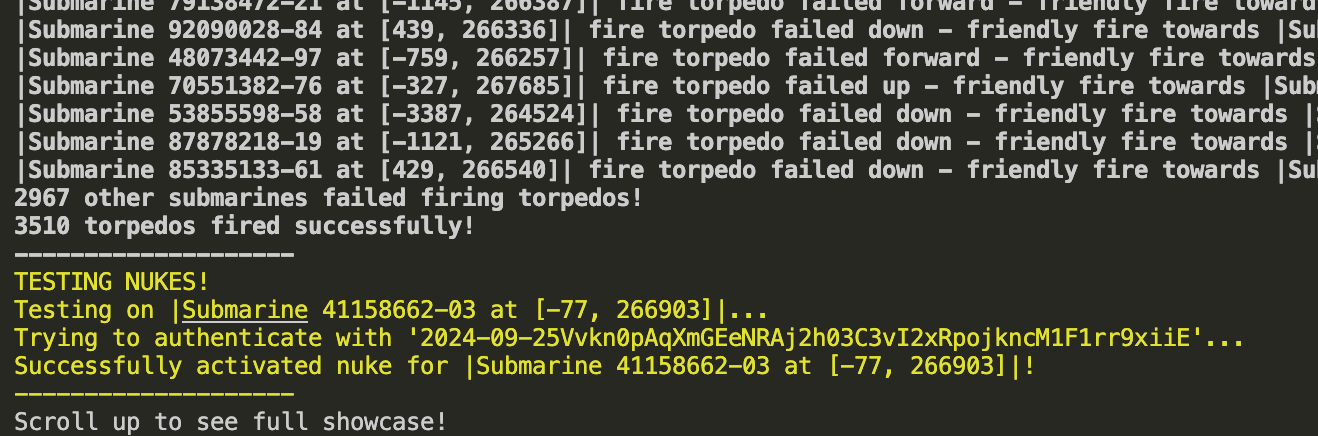
## Bilaga 1



## Bilaga 2



## Bilaga 3



1. Andrew P. Black, “Ole-Johan Dahl A.M. Turing Award Laureate”, A.M. Turing, 2001, <https://amturing.acm.org/award_winners/dahl_6917600.cfm> (åtkomstdatum 24 september 2024) [↑](#footnote-ref-1)