1 Introduksjon

HarbNet rammeverket er ett rammeverk for simulere en kontainer havn. Rammeverket lar brukeren opprette skip, kontainere som skal brukes i simuleringen samt definere antall og størelser på brygger tilknyttet havnen og lagringskapasiteten til havnen. Brukeren kan spesifisere effektiviteten til de forskjellige akrivitetene som foregår på havnen og rammeverket lar brukeren hente ut historisk data om alt som har skjedd i havnen under simuleringen.

Vår ambisjon er at rammeverket skal være komplekst nok til å kunne simulere en stor del av aktiviteten som foregår på en kontainerhavn, men intuetivt nok til at brukere kan bruke det uten å måte lese ekstern dokumentasjon.

Rammeverket har gått igjenom flere iterasjoner under utviklingen og har økt i omfang med hver iterasjon. Videre i dette dokumentet har vi dokumentert rammeverkets tilstand i hver iterasjon.

2 Iterasjon 1

2.1 Krav

Kravene til API-et er i stor del gitt av oppgaven. En bruker av API løsningen skal kunne opprette og utføre en simulering av en havn i en gitt tidsperiode som er bestemt av brukeren selv. Brukeren skal kunne definere skipene som ankommer havnen, kunne justere antallet og typen skip som ankommer kaia, samt justere seilasene for hvert enkelt skip i form av hvor mange dager skipets seilas tar eller om det er en enkeltseilas. Bruker skal kunne definere hvilke skipstyper som passer til hver kaiplass. I tillegg skal brukeren kunne hente ut historikk fra simuleringen.

Når det kommer til skipstypene som er brukt har vi bestemt at vi deler det opp i 3 kategorier basert på størrelsen på skipet. Liten, middels og store lasteskip. Hvert av

skipstørrelsene skal ha forskjellig kapasitet når det kommer til antallet på containere de kan ha ombord og maks vekten på lasten de kan laste.

I tillegg til disse kravene gitt fra oppgaveteksten har vi definert noen tilleggskrav som denne simuleringen skal ta hensyn til. Brukeren skal kunne definere hvor mange shipping kontainere som ett skip har ombord ved første ankomst til kai.

Oppgaven ber bare om at man skal kunne hente ut historikk om hvor kontainere var lokalisert på de forskjellige tidspunktene, men vi bestemte oss om å utvide dette kravet til også å inkludere hvor alle skip befinner seg på ett gitt tidspunkt. Ved å få historikk om både skip og kontainere vil man kunne få ett bilde av hvordan hele kaia har operert gjennom simuleringen.

Kravspesifikasjon

1. Sette opp havnestruktur (*Oppfyller minst Krav 1 i oppgaven*)

1.1 Det skal kunne konfigureres antall kaiplasser, losseplasser og venteplasser.

1.2 Det skal kunne konfigureres antall containerplasser og plassering av containerplasser.

1.2.1 Det skal kunne konfigureres flyttingen av container fra skip til containerplass.

2. Konfigurasjon av seilinger (*Oppfyller minst krav 2 fra oppgaven*)

2.1 Det skal kunne settes opp enkeltseilinger som utføres på et gitt tidspunkt.

2.2 Det skal kunne settes opp gjentagende seilinger som utføres daglig og ukentlig.

2.3 Det skal kunne konfigureres hvilken skipstype som skal brukes på en seiling.

3. Kaiplasser (*Oppfyller minst krav 3 fra oppgaven*)

3.1 Det skal kunne konfigureres hvilke skipstyper som tillates på en kaiplass.

4. Historikk (*Oppfyller minst krav 4 og krav 5 fra oppgaven*)

4.1 Bruker skal kunne hente ut full historikk med hendelser for et skip

4.1.1 En hendelse skal inneholde skipets lokasjon

4.1.2 En hendelse skal inneholde dato og klokkeslett

4.1.3 Historikken skal inneholde både start og slutt av aktivitet

4.2 Bruker skal kunne hente ut informasjon om hvor en container var lokalisert

5. Simulering

5.1 Det skal kunne settes antall skip av de forskjellige størrelsestypene

5.2 Det skal kunne settes start –og sluttdato på simulering

2.2 Første API-utkast

I første utkastet av API-et tenkte vi at det bare skulle være en API klasse som definerte hele API-et brukeren hadde tilgang til. Denne API klassen skulle gi brukeren tilgang til å konfigurere alle de nødvendige spesifikasjonene for simuleringen de ønsket å gjennomføre så skulle denne API klassen videreformidle den nødvendige informasjonen til de bakenforliggende klassene som brukeren ikke hadde tilgang til.

Tanken her var at brukeren ikke skulle trenge å forholde seg til noe av den konkrete implementasjonen av simuleringen i den hensikt å minimere muligheten bruker hadde for å gjøre feil ved oppsett av en simulering.

Etter litt diskusjon gikk vi senere fra denne måten å sette opp API-et på til fordel for at bruker får tilgang til flere av de bakenforliggende klassene slik som Ship, Harbor og Container slik at bruker kan sette opp simuleringen på en mer vanlig programmatisk måte ved å opprette objekter og sende inn variabler slik bruker sannsynligvis er vant med fra vanlig programmering.

Vi brukte fremdeles dette første utkastet til å vurdere hvilke metoder, funksjoner og klasser bruker burde ha tilgang til å det fullstendige API-et som vi utviklet senere, selv om vi endte med å droppe og endre en god del av metodene under utvikling etter testing.

Enkelte metoder slik som Api.Create() ga heller ikke mening med den nye strukturen vi endte opp med å ha på API-et da denne metoden ble erstattet av konstruktørene for de forskjellige klassene.
Andre metoder eksisterer fremdeles i det nye API-et, men har endret navn. Slik som ShipTurnAroundTimeInHours som har endret navn til RoundTripInDays. Dette kom som et resultat av den nye strukturen da det sistnevnte forhåpentligvis er litt mer forståelig for brukeren og å få tiden i dager gir litt mer mening utafra tidsperspektivet simuleringene utspiller seg. Å få antall dager ett skip sin seilas tar er sannsynligvis litt mer oversiktlig en å få det i ett hundretalls timer.

Under er det første utkastet til interfacet for API klassen vi originalt planla å utvikle:

```
namespace HarbFramework
{
    0 references
    internal interface IApi
    {
        0 references
        public void create(); //Oppretter simuleringen
        0 references
        public String setName(String name); //returnerer navnet
        0 references
        public Guid createPort(DockSize portSize); //returnerer Guid til porten som ble laget
        0 references
        public Guid createShip(ShipSize shipSize); //Returnerer Guid til skipet som ble laget
        0 references
        public Guid createShip(ShipSize shipSize, DateTime startDate); //Returnerer Guid til skipet som ble laget
        0 references
        public Guid createShip(ShipSize shipSize, DateTime startDate, int roundTripInDays); //Returnerer Guid til skipet som ble laget
        0 references
        public Boolean removeShip(Guid shipId); //Returnerer True hvis skipet ble fjernet, false ellers.
        0 references
        public Boolean removePort(Guid PortId); //Returnerer True hvis havna ble fjernet, false ellers.
        0 references
        public void simulationDuration(DateTime startDate, DateTime endDate);
        0 references
        public void run(); //starte simuleringen
        0 references
        public ICollection<Event> getContasinerHistory(Guid containerID);
        0 references
        public ICollection<Event> getShipHistory(Guid shipID);
        0 references
        public ICollection<Event> getWeatherHistory();
        0 references
        public ICollection<Guid> getContainerIDs();
        0 references
        public ICollection<Guid> getShipIDs();
        0 references
        public void printHistoryToConsole();
        0 references
        public void printHistoryToFile(String fileName);
        0 references
        public int getShipTurnaroundTimeInHours(); //får int med tid i timer.
        0 references
        public int getShipTurnaroundTimeInHours(DateTime startDate, DateTime endDate); //får int med tid i timer beregnet mellom start og sluttdato
        0 references
        public int getContainerTurnAroundTimeInHours();//får int med tid i timer.
        0 references
        public int getContainerTurnAroundTimeInHours(DateTime startDate, DateTime endDate); //får int med tid i timer beregnet mellom start og sluttdato
        0 references
        public int getAverageLoadTimeInHours(); //får int med tid i timer.
        0 references
        public int getAverageLoadTimeInHours(DateTime startDate, DateTime endDate);  //får int med tid i timer beregnet mellom start og sluttdato
        0 references
        public int getAverageQueuingTimeInHours(); //får int med tid i timer skip må vente i kø.
        0 references
        public int getAverageQueuingTimeInHours(DateTime startDate, DateTime endDate); //får int med tid i timer skip må vente i kø.
        0 references
        public int getContainersMoved(); //Antall containere flyttet i løpet av simuleringen
        0 references
        public int getContainersMoved(DateTime startDate, DateTime endDate);
        0 references
        public int getNumberOfDockings(); //Antall skip docket i løpet av simuleringen
        0 references
        public int getNumDockings(DateTime startDate); //Antall skip docket i løpet av simuleringen
    }
}
```

2.3 Dokumentasjon av fullstendig API

HarbFramework API-et er definert i 5 forskjellige Interface. Interfacene definerer kontrakten for klassene og metodene som skal være tilgjengelig for brukere av API-et. Brukeren kan opprette objekter av klassene som er tilknyttet hvert av de forskjellige API interfacene og bruke metodene som interfacene definerer. Klassene vil ha flere underliggende metoder en det interfacene definerer, men disse metodene er ikke tilgjengelig gjennom API-et og ikke metoder brukeren skal trenge å forholde seg til.

IEvent

IEvent definerer APIet for event objekter. Disse objektene holder informasjon om historikken til skip og kontainere som brukes i simuleringen. Hvert objekt holder informasjon om statusen til ett skip eller kontainer på ett gitt tidspunkt. Objektene opprettes hver gang en kontainer eller skip endrer status (arbeidsoppgave) fra en oppgave til en annen. For eksempel vil ett event objekt bli opprettet når ett skip går fra å laste av kontainere (Unloading) til å laste på nye kontainere (Loading). Disse event objektene vil bli lagt til hvert enkelt skips eller kontainers historie slik at de kan hentes ut for å få en oversikt over hva skipet eller kontaineren har gjort gjennom forløpet til simuleringen.

IEvent definerer følgende metoder:

public Guid Subject { get; }

Denne metoden returnerer Guid til den spesifikke kontaineren eller skipet som eventen er knyttet til. Hvert skip og kontainer har en unik ID som identifiserer det skipet eller kontaineren. Variabelen subjekt identifiserer dette skipet eller kontaineren slik at vi kan vite hvilke skip eller kontainer eventen handler om.

public Guid SubjectLocation { get; }

Denne metoden returnerer Guid til lokasjonen som Subjectet til eventen befinner seg i det øyeblikket event objektet ble opprettet. SubjectLocation kan for eksempel være skipet som en kontainer befinner seg på. En havnplass som ett skip befinner seg på eller en kontainerplass på havna som en kontainer befinner seg på. De forskjellige lokasjonene er som følger:

For skip: Anchorage, Transit, LoadingDock, ShipDock.

For containere: Ship, ContainerSpace.

Alle disse forskjellige lokasjonene har sin egen ID. Hvert unike skip, loading dock, ShipDock vil ha sin egen unike ID.

public DateTime PointInTime { get; }

Denne metoden returnerer ett DateTime objekt med tidspunktet som eventen ble opprettet. Eventer opprettes når Subjectet de omhandler endrer status. For eksempel hvis en kontainer går fra å bli lastet av ett skip (Unloading) til å bli lagret på et containerspace på havnen (InStorage). Du kan dermed se ut ifra PointInTime variabelen og Statusen når Subjectet fikk sin nåværende status.

public Status Status { get; }

Denne metoden returnerer en Status enum som beskriver statusen Subjectet har fått idet Event objektet opprettes. Status enumen gir informasjon om hva som skjer med, eller hva subjectet gjør i det øyeblikket Event objektet blir opprettet.

Statusene som ett subject kan ha er som følger:

For Ship:

- **DockingToLoadingDock**
  – Skipet legger til havn ved en losseplass hvor skipet kan laste av og på kontainere
- **DockingToShipDock**
  – Skipet legger til havn ved en vanlig kaiplass
- **Undocking**
  – Skipet legger fra havn
- **Loading**
  – Skipet laster kontainere fra havnen til sitt eget lager
- **Unloading**
  – Skipet lasterkontainere fra sitt eget lager til havnen
- **Transit**
  – Skipet er i transit ute på havet
- **Anchoring**
  – Skipet ligger til anker hvor det venter på å få en ledig kaiplass

For Container:

- **Loading**
  – Kontaineren blir lastet ombord på ett skip
- **Unloading**
  – Kontaineren blir lastet av ett skip og til en ledig ContainerSpace på havnen
- **Transit**
  – Kontaineren er ombord på ett skip som er i Transit på havet
- **InStorage**
  – Kontaineren er lagret på en ContainerSpace på havnen

IContainer

IContainer definerer APIet for Container klassen. Objekter av denne klassen holder representerer en container i simuleringen og holder på relevant informasjon om denne kontaineren slik som IDen til kontaineren, historikken til kontaineren, størrelsen på containeren, vekten til kontaineren og containerens nåværende lokasjon. Objektene opprettes typisk i skips lasterom samtidig som nye skip opprettes.

Det er vert å merke seg at konstruktøren for container klassen ikke er tilgjengelig for brukere av APIet, istede opprettes containere samtidig med opprettelse av skip. Dette er for å minimere arbeidet brukere må gjøre for å kunne starte en ny simulering.

Følgende metoder er gitt i IContainer interfacet:

public Guid ID { get; }
Denne metoden returnerer ett Guid objekt som holder på den unike IDen til containeren. IDen kan brukes til å identifisere og finne spesifikke kontainere i andre metodekall.

public IList<Event> History { get; }
Denne metoden returnerer en IList med Event objekter (Se beskrivelse av IEvent for nærmere forklaring av disse objektene). Denne listen inneholder ett Event objekt for hver gang kontaineren har endret Status (se beskrivelse av Status enum). Listen er organisert fra første status endring til siste, der første statusendring har index 0 i listen og den siste status endringen har den siste indexen i listen. Man kan slik få en komplett

oversikt over hva en container har gjort og hvor den har befunnet seg til enhver tid gjennom hele simuleringen.

public ContainerSize Size { get; }

Denne metoden returnerer en ContainerSize enum (se beskrivelse av Container Size enum for nærmere informasjon) som definerer størrelsen på kontaineren. Kontainerens vekt er gitt ut ifra størrelsen og forskjellige container størrelser krever forskjellig størrelse på lagerplassen den kan få tildelt på havnen.

public int WeightInTonn { get; }

Denne metoden returnerer en int med vekten til containeren gitt i tonn. Vekten på en kontainer kan variere basert på størrelsen og skip har en samlet maksvekt på kontainere de kan laste ombord.

public Guid CurrentPosition { get; }

Denne metoden returnerer ett Guid objekt som holder på IDen til objektet som i dette øyeblikket lagrer containeren. Dette kan være IDen til ett skip kontaineren er lagret ombord hos, eller det kan være en lagerplass på havnen som kontaineren er lagret på mens den venter på at ett skip skal laste den ombord.

public Status GetStatus();

Denne metoden returnerer en status enum (Se enum status for mer informasjon) som indikerer statusen til skipet på det tidspunktet metoden blir kalt.

IShip

IShip definerer APIet til skip objekter. Ett skip objekt definerer ett enkelt skip som skal brukes i simuleringen. Alle skip i simuleringen er lasteskip som kan laste kontainere. Skip kan komme i flere størrelser (se ShipSize enum for mer informasjon). Størrelsen til skipet vil definere maksvekten av lasten ett skip kan ha ombord, antall kontainere den har plass til i lageret sitt og hvilke størrelse på havneplass skipet krever for å kunne legge til kai.

Skip kan enten opprettes for en enkeltseilaser eller for kontinuerlige seilaser. For enkeltseilaser vil skipet ankomme havnen ved en gitt dato, legge seg til en ledig lastehavn (loading dock), laste av lasten sin for deretter og flytte seg til en vanlig skipshavn (ship dock). Der vil den ligge ut simuleringen.

For ett skip med kontinuerlig seilas kan det settes antall dager det tar fra ett skip drar fra kai til skipet er tilbake igjen ved havnen. Skipet vil deretter legge seg til en ledig lastehavn (loading dock), laste av lasten sin, laste på ny last for deretter og legge fra kai og dra ut på havet igjen. Skipet vil repetere denne syklusen for hele lengden av simuleringen.

```
public Ship (ShipSize shipSize, DateTime StartDate, bool IsForASingleTrip, int roundTripInDays, int numberOfcontainersOnBoard)
```

Denne konstruktøren tar imot 5 attributter:

- **shipSize:** tar imot en enum ShipSize (se seksjon om Enum Shipsize for mer informasjon) som bestemmer størrelsen på skipet. Størrelsen på skipet vil definere maksvekten på den totale lasten ett skip kan ha ombord og antallet kontainere ett skip har plass til i lagerrommet, i tilleg til hvor stor kaiplass skipet trenger når det legger til kai.
- **startDate**: tar imot ett DateTime objekt som definerer tidspunktet hvor skipet først ankommer havnen. Skipet vil ankomme havnen for å laste av sin første last på datoen gitt i attributtet.
- **isForASingleTrip:** Tar imot en boolsk verdi (true/false) som sier om skipet er et enkeltseiling-skip eller ikke.
- **roundTripInDays:** tar imot en int verdi som definerer hvor mange dager det tar fra ett skip legger fra kai, drar ut på havet til det er tilbake igjen med ny last.
- **numberOfcontainersOnBoard:** tar imot en int verdi og definerer hvor mange kontainere skipet har ombord første gangen skipet ankommer havnen. Denne attributtene definerer altså ikke maks antall kontainere ett skip kan ha ombord ettersom dette defineres av skipets størrelse, men antallet containere som skipet har i lasterommet ved første ankomst til havnen. Ved opprettelse av skip vil programmet forsøke å generere ett nogen lunde likt antall kontainere av hver

størrelse i skipets last. Man kan dermed forvente at hvis man sender inn en verdi av 3 til denne attributtene vil skipet starte med 1 liten kontainer, 1 medium kontainer og 1 stor kontainer ombord.

I tillegg er disse metodene gitt i IShip interfacet:

public Guid ID { get; }

Denne metoden returnerer et Guid objekt som holder på den unike IDen til skipet. Denne IDen kan brukes til å identifisere det spesifikke skipet i andre metodekall.

public ShipSize ShipSize { get; }

Denne metoden returnerer ett ShipSize enum (se Enum Shipsize for mer informasjon) som definerer størrelsen på skipet. Størrelsen på skipet vil definere maksvekten på den totale lasten ett skip kan ha ombord og antallet kontainere ett skip har plass til i lagerrommet, i tilleg til hvor stor kaiplass skipet trenger når det legger til kai.

public DateTime StartDate { get; }

Denne metoden returnerer ett DateTime objekt som gir tidspunktet for første gang skipet ankom havnen.

public int RoundTripInDays { get; }

Denne metoden returnerer en int verdi som sier hvor mange dager det tar fra skipet legger fra kaien med ny last til den er tilbake igjen til havnen. Altså tiden det tar for skipet å dra ut på havet for så å komme tilbake til havnen igjen.

public Guid CurrentLocation { get; }

Denne metoden returnerer Guid til den nåværende possisjonen til skipet.
Posisjonene ett skip kan befinne seg er:

- **Transit**
  – Skipet er på havet.
- **Anchorage**
  – Skipet ligger til anker og venter på å få en ledig kaiplass til å kunne laste av lasten.
- **LoadingDock**
  – Skipet ligger til en laste kai hvor den laster av og/eller på kontainere.
  – Hver enkelt loadingDock har sin unike Guid.

- **ShipDock**
  – Skipet er ferdig med enkelt-seilasen sin og ligger nå til kai.
  – Hver enkelt shipDock har sin unike Guid.

public IList<Event> History { get; }
Denne metoden returnerer en IList med Event objekter (Se beskrivelse av IEvent for nermere forklaring av disse objektene). Denne listen inneholder ett Event objekt for hver gang skipet har endret Status (se beskrivelse av Status enum). Listen er organisert fra første status endring til siste, der første statusendring har index 0 i listen og den siste status endringen har den siste indexen i listen. Man kan slik få en komplett oversikt over hva en skipetr har gjort og hvor det har befunnet seg til enhver tid gjennom hele simuleringen.

public IList<Container> ContainersOnBoard { get; }
Denne metoden returnerer en IList med Container objekter som representerer containerene som skipet har i sitt lasterom i nåværende øyeblikk.

public int ContainerCapacity { get; }
Denne metoden returnerer en int verdi som representerer hvor mange containere skipet har plass til i lasterommet sitt. En kontainer vil ta opp 1 plass i lasterommet uavhengig av størrelsen på kontaineren.

public int MaxWeightInTonn { get; }
Denne metoden returnerer en int verdi som representerer hva maksvekten i tonn skipet kan være før den synker. Denne maksvekten inkluderer vekten på selve skipet pluss vekten på lasten som skipet har ombord. Hvor stor maksvekt ett skip kan ha er basert på skipets størrelse. Større skip vil ha en større maksvekt. Maksvekten til skipet kan ikke overskrides.

public int BaseWeightInTonn { get; }
Denne metoden returnerer en int verdi som representerer vekten i tonn skipet er når den ikke har noen last ombord. Altså når lasterommet er tomt. Denne vekten medgår i den nåværende vekten til skipet (CurrentWeightInTonn). Base vekten til skipet er basert på størrelsen til skipet. Større skip vil ha en større dødvekt.

public int CurrentWeightInTonn { get; }

Denne metoden returnerer en int verdi som representerer vekten i tonn som skipet i øyeblikket metoden kalles. Vekten kalkuleres med grunnlag på vekten skipet har når det er tomt (BaseWeightInTonn) + summen av vekten til alle kontainerne som skipet har i sitt lasterom. Dette tallet kan aldri overskride maksvekten skipet kan ha (MaxWeightInTonn).

IHarbor

IHarbor definerer APIet for harbor objekter. Ett harbor objekt representerer en enkelt havn som kan brukes i simuleringen. Hver simulering bruker kun en havn. Harbor er objektet som driver det meste av det som skjer i simuleringen. Den holder på informasjon om skipene som brukes i simuleringen og hvor skipene befinner seg. Den har også lastekaiplasser (loadingDocks), kaiplasser der skip kan ligge til havn over lengere perioder (ShipDocks) og lagerplasser for kontainere (ContainerSpaces). Skip vil ankomme havnen og laste av og på kontainere før de drat ut på havet igjen. Harbor objektet holder på all denne informasjonen om hvilke skip som er med i simuleringen og hvor de befinner seg.

Harbor objekter har en public konstruktør som ikke er gitt i interfacet.
Denne konstruktøren ser slik ut:

```
public Harbor(IList<Ship> listOfShips, int numberOfSmallLoadingDocks,
int numberOfMediumLoadingDocks, int numberOfLargeLoadingDocks,
int numberOfSmallShipDocks, int numberOfMediumShipDocks,
int numberOfLargeShipDocks, int numberOfSmallContainerSpaces,
int numberOfMediumContainerSpaces, int numberOfLargeContainerSpaces)
```

Denne konstruktøren tar 10 attributter som er som følger:

- **listOfShips:** Denne attributtene er en liste med Ship objekter av type IList<Ship> som skal brukes i simuleringen. Alle skip vil starte på ankerplassen mens de venter på å få komme til en ledig kaiplass for å kunne laste av lasten sin.
- **numberOfSmallLoadingDocks:** Denne attributtene tar en int verdi som angir hvor mange laste-kai-plasser som havnen har tilgjengelig for sip av størrelse Small (se

Enum Shipsize for mer info). LoadingDocks eller laste-kai-plasser er kaiplasser der skip kan legge til og laste av og på kontainere før de drar ut på havet igjen. Hver LoadingDock kan ta ett skip av tilsvarende størrelse som seg selv.

- **numberOfMediumLoadingDocks:** Tilsvarende numberOfSmallLoadingDocks bare for loading docks av medium skipstørrelse i stedet.
- **numberOfLargeLoadingDocks:** Tilsvarende numberOfSmallLoadingDocks bare for loading docks av Large skipstørrelse i stedet.
- **numberOfSmallShipDocks:** Denne attributten tar en int verdi som angir hvor mange kaiplasser havnen skal ha tilgjengelig for skip av størrelse Small (see Enum ship size for mer info) for å kunne ligge til kai når de ikke laster av og på kontainere. Det er typisk sett skip som bare gjør en enkeltseilas som benytter seg av denne typen kaiplasser. Ett skip av denne typen vil legge seg til en ledig ShipDock som matcher sin egen størrelse når den er ferdig med å laste av lasten sin. Der vil skipet ligge til simuleringen er ferdig.
- **numberOfMediumShipDocks:** Tilsvarende numberOfSmallShipDocks bare for docks av størelse medium istede.
- **numberOfLargerShipDocks:** Tilsvarende numberOfSmallShipDocks bare for docks av størelse large istede.
- **numberOfSmallContainerSpaces:** Denne attributten tar imot en int verdi som definerer hvor mange lagerplasser (container space) for kontainere av størrelse Small (see Enum ContainerSize for mer info) havnen skal ha tilgjengelig. En containerSpace kan lagre en kontainer av tilsvarende størelse som seg selv. For eksempel kan en smal container space lagre en smal container.
- **numberOfMediumContainerSpaces:** tilsvarende numberOfSmallContainerSpaces bare for Container Spaces av størrelse medium i stedet.
- **numberOfLargeContainerSpaces:** tilsvarende numberOfSmallContainerSpaces bare for Container Spaces av størrelse large i stedet.

I tillegg er disse metodene gitt i interfacet:

public Guid ID { get; }

Denne metoden returnerer ett Guid objekt som definerer den unike IDen til Harbor objektet. IDen kan brukes for å identifisere og skille harbor objekter fra hverandre.

public IDictionary<Guid, bool> LoadingDockIsFreeForAllDocks();

Denne metoden returnerer en liste sortert i key-value par der Key er ett Guid objekt som representerer den unike IDen til en enkel laste-kai-plass (loading dock), og value er en bool verdi som har verdien True hvis laste-kai-plassen er ledig og False hvis den er opptatt. Listen inneholder slike key-value par for alle laste-kai plassene på havnen og kan brukes for å få en oversikt over hvilke kaiplasser som er ledig eller opptatt på tidspunktet metoden blir kalt.

public bool LoadingDockIsFree(Guid dockID);

Denne metoden tar imot en Guid til en LoadingDock (laste-kai-plass) og returnerer en bool verdi som er true hvis kaiplassen er ledig og false hvis den er opptatt. Den kan dermed brukes for å sjekke ledigheten til en enkelt laste-kai-plass.

public Status GetShipStatus(Guid ShipID);

Denne metoden tar imot ett Guid objekt med IDen til ett enkelt ship objekt og returnerer en Status enum (se enum ship status for mer informasjon) som representerer statusen skipet har når metoden blir kalt.

Ett skip kan ha følgende statuser:

- **Transit**
  – Skipet er på havet.
- **Anchorage**
  – Skipet ligger til anker og venter på å få en ledig kaiplass til å kunne laste av lasten.
- **LoadingDock**
  – Skipet ligger til en lastekai hvor den laster av og/eller på kontainere.
  – Hver enkelt loadingDock har sin unike Guid.
- **ShipDock**
  – Skipet er ferdig med enkelt-seilasen sin og ligger nå til kai.
  – Hver enkelt shipDock har sin unike Guid.

public IDictionary<Guid, bool> ShipDockIsFreeForAllDocks();

Denne metoden returnerer en value-key liste av typen IDictionary<Guid, bool> der key verdiene er objekter av typen Guid som representerer de unike IDene til hver enkelt shipDock (Kaiplass der skip ligger til kai uten å kunne laste av og på containere) og valuene er bool verdier som er true hvis docken er ledig og False hvis den er opptatt.

public IDictionary<Ship, Status> GetStatusAllShips();

Denne metoden returnerer en key-value liste av typen IDictionary<Ship, Status> der key verdien er ett Ship objekt og value verdien er en Status enum (se Enum status for mer info) som beskriver statusen skipet har i øyeblikket metoden blir kalt. Man kan på denne måten få oversikt over hva hvert enkelt skip i simuleringen gjør på tidspunktet metoden blir kalt.

public Guid AnchorageID { get; }

Denne metoden returnerer ett Guid objekt som holder på den unike IDen til Anchorage. Anchorage er en posisjon som skip kan befinne seg i. Den representerer når skip ligger til anker og venter på å få en ledig kaiplass ved havnen.

public Guid TransitLocationID { get; }

Denne metoden returnerer ett Guid objekt som holder på den unike IDen til Transit lokasjonen. Transit er en posisjon som ett skip kan befinne seg i. Den representerer når skip er på havet.

ILog

ILog definerer APIet til Log klassen. Log klassen kan brukes til å lage Log objekter som simuleringen bruker for å bokføre statusen til alle skip og kontainere i simuleringen på ett gitt tidspunkt. Simuleringen vil hvert døgn opprette en nytt Log objekt som forteller om status og plassering som alle skip og containere har i det øyeblikket logobjektet ble opprettet.

ILog interfacet definerer følgende metoder:

public DateTime Time { get; }

Denne metoden returnerer ett DateTime objekt som angir tidspunktet i simuleringen logobjektet ble opprettet. Denne verdien angir altså tidspunktet som alle skip og containere hadde status og plassering som er angitt i log objektet.

public IList<Ship> ShipsInAnchorage { get; }
Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som lå til anker på det tidspunktet hvor Log objektet ble opprettet.

public IList<Ship> ShipsInTransit { get; }
Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som var på havet på det tidspunktet Log objektet ble opprettet.

public IList<Ship> ShipsDockedInShippingDocks { get; }
Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som lå til en laste-kai (Loading dock) og lastet av og på containere på det tidspunktet Log objektet ble opprettet.

public IList<Ship> ShipsDockedInShippingDocks { get; }
Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som lå til en kaiplass for skip som ikke laster av eller på containere på det tidspunktet Log objektet ble opprettet

public IList<Container> ContainersInHarbour { get; }
Denne metoden returnerer en liste av type IList<Container> som inneholder alle containere som lå lagret til land på havnen i på det tidspunktet Log objektet ble opprettet.

public void PrintInfoForAllShips();
Denne metoden printer informasjon om alle skip og hvor de befinner seg til konsoll.

public void PrintInfoForAllContainers);
Denne metoden skriver ut informasjon om alle containere og hvor de befinner seg til konsoll.

ISimulation

ISimulation definerer APIet til Simulation klassen. Simulation klassen kan brukes for å lage Simulation objekter som brukes for å kjøre simuleringen til en havn.

Simulation klassen har en public konstruktør som ikke er definert i interfacet. Denne konstruktøren ser slik ut:

```
public Simulation (Harbor harbor, DateTime simulationStartTime,
DateTime simulationEndTime)
```

Konstruktøren har 3 attributter:

- **harbor**: tar inn et Harbor-objekt som er havnen som skal brukes i simuleringen.
- **simulationStartTime**: tar imot ett DateTime objekt som angir tidspunktet som er første dag simuleringen skal starte på.
- **simulationEndTime**: tar imot ett DateTime objekt som angir tidspunktet som simuleringen skal kjøre til. Altså tidspunktet simuleringen skal stoppe.

I tillegg angir interfacet følgende metoder:

public IList<Log> History { get; }
Denne metoden returnerer en IList<Log> med log objekter (Se ILog for mer informasjon om disse). Denne listen inneholder ett Log objekt for hver dag simuleringen har pågått. Nye logg objekter opprettes og legges til listen klokken 12 om natten hver 24 time. Listen er organisert sekvensielt slik at det første log objektet som blir opprettet ligger først på index 0 og det siste log objektet som ble opprettet ligger på den høyeste indexen i listen. Man kan slik ved hjelp av denne listen få en komplett oversikt over posisjonen og statusen til alle skip og kontainere gjennom hele forløpet til simuleringen.

public IList<Log> Run();
Denne metoden brukes til å starte simuleringen. Når metoden kalles vil simuleringen starte og kjøre fra tidspunktet gitt i konstruktørens start time og til tidspunktet gitt i konstruktørens end time. Den returnerer så en IList<Log> med log objekter som forteller om historien til hvordan simuleringen har utspilt seg.

public void PrintShipHistory();

Denne metoden printer til konsoll hele historien til alle skip for den forrige simuleringen som har blitt utført.

public void PrintContainerHistory();

Denne metoden printer til konsoll hele historien til alle kontainere for den forige simuleringen som har blitt utført.

Enum ContainerSize

Enumen ContainerSize er tilgjengelig gjennom APIet. Enumen angir størrelsen til en container. Alle containere har en container size enum som representerer størrelsen kontaineren kan ha.

Størrelsene er som følger:

- None = 0,
- Small = 10,
- Medium = 15,
- Large = 20

Verdiene gitt etter hver av størrelsene er vekten på containeren i tonn. For eksempel vil en Large container veie 20 tonn.

Enum ShipSize

Enumen ShipSize er tilgjengelig gjennom APIet. Enumen angir størrelsene skip kan ha. Alle skip har en ShipSize enum som definerer størrelsen til skipet.

Størrelsene er som følger:

- None = 0,
- Small,
- Medium,
- Large

Enum Status

Enumen Status er tilgjengelig gjenom APIet. Enumen angir statuser som skip og kontainere kan ha gjennom simuleringen. Hver status representerer noe ett skip eller kontainer kan gjøre på ett gitt tidspunkt.

Statusene er som følger:

- None = 0
- **DockingToLoadingDock**
  – Skip legger seg til en laste-kai
- **DockingToShipDock**
  – Skip legger seg til en kai hvor de ikke kan laste.
- **Undocking**
  – Skip legger fra kai
- **Loading**
  – Skip laster ombord containere eller en container lastes ombord på ett skip.
- **Unloading**
  – Skip laster av containere eller en container lastes av ett skip.
- **Transit**
  – Skip er på havet.
- **InStorage**
  – Container er lagret på land på havna
- **Anchoring**
  – Skip ligger til anker og venter på en ledig kaiplass
- **UnloadingDone**
  – Skip er ferdig med å laste av lasten sin.
- **LoadingDone**
  – Skip er ferdig med å laste på ny last

2.4 Dokumentasjon og diskusjon av implementasjon

Implementasjon av Simulering:

Vårt rammeverk for simulering av en havn er delt opp i 4 hovedklasser. Harbor, som holder på all informasjonen om havnen, hvor skip er lokalisert og hvor kontainere er lagret. Ship som holder på informasjon om hvert enkelt skip, Container som holder på informasjon om hver enkelt kontainer og Simulation, som driver simuleringen av hele havnen.

I prosjektet fikk vi oppgave å simulere en båthavn, Når vi fikk oppgaven begynte vi å tenke igjennom hva som måtte til for å få dette til å fungere som en ekte havn. For å få en simulering av en havn til å fungere er det viktig at man legger til alle de delene en havn består av. For at dette skulle fungere fant vi ut at vi måtte ha et skip, det skipet måtte ha mulighet til å vente i en kø, ankre, losse, docke og undocke fra havnen samt load og unload av kontainere.

I klassen har vi valgt å bruke variablene startTime, currentTime, og endTime.
Vi diskuterte litt om hvordan vi skulle kunne starte og stoppe en simulering, vi fant ut av at det beste var at en bruker skulle kunne sette start tiden de selv ønsket og hvor lenge simuleringen skulle kjøre. Vi måtte også ha en måte for simuleringen å vite hvilket klokkeslett/tid det var i simuleringen for logging.
det er også en liste som inneholder log objekter som blir kalt History denne listen er en iList fordi at i kapittel 8 blir det diskutert om lister og hvordan man skal bruke de.

- "DO NOT use ArrayList or List<T> in public APIs (...) Instead of using these concrete types, consider Collection<T>, IEnumerable<T>, IList<T>, IReadOnlyList<T>, or other collection abstractions. " (Cwalina; Jeremy; Brad. et al., 2020 s. 295).

Når det kom til selve metoden som skulle kjøre simuleringen fant vi ut av at navnet “Run” var det som ga mest mening, det er lett for en bruker å forstå hva den metoden gjør, selv med lite kunnskap innenfor programmering. I denne metoden har vi valgt å

først lage ett nytt log objekt.
Når selve while løkken begynner valgte vi å bruke de parameterne som ble diskutert currentTime og endTime. Det gir While løkken en exit-condition som kommer an på hvor lenge en bruker ønsker at simuleringen skal kjøre.

I konstruktøren til en harbor har vi lagt til en iList som inneholder alle skip i simuleringen. Grunnen til dette er at vi skal ha tilgang på alle skipene som blir lagt inn i simuleringen. I while løkken har denne listen blitt brukt mye for å gå igjennom hvert skip. Vi fant ut at dette var en god måte å løse det med å få tak i skip og statusene de har, for eksempel Loading - skip laster ombord containere eller en container lastes ombord på ett skip og Unloading - skip laster av containere eller en container lastes av ett skip. Statusene har vi valgt å sette som enum, vi kom frem til at det å lage konstante verdier som ikke kan endres var det som ga mest mening, det ga oss lettlest og kompakt kode.

I skip klassen har vi også metoden AddHistoryEvent. Denne metoden logger currentTime, en Id(harborId, shipId,anchorageId etc) og statusen til skipet. Navn valget på metoden var også noe vi diskuterte, ettersom at boken sier at alle events må være navngitt ett verb :

- “DO name events with a verb or a verb phrase. Examples include Clicked, Painting, and DroppedDown.” (Cwalina; Jeremy; Brad. et al., 2020 s. 77)
- “DO give events names with a concept of before and after, using the present and past tenses, such as Closing and Closed. For example, a close event that is raised before a window is closed would be called Closing, and one that is raised after the window is closed would be called Closed.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 77)
- DO NOT use “Before” or “After” prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 77).

I simuleringen diskuterte vi hvordan vi skulle håndtere at det som skal gjøres med og av skipene må skje over tid, vi ville unngå at alt kjørte uten noen stopp. Vi fant ut at vi

skulle lage en variabel som het NextStepCheck. Denne ville se om ett skip hadde blitt gjort noe med i en runde i while løkken.
Senere kom vi frem til at navngivningen ikke var god, det var vanskelig å vite hva denne variabelen representerte og vi endret den derfor til HasBeenAltered, for bedre lesbarhet og forståelighet. Dette blir også diskutert i boken:

- "DO choose easily readable identifier names.
  For example, a property named HorizontalAlignment is more English readable than AlignmentHorisontal." (Cwalina; Jeremy; Brad. et al., 2020 s. 52)
- "Do favor readability over brevity. The property name CanScrollHorizontally is better than ScrollableX(an obscure refernce to the X-axis)"
  (Cwalina; Jeremy; Brad. et al., 2020 s. 52)

Implementasjon av API:

Gjennom utvikling av prosjektet hadde vi flere diskusjoner om hvordan brukeren skulle få tilgang til de forskjellige elementene i simuleringen. Vi kom etterhvert frem til at det var bedre om brukeren kunne opprette objekter av noen av klassene som brukes i simuleringen, henholdsvis Harbor, Container og Ship klassene. Dette i ett håp om at det skulle bli mer intuitivt for brukeren å få tilgang til disse klassene og kunne sette opp simuleringen ved å opprette objekter av skipene som skulle være med i simuleringen og havnen som skulle brukes i simuleringen.

Majoriteten av funksjonene og en stor del av variablene vil fremdeles være utilgjengelig for brukeren ettersom vi anser mesteparten av den bakenforliggende logikken av hvordan selve simuleringen kjøres for ikke å være særlig relevant for brukeren.

Brukeren burde i hovedsak bare forholde seg til den informasjonen som skal sendes inn før simuleringen starter og kunne få ut informasjon om hvordan simuleringen har utspilt seg. Logikken som styrer simuleringen underveis eller variabler som bare brukes for at simuleringen skal håndtere dataene riktig underveis anser vi ikke som noe brukeren skal trenge å forholde seg til.

For at brukeren skal kunne lese informasjonen om hvordan simuleringen har vert underveis oppretter simuleringen en historikk i form av en liste med Log-objekter, der hvert Log-objekt holde på informasjonen om tilstanden til alle skip og kontainere i det

øyeblikket Logg objektet ble opprettet. Det ble derfor viktig og også gi brukeren tilgang til Logg klassen slik at de kan bruke denne listen med Historikk for å se hvordan simuleringen har utspilt seg for den relevante tidsperioden.

For at bruker skal kunne få nødvendig informasjon om tilstanden til simuleringen, i tillegg til å kunne sende inn data til havnen var det nødvendig å kunne gi ut og få inn lister med objekter. Både i form av tradisjonelle lister og key-value pairs.

Når det kommer til lister diskuterer læreboken dette i kapittel 8:

- "DO NOT use ArrayList or List<T> in public APIs (...). Instead of using these concrete types, consider Collection<T>, IEnumerable<T>, IList<T>, IReadOnlyList<T>, or other collection abstractions."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 295).

Enkelte listetyper som ArrayList i C# kan være litt problematisk for brukeren ettersom det kan være uklart hva slags objekter eller verdityper listene inneholder eller burde inneholde. Vi valgte derfor i de tilfellene hvor metodene returnerte lister til brukeren, eller brukeren sender inn lister til klassene å bruke IList<T>. Dette gjør at listene alltid blir sterkt typet og brukeren alltid har en kontrakt i forhold til hva slags funksjonalitet listene kan forventes å ha.

I tillegg har IList<T> mesteparten av funksjonaliteten vi så etter i lister og er i ett format som de fleste brukere burde være vant til fra andre programmeringspråk. IList<T> trenger ofte konkrete implementasjoner for å kunne brukes i den bakenforliggende logikken som kjøres i simuleringen. I de tilfeller hvor dette er nødvendig brukte vi IList<T> = new List<T>. Ettersom List blir begrenset til å følge kontrakten i IList vil den fremdeles følge angivningen til læreboken.

Når det kommer til lister med Key-Value pairs så har boken dette å si:

- "DO NOT use Hashtable or Dictionary<TKey,TValue> in public APIs. (...) Public APIs should use IDictionary, IDictionary <TKey, TValue>, or a custom type implementing one or both of the interfaces."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 296).

Hashtable i likhet med ArrayLister har svakheten at det kan være vanskelig for brukere å vite hva de kan forvente ut eller burde senne inn til denne listeformen.

Vi bestemte oss derfor for å bruke IDictionary<TKey, TValue> for alle key-value pair lister som sendes inn eller returneres ved hjelp av APIet. IDictionary har fordelen med at det er sterkt typet så brukeren alltid ved hva de kan forvente å få tilbake.
I enkelte tilfeller i den interne implementasjonen har vi valgt å bruke andre listetyper for Key-value pairs. Men dette er bare tilfeller der disse listene ikke skal returneres eller sendes inn av bruker.

Navngivning har vært mye diskutert gjennom prosjektoppgaven. Boken stiller en rekke krav til navngivning, men det viktigste av alt er at navngivningen er intuitiv for brukerne av APIet. Vanlige konvensjoner i programmering som camel og pascal casing har vi fulgt. Boken sier:

- “DO use PascalCasing for the names of namespaces, types, members, and generic type parameters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 43)
- “DO use camelCasing for parameter names.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 43).

Dette er vanlig konvensjoner i programmering og vi så ingen grunn til å ikke følge disse retningslinjene.

To andre retningslinjer fra boka når det kom til store bokstaver som ble relevant for prosjektet vårt var:

- “DO capitalize both characters of two-character acronyms, except the first word of a camel-cased identifier.” (Cwalina; Jeremy; Brad. et al., 2020 s. 45)
- “DO capitalize only the first character of acronyms with three or more characters, except the first word of a camel-cased identifier.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 46).

Dette er litt annerledes en hva vi var vant til fra andre programmeringsspråk hvor navngivning av akronymer ofte fulgte standardene for camelCasing og PascalCasing likt som ett hvilket som helst annet ord. Vi hadde derfor en del diskusjoner om hvilke tilfeller navn som "ID" skulle være i store bokstaver eller ikke ut ifra om de var variabel,

parameter eller metodenavn. Vi ønsket å følge boken her ettersom konvensjoner for navngivning for de spesifikke programmeringsspråket vi skriver i burde trumfe konvensjoner fra andre språk. Det gjør det lettere for brukeren å skjønne hvilke konvensjoner de kan forvente i navngivningen.

Når det kommer til generell navngivning av metoder og variabler er det vanskelig å vite helt på detaljnivå hva som er god og dårlig navngivning før det har blitt testet mot brukere.

Navn på public metoder og variabler kan virke intuitivt for oss som sitter med detaljkunnskap om hvordan rammeverket er bygd opp og hvordan logikken i simuleringen fungerer. Men det vil sannsynligvis vise seg at mye av det vi anser for intuitivt når det kommer til navngivning ikke er like intuitivt for brukere uten kunnskap til systemet. Vi har prøvd så godt vi kan å la metode og variabelnavn være beskrivende for hva brukere kunne forvente at metoden gjør samtidig som vi prøvde å holde navnene så korte som mulig. Vi prøver å holde oss unna akronymer så mye som mulig ved mindre akronymet er så universelt brukt at det praktisk talt er en del av dagligtale.
For eksempel akronymet "ID" som vi anser som mer intuitivt og hyppigere brukt i dagligtale en "Identifikasjon".

Boken sier dette om bruk av akronymer:

- " DO NOT use abbreviations or contractions as part of identifier names." (Cwalina; Jeremy; Brad. et al., 2020 s. 55)
- “DO NOT use any acronyms that are not widely accepted, and even if they are, only when necessary.” (Cwalina; Jeremy; Brad. et al., 2020 s. 55).

Akronymer kan fort bli forvirrende for brukere hvis de ikke vet hva akronymet er en forkortelse for og det kan være fort gjort å anta at ett akronym er lettere å forstå en det det faktisk er. Vi valgte derfor å holde oss unna dem så mye som mulig.

Boken går også nærmere inn på navngivning av parametere til metoder:

- “CONSIDER using named based on parameter meaning rather than parameter type.” (Cwalina; Jeremy; Brad. et al., 2020 s. 80).

Ettersom de fleste IDE-er vil gi informasjon om parameter typene som skal dendes inn, blir denne informasjonen overflødig i parameternavnene. Vi valgte å følge denne retningslinjen fordi ønsket å prøve å gi brukeren så mye informasjon som mulig om hva det er de sender inn til metoder. Siden informasjon om parametertypen alt blir dekket av IDE-er, anser vi det som bedre å bruke parameternavnene til å gi brukeren mer informasjon om hva parameteret faktisk skal brukes til.

Når det kommer til Enumer gir boken følgende råd:

- “Consider giving a property the same name as its type. For example, the following property correctly gets and sets an enum value named Color, so the property is named Color.” (Cwalina; Jeremy; Brad. et al., 2020 s. 76).

Ettersom typene til en enum kan være noe uklar ettersom det er typer som defineres av utviklerne. Vi anser det som viktig at enumnavnene faktisk beskriver hva det er enumen definerer.

Det burde være åpenbart ut ifra navnet "ShipSize" hva denne enumen burde brukes til og hvilke verdier den definerer. Vi velger derfor å følge denne konvensjonen ettersom enum navnet allerede burde være det mest beskrivende navnet for hva det er enumen definerer. Hvis vi kommer i en situasjon hvor vi oppretter en variabel av en enum type og vi ønsker å navngi den noe annet en hva enumen heter, burde det være ett hint til at enumen burde navngis mer beskrivende.

I våre øyne burde navnene til metoder og variabler hjelpe til å lære brukeren systemet uten at de skal kunne trenge å slå opp i dokumentasjonen for å kunne forstå det. Men det er en balansegang mellom å ha navn som beskriver systemet og navn som ikke blir så lange at de blir intuitive for brukeren å bruke. Boken sier:

- "DO favor readability over brevity." (Cwalina; Jeremy; Brad. et al., 2020 s. 52).

Det er bedre med ett langt beskrivende navn en ett kort diffust navn. Så i de tilfellene hvor vi kan unngå misforståelser fra brukeren ved å gjøre navnene lengere har vi valgt å gjøre det. Hva slags misforståelser brukeren kan ha kan være vanskelig å vite før brukertesting og til syvende og sist blir en slags gjettelek fra vår side. Vi forventer derfor

at navnene slik de står nå i API-et antageligvis har mangler vi ikke enda er klar over og at en del av det sannsynligvis burde endres etter brukertesting.

Delinlevering 2

2.5 Plan for brukertesting

Hva vi ønsket å oppnå fra brukertestene

Vi ønsket å teste hvor intuitivt navngivingen på metoder og parametere er, samt få tilbakemelding på hvor rett fram noen av de sentrale metodekallene egentlig er.
For eksempel hvilken objekttype som bruker vil bruke for å hente ut historien til et skip; Vil de bruke Ship-objektet, eller Simulation-objektet?

Planlegging av oppgaver

Vi delte oppgavene opp i flere mindre deler, som til slutt ville bygge opp den siste oppgaven.

Oppgavene som ble planlagt var følgende:

1. **Opprett følgende tre skip:**
    1. Stort skip med navn “Titanic”, ment for et enkelseilas, med 30 kontainere ombord.
    2. Lite skip med navn “Bebop”, for seilas på 6 dager tur retur, med 10 kontainere ombord.
    3. Medium skip med navn “Den sorte dame”, for seilas på 10 dager tur retur, med 20 kontainere ombord.

    <u>Målet med oppgaven:</u>

Med denne oppgaven håper vi å se hvor intuitivt det er å velge «Ship» klassen for opprettelse av skip, og hvor intuitiv parameterne til «Ship» konstruktøren er.
Vi vil at bruker skal opprette tre forskjellige skip, for å teste enkeltseilas-skip (skip 2) og gjentagende skip (skip 2) parameteret (isForASingleTrip), samt om parameteren for tur/retur lengden (roundTripInDays) er intuitiv (skip 3).

2. **Lag en havn som inneholder skipene ovenfor.**

   Målet med oppgaven:

   Denne oppgaven er ment til å vise om parameterne til «Harbor»-konstruktøren har gode navngivinger og er intuitive – om nødvendig spesifikasjoner er intuitivt for brukeren. Det er mer som må konfigureres med en havn, så gode parameternavn er viktig.

3. **Kjør en simulering på havnen og skipene gitt ovenfor fra dagens dato og 2 uker frem i tid.**

   Målet med oppgaven:

   Målet her er å se hvilken klasse, og hvordan, bruker vil opprette og kjøre en simulering. I tillegg ser vi etter om parameternavn og mengden parametere til «Simulation»-konstruktøren er forståelig og naturlig for bruker.

4. **Skriv ut informasjonen om alle kontainere til konsoll**

   Målet med oppgaven:

   Vi vil se hvor det er naturlig for bruker å gjøre metodekall for å skrive ut informasjon om alle kontainerne i simuleringen (havn og skip). Vår implementasjon har at metodekallet gjøres på Simulation objektet.
   Vi ser da også etter om navngivingen av metoden er god.

5. **Skriv ut informasjonen om alle skip til konsoll**

   Målet med oppgaven:

   Denne oppgaven er lik oppgave 4, i at vi ønsker å se hvor det er intuitivt å gjøre metodekall for å skrive ut informasjon for alle skip. Vår implementasjon har at metodekallet gjøres på Simulation objektet. Navngivingen på metoden er også viktig her.

6. **Skriv ut historien til skipet Den sorte dame til konsoll**

Målet med oppgaven:

Samme som 5 og 6, ser vi etter hvor det er intuitivt for bruker å gjøre metodekallet, i tillegg til navngivingen av metoden. Vår implementasjon har at dette spesifikke metodekallet gjøres på skip-objektet som historien skal skrives ut for (I dette tilfellet «Den sorte dame»-objektet)

7. **Gå gjennom koden og fortell oss hva du tolker hver kodelinje som.**

   Målet med oppgaven:

   Dette er en ekstra oppgave, hvor vi vil brukeren gå gjennom det vi har gjort, og forklare hva hver kodelinje tolkes som, i korte drag. Tanken bak dette er at vi vil få en rask forståelse for hva som er veldig intuitivt og hva som kanskje er vanskeligere å forklare oss.

2.6 Gjennomføring og evaluering av brukertesting

Plan for gjennomførelse

Vi diskuterte viktigheten for fasilitator å gi bruker tilbakemelding og "motiverende" ord, spesielt om noe skulle bli vanskelig for dem. Hvis bruker stopper opp, ikke vet eller det blir stillhet i tankegangen, vil det være viktig å fortelle bruker at dette er til stor hjelp, uten å være forstyrrende eller påtrengende. Vi ville også ha så avslappende omgivelser som mulig, så samtale og godt humør utover det vi faktisk holdt på med ville være viktig.

Vi planla for skjermopptak med OBS, og uten lyd for å forsikre at brukerens personvern og komfort var tatt hensyn til.

Vi hadde 4 brukere og valgte å dele disse inn i tre testgrupper – en testgruppe med to brukere og to testgrupper med én bruker i hver.

Vi ble enige om at fasilitator skulle gi bruker god tilbakemelding hver gang, både når det bruker gjorde funket og ikke funket. Når noe ikke funket, eller ikke var det koden var lagt opp til å gjøre, skulle fasilitator poengtere til bruker hvordan dette var god tilbakemelding for oss og hvordan dette hjelper oss å vite.

Resultater

Mange av oppgavene gikk brukerne greit gjennom i forhold til API-designet vårt, kun med litt stopp for å tenke høyt hva noe betød i relasjon til en havn, men dette virket mer som et problem med domenet enn navngivingen. For eksempel var det en bruker som lurte på hva en "container space" var på en havn. Dette følte vi at vi kunne sette bruker inn i, uten at det ville "avsløre" en løsning.

Oppgave 1 – Opprett følgende tre skip ...:

Alle brukerne kom seg gjennom denne oppgaven uten store problemer, og det meste virket veldig intuitivt. Vi spurte hva om brukerne kunne forklare oss hva de tenker isForASingleTrip og roundTripInDays bestemmer, og alle forklarte det slik vi har tenkt.

Fasilitator spurte brukerne om hvilken container størrelse de trodde de forskjellige skipene kunne laste. Her var det usikkerheter, hvor noen trodde det hadde med skip-størrelsen, og andre trodde ikke det hadde noe med skip-størrelsen å gjøre. Sånn som det er implementert nå, lastes skipene med ca. like mange small, medium og large containere, uansett skip-størrelse.

Dette fikk oss til å se et behov for bedre informasjon og konfigurasjon for brukeren rundt hvilke container-størrelser som et skip kan laste. Vi tar derfor dette med oss inn i forbedringen av API-et.

Oppgave 2 – Lag en havn som inneholder skipene ovenfor:

De fleste brukerne kom seg fint gjennom opprettelse av havn, det var kun litt høyttenking rundt hva noen ting betød i relasjon til en havn. En bruker lurte for eksempel på hva en «container space» var. Dette følte vi at vi kunne sette bruker inn i, uten at det ville «avsløre» en løsning. Vi tok også dette med oss videre for å se om vi kan finne en bedre navngiving for dette.

De nødvendige parameterne ga også mening for brukerne, og vi fikk tilbakemelding om at navngivingene var gode.

Oppgave 3 – Kjør en simulering på havnen og skipene ovenfor fra dagens dato og 2 uker frem i tid:

Alle brukerne så det nødvendig å opprette et eget simulerings-objekt til dette. Brukerne brukte også inutitivt «Simulation»-klassen. Navngivingen av parameterne er gode, da brukerne lett fylte inn nødvendig informasjon.
Det var også intuitivt å bruke metodekall på «Simulation»-objektet, og «run()» er en god navngiving.

Oppgave 4 – Skriv ut informasjonen om alle kontainere til konsoll:
Alle brukerne brukte intuitivt «Simulation»-objektet til å gjøre metodekall. Flere av de skrev med en gang «Print» og så hva som dukket opp, og valgte intuitivt «PrintContainerHistory». Andre tittet på listen over tilgjengelige metoder, og gikk deretter intuitivt for «PrintContainerHistory».

Oppgave 5 – Skriv ut informasjonen om alle skip til konsoll:
Som oppgave 4, brukte alle brukerne «Simulation»-objektet til å gjøre metodekall. De valgte også alle intuitivt «PrintShipHistory» som metoden.

Oppgave 6 – Skriv ut historien til skipet Den sorte dame til konsoll:
Her viser oppgaven at oppsettet vi har nå for å skrive ut historien til et skip, ikke er intuitivt for bruker.
Alle brukerne gikk så og si for metodekall på «Simulation»-objektet, og tok i bruk «PrintShipHistory» metoden med skipet som parameter:
simulering.PrintShipHistory(denSorteDame);

Vår implementasjon tar utgangspunkt i at metodekallet gjøres på skipobjektet selv, og ikke via «Simulation»-objektet: denSorteDame.PrintHistory();

Dette var det altså ingen av brukerne som gjorde på første forsøk.

Oppgave 7 – Gå gjennom koden og fortell oss hva du tolker hver kodelinje som:
Alle brukerne forklarte meste parten av kodelinjene likt som vi har som intensjon.

Mange av kodelinjene hadde brukerne allerede forklart godt gjennom tidligere oppgaver, så disse ble i disse tilfellene hoppet over.

Men de siste, relatert til «Log»-objekter, forklarte brukerne nokså godt, men navngivingen var ikke like klar på hvilket tidsintervall et Log-objekt holdt på.
Flere brukere trodde ett Log-objekt lagret informasjon for hele simuleringen.
Vår gjeldende implementasjon er at ett Log objekt holder på én dag (24 timer) med informasjon. Denne tilbakemeldingen tar vi med oss videre til forbedring av API-et.

Når brukerne skulle forklare kodelinjen relatert til "Event"-klassen, trodde de fleste brukerne dette var relatert til Event klassen i C#. Vår Event klasse holder på historien til statusendringer i skip og containere. Også etter å ha lært om Event og startet Event-implementasjon i API-et. ser vi også selv at dette er en veldig dårlig navngiving, da den kolliderer med standardklassen Event i C#.
Dette forsterkes også fra følgende regel i boka:

- "AVOID using identifiers that conflict with keywords of widely used programming languages." (Cwalina; Jeremy; Brad. et al., 2020 s. 53)

Vi tar derfor også dette med oss videre til forbedring i API-et.

Refleksjon

I oppgave 2, hvor bruker oppretter en havn, hadde det kanskje vært en bedre løsning å ha spesifikasjonene til havnen klar. Selve spesfikasjonene for antall loading docks, osv. Er domene-relaterte tall. Slik oppgaven var nå, måtte bruker også sette seg litt inn i havn-domenet, som skapte tenkinger og undringer rundt dette. Hadde vi hatt disse spesifikasjonene klare, kunne man unngått at bruker også måtte være kreativ og finne på for eksempel et «bra» antall loading docks osv, som i bunn og grunn ikke viktig for selve API-ets intuitivitet. Dette kunne også gitt bruker mer komfort.

Vi forklarte uansett til bruker hva de domene-spesifikke detaljene var og hva de gikk ut på, gjennom hele brukertesten.

Konklusjon

Navngivingene var forståelige og intuitive for brukerne.

Når det kom til metodekallet for å printe historie fra simuleringen (printShipHistory og printHistory), så var det ikke intuitivt for bruker å kalle «printHistory» på skip-objektet selv, som vi hadde implementert, men heller via Simulation objektet.

Dette ga oss en god tilbakemelding om at det som er desidert mest intuitivt, er å kalle slike metoder på simulering-objektet selv, uansett om det gjelder et spesifikt objekt eller generelt.

På en annen side kan det også argumenteres for at denne intuitiviteten kommer fra de tidligere oppgaven, hvor lignende metodekall ble gjort på «Simulation»-objektet. Hadde metoden for å skrive ut historien til ett enkelt skip vært før de andre to, er det mulig at det ikke hadde vært like intuitivt med et kall via «Simulation»-objektet.

Vi tar likevel med oss denne tilbakemeldingen og planlegger å forbedre printShipHistory metoden, ved å implementere en overload slik at samme metode kan enten ta ingen parametere for å printe alle skip, eller ta et Skip-objekt for å printe dens historie.

3 Evaluering og forbedring av tildelt gruppes API

Vi hadde et positivt inntrykk til den andre gruppens API, med noen forandringer i navngiving og metode-tilhørighet.

Generelle forandringer:

- Vi forandret alle ID-er til å bruke Guid i stedet for int.
- Vi forandret typene List til IList, for å følge retningslinjen:
    - "DO NOT use ArrayList or List<T> in public APIs (...) Instead of using these concrete types, consider Collection<T>, IEnumerable<T>, IList<T>, IReadOnlyList<T>, or other collection abstractions"
      (Cwalina; Jeremy; Brad. et al., 2020, s. 295).

Tar i bruk interface og ikke kun rene klasser

Alle klassene vi fikk fra dokumentasjonen, viser rene klasser ("class") og ikke noe bruk av interface. Vi har derfor implementert interfacer i tillegg.

I forhold til boka, så er det en liten gråsone om vi skal ha interface eller ikke, men i oppgaven så står det at vi skal ha interface, så derfor har vi implementert interface for alle klassene.

Class Airport

Vi har lagt til en liste PlanesInAirport med alle fly som befinner seg på flyplassen, for å ha en oversikt over alle flyene. Tanken er da at man kan hente et fly-objekt fra denne listen, og få flyets plassering osv. Siden denne ikke er ReadOnlyCollection, så har bruker mulighet til å legge til (.Add()) Plane objekter direkte i listen, men slik API-et er designet nå, så er det ikke et maks antall fly som kan være på flyplassen, og dermed ikke kodebrytende om bruker velger å legge til manuelt.

I tillegg har vi flyttet noen metoder fra andre klasser og inn i Airport, siden vi anser Airport som den overseende klasse, og eier av de fleste andre instanser. Vi flyttet for eksempel ReserveRunway og ReleaseRunway til Airport fra Runway klassen. Dette på

bakgrunn og lærdom fra våre egne brukertester, hvor brukere mer intuitivt gikk for kall via “eieren” enn på objektet selv. For eksempel så viste brukertestene at det var mer intuitivt å kalle

Airport.ReserveRunway(Runway1) enn å kalle Runway1.ReserveRunway().

```
namespace Gruppe8.TekniskTakeoverAirportAPI
{
    7 references
    public class Airport : IAirport
    {
        2 references
        public int AirportCode { get; }

        2 references
        public string Name { get; }

        2 references
        public IList<Gate> Gates { get; }

        2 references
        public IList<Runway> Runways { get; }

        // Adding a list with all planes in the airport at a given time.
        2 references
        public IList<Plane> planesInAirport { get; private set; }


        0 references
        public Airport(int airportCode, string name)
        {
            AirportCode = airportCode;
            Name = name;
            Gates = new List<Gate>();
            Runways = new List<Runway>();
            planesInAirport = new List<Plane>();
        }

        1 reference
        public void AddGate(Gate gate) { }

        1 reference
        public void AddRunway(Runway runway) { }

        1 reference
        public Runway GetAvailableRunwayForDeparture() { throw new NotImplementedException(); }

        // Added method to get an available runway to use for arriving plane.
        1 reference
        public Runway GetAvailableRunwayForArrival() { throw new NotImplementedException(); }

        // Guid for IDs and not int
        1 reference
        public int GetQueueTime(Guid gateId) { throw new NotImplementedException(); }

        1 reference
        public int GetTakeoffTime(Plane plane) { throw new NotImplementedException(); }

        // New method: Remove gate
        1 reference
        public Gate RemoveGate(Gate gate) { throw new NotImplementedException(); }

        1 reference
        public Runway RemoveRunway(Runway runway) { throw new NotImplementedException(); }

        // Moved ReserveRunway and ReleaseRunway to Airport from Runway class.
        1 reference
        public void ReserveRunway() { throw new NotImplementedException(); }

        1 reference
        public void ReleaseRunway() { throw new NotImplementedException(); }

        // Moved AssignGate in Fligh(tData) to Airport and renamed with more descriptive naming.
        1 reference
        public void AssignArrivingPlaneToGate(Gate gate, Flight flightData) { throw new NotImplementedException(); }

    };
}
```

Enum GateGroup

Vi slet litt med å forstå hva A, B og C representerte, da det ikke var noen XML-kommentar som forklarte dette, og ikke noe i dokumentasjonen heller. Vi ble fortalt at disse representerte “International Flights”, “Domestic Flights” og “Connecting Flights”. Vi var enige om at XML kommentarer hadde ordnet opp i usikkerheten, så vi gjorde ikke noen store forandringer der utenom å legge til XML kommentarer. Det var problemer

ved oppgaven som gjorde at XML kommentarene ikke kom med med .DLL filene, men vi ble fortalt at det uansett ikke eksistrerte noen XML-kommentar for dette. Et alternativ hadde vært å bruke International, Domestic og Connecting som GateGroup navnene i stedet for A, B og C. Dette hadde da vært mer brukerrettet enn domenerettet, og ikke et kompromiss vi så nødvendig for øyeblikket, nå som XML kommentarer er til stede.

Class PlaneType --> Class Plane

Vi forandret navn på PlaneType til Plane. Vi så dette som et mer intuitivt navngiving, da klassen handler mer om all informasjon om et fly og ikke kun typen fly (f.eks passasjerfly, militærfly, cargofly osv). Et alternativ var å kalle den PlaneModel, som ville være mer rettet mot modell av fly (f.eks Boeing 737) enn hva slags type fly. Vi landet på Plane fordi PlaneModel kunne havne i samme misforståelse som PlaneType, og med tanke på kompromiss mellom domene og brukervennlighet, var Plane mer intuitivt og skaper mindre usikkerheter ved bruk.

Class FlightData --> Class Flight

Vi hadde usikkerheter rundt navnet FlightData og syns egentlig ikke navnet var intuitivt nok. Likevel hadde vi også vanskelig med å komme opp med noe bedre navn. Når gruppen som hadde utviklet dette forklarte at dette skulle representere en flybillett endte vi opp med å fjerne Data fra navnet, for å fjerne usikkerhet rundt hva Data betydde i sammenhenger og syns at Flight er like beskrivende alene.

Class Runway

Vi la til en kø for fly som venter på takeoff. Det vil si at fly blir tildelt runway, og står i "kø" (venter på sin tur) ved gate. Slik vet vi hvilke fly som skal på hvilken runway. Noen fly

trenger kanskje lengre runway for takeoff, så selv om runway 1 er ledig så er denne kanskje for kort og flyet trenger minst lengden til runway 2 for takeoff.

I tillegg flyttet vi ReserveRunway og ReleaseRunway til Airport, da vi ser det som mer intuitivt for bruker å kalle dette på den "overseende" klassen og ikke på objektet selv. Dette er også basert gjennom vår egen brukertesting, hvor brukerne mer intuitivt gikk for kall på "eieren" av objektet, og ikke objektet selv. Airport "eier" runway.

Enum PlaneSize

Vi opprettet Enum PlaneSize, da en flymodell (f.eks "Boeing 737") ikke sier oss nok om hvor stort dette flyet er. Dette for tilfeller hvor for eksempel en gate eller runway ikke er stor/lang nok for et fly med størrelse "Large". Vi vurderte også å lage Enum for de forskjellige flymodellene, men valgte heller å lande på størrelsen på flyet.
Dette var også noe dokumentasjonen til gruppen allerede hintet til, men ikke hadde implementert enda.

Class Simulation

Flyttet simulering-relaterte metoder inn i Simulation klasse. Dette var påpekt som planlagt allerede i dokumentasjonen, så ikke nødvendigvis en stor endring fra vår side, kanskje mer en bekreftende "endring". Vi ser også det som designmessig riktig å ha en Simulation klasse med all logikk for selve simuleringen, og ikke et kall i hver klasse.

2.8 Forbedring av deres eget API

Forbedringer med bakgrunn i brukertesting

Gjennom brukertestingen var det noen utfordringer med hvordan vårt API er satt opp som gjorde det vanskelig for brukerne å intuitivt skjønne hvilke metodekall de skulle gjøre for å hente ut informasjon fra APIet. En gjentagende feil brukerne gjorde når de skulle printe ut historien for ett enkelt skip i en simulering var å prøve å gjøre dette

gjennom simuleringsklassen istedenfor gjennom skip objektet som representerte skipet som historien skulle hentes fra. Det ble tydelig at det sannsynligvis er mer intuitivt for brukere å kunne hente ut all informasjon som genereres gjennom en simulering gjennom simuleringsklassen istedenfor å hente det ut fra de underliggende objektene som benyttes i simuleringen.
Vi bestemte oss derfor for å lage en overload for PrintShipHistory:

Public void PrintShipHistory(Ship shipToBePrinted)

I Simulation klassen som lar deg sende inn ett skip objekt som metoden deretter skriver til konsoll historien til skipet man sender inn. Alle brukerne vi testet forsøkte først å kalle på denne metoden ved å sende inn skip til den for å printe skipets historie. Det er dermed ett godt tegn på at dette er en intuitiv måte å bruke denne metoden på.

I brukertestingen oppdaget vi også at brukerne hadde problemer med å skjønne hvor langt tidsintervall Log klassen lagret informasjon for. Brukerne skjønte at log klassen holdt på historisk informasjon om simuleringen, men flere av dem trodde at ett log objekt holdt på informasjon om hele simuleringen. For å gjøre hvilken informasjon som log objektet holder på tydeligere for brukeren har vi omdøpt Log klassen til å nå hete DailyLog. Dette gjør det forhåpentligvis mye tydeligere for brukeren basert på navnet å skjønne at ett log objekt holder på informasjon om en dag i simuleringen. I ett annet relatert problem som kom frem var at flere brukere naturlig nok trodde at Event klassen som holder på historien til statusendringer i skip og kontainere var relatert til eventer i C#. Dette er en veldig naturlig feil å gjøre basert på den dårlige navngivningen til Event klassen. Vi har dermed endret navnet på denne klassen til nå å hete StatusLog. Det nye navnet vil også forhåpentligvis gjøre det litt klarere hva slags informasjon denne klassen faktisk holder på.

En annen utfordring vi møtte på i brukertestene var at brukere hadde forskjellig oppfattelse av hvilke kontainerstørelser skipene lastet. Enkelte av brukerne trodde kontainerstørelsene samsvarte med skipsstørelsene, mens andre ikke trodde det var

noen sammenheng mellom disse to. Slik skips klassen var satt opp langde skipet like mange kontainere av hver størelse når skipet ble opprettet. Så hvis brukeren opprettet ett skip med 15 konteinere ombord ville kontruktøren laste skipet med 5 små kontainere, 5 medium kontainere og 5 store kontainere. Ulempen med dette er for det første at denne logikken er usynelig for brukeren så det er vanskelig å intuetivt skjønne at det er sånn opprettelse av kontainere fungerte og for det andre blir det vanskelig for brukeren å vite nøyaktig hvor mange av hver kontainertype de har. Hvis brukeren ønsker ett bestemt antall av en bestemt kontainertype, er det heller ingen enkelt mulighet for brukeren å få til dette på. Vi bestemte oss derfor basert på brukertestene å gjøre endringer i kontruktøren for Ship klassen slik at brukeren nå kan bestemme hvor mange av hver kontainertype de ønsker at skipet skal lastes med. Den nye kontruktøren ser nå slik ut:

public Ship(string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip, int roundTripInDays, int numberOfSmallContainersOnBoard, int numberOfMediumContainersOnBoard, int numberOfLargeContainersOnBoard)

ToString

I tillegg har vi sett på forbedringer vi kan gjøre i forhold til å få informasjon ut av APIet. Ved at de fleste variabler i public-klassene har gettere for å hente ut informasjon fra de er det allerede en rekke måter brukere kan hente ut informasjon fra APIet og simuleringer som blir kjørt. I tillegg er de fleste listene som representerer plasseringen til skip og containere tilgjengelig for brukere og egne klasser som DailyLog (tidligere kalt Log) og StatusLog (tidligere kalt Event) som holder på historien til skip og kontainere i havna som også er tilgjengelige for brukere. Men det som ble skrevet ut til konsollen gjennom print klassene var ikke tilgjengelig for brukeren i den tidligere versjonen av APIet. Vi gikk derfor igjennom public-klassene og laget en rekke ToString metoder som gjør denne informasjonen hentbar for brukere som ønsker den. Læreboken anbefaler å overskrive ToString metoder der det er mulig: “DO override ToString whenever an interesting human-readable string can be returned. The default implementation is not very useful, and a custom implementation can almost always provide more value.”

(Cwalina; Jeremy; Brad. et al., 2020 s. 316).
Det er bedre for brukeren å ha en lett måte å lage strings ut av informasjonen som finnes i de forskjellige objektene en å få tilbake standard infoen som ToString() sender tilbake ved default. Vi lagde også noen egne metoder som også returnerer informasjon fra klassene utover det vår vanlige ToString gjør.

Disse nye metodene er:

I Simulation klassen:

Override public String ToString()

Denne metoden returnerer en String som inneholder start tidspunktet og sluttidspunktet for simuleringen og ID til havnen som brukes i simuleringen.

public String HistoryToString()

Denne metoden returnerer en string som inneholder historien til alle skip i simuleringen. Hvis simuleringen aldri har blitt kjørt returnerer den en tom string.

public String HistoryToString(String ShipsOrContainers)

Denne metoden tar inn en string verdi som kan være enten “ship”, “Ships”, “container” eller “Containers”. Hvis en av de to første alternativene sendes inn sender metoden tilbake en string som inneholder hele historien til alle skip i simuleringen. Hvis en av de to sistnevnte verdiene sendes inn sender metoden tilbake en string som inneholder hele historien til alle kontainere i simuleringen. Hvis brukeren sender inn en annen verdi en de nevnt ovenfor, eller om det ikke har blitt kjørt noen simulering sender metoden tilbake en tom string.

public String HistoryToString(Ship ship)

Denne metoden tar inn ett Ship objekt og returnerer en String som inneholder hele historien til skipet som ble sendt inn.

I harbor klassen:

override public String ToString()

Denne metoden returnerer en String som inneholder informasjon om hele havnen, hvor alle skip i havnen befinner seg og hvor mange kontainerplasser som er brukt og hvor mange som er tilgjengelig.

I Ship klassen:

override public String ToString()

Denne metoden returnerer en String som inneholder informasjon om skipet, dets status og kontainerene den har i lasterommet.

override public String HistoryToString()

Denne metoden returnerer en String som inneholder informasjon om hele skipets historie. Hvis skipet ikke har vert med i en simulering returneres en tom String.

I Container klassen:

override public String ToString()

Denne metoden returnerer en String som inneholder informasjon om kontaineren, dets status og vekt

override public String HistoryToString()

Denne metoden returnerer en String som inneholder informasjon om hele containerens historie gjennom en simulering. Hvis containeren ikke har vert med i en simulering returneres en tom string.

I DailyLog klassen:

override public String ToString()

Denne metoden returnerer en String som sier tidspunktet objektets informasjon er relevant for i tillegg til antallet skip i de forkjellige lokasjonene.

public String HistoryToString()

Denne metoden returnerer en string verdi som inneholder informasjon om lokasjon og status til alle skip den dagen i simuleringen som DailyLog objektet omfatter.

public String HistoryToString(String ShipsOrContainers)

Denne metoden tar inn en string verdi som kan være enten “ship”, “Ships”, “container” eller “Containers. Hvis en av de to første alternativene sendes inn sender metoden tilbake en string som inneholder lokasjon, informasjon og status til alle skip i simuleringen. Hvis en av de to sistnevnte verdiene sendes inn sender metoden tilbake en string som inneholder informasjon om alle kontainere i simuleringen på den dagen som DailyLog objektet omfatter. Hvis brukeren sender inn en annen verdi en de nevnt ovenfor, eller om det ikke har blitt kjørt noen simulering sender metoden tilbake en tom string.

I StatusLog klassen:

override public String ToString()

Denne metoden returnerer en String som inneholder all informasjonen lagret i StatusLog objetet. Altså ID, lokasjon og Status til kontaineren eller skipet StatusLogen handler om i tillegg til tidspunktet informasjonen er relevant for.

Boken sier “DO try to keep the string returned from ToString short. The debugger uses ToString to get a textual representation of an object to be shown to the developer. If the string is longer than the debugger can display (typically less than one screen length), the debugging experience is hindered.” (Cwalina; Jeremy; Brad. et al., 2020 s. 317). Vi har prøvd å holde de standard ToString metodene så korte som mulig, mens de fremdeles inneholder så mye informasjon om objektet som mulig. Dette er også grunnen til at vi valgte å separere ut utskriftene relatert til historien til en simulering til egne metoder kalt HistoryToString() istedenfor bare ToString(). Dette gjør at vi minimerer tilfeller der ToString() kan risikere å bli for lagt i tilleg til at det forhåpentligvis blir enklere for brukeren å skjønne hva det er metodene faktisk returnerer i forhold til informasjon. Det at alle metodene som gir lignende informasjon også har samme navn gjør det forhåpentligvis også enklere for brukeren å intuetivt skjønne hvilken metode de burde bruke for å få den informasjonen de er ute etter. Denne separasjonen hjalp oss også å ungå tilfeller der ToString returnerte tomme stringer hvis objectet ToString metoden var knyttet til ikke hadde deltatt i en simulering. Noe læreboken også advarer mot: “DO NOT return an empty string or null from ToString. " (Cwalina; Jeremy; Brad. et al., 2020 s. 318.)

ReadOnlyCollection

Siden sist innlevering har vi også gjort oss noen egne erfaringer når det kommer til APIet. Slik APIet var lagt opp hadde brukere tilgang til å endre alle lister som ble returnert fra APIet. I de fleste tilfellene er dette helt greit. Vår filosofi er at brukeren i størst mulig grad skal kunne endre og justere på verdier og objekter i rammeverket slik at de har størst mulig kontroll over hvordan simuleringen ser ut, hva de får ut av den og hva de kan gjøre med det de får ut. Det er i midlertidig noen lister som det ikke gir like mye mening at brukeren skal kunne endre, nemlig listene som omhandler historien som genereres i simuleringen. Det vil si historien om hvordan havnen så ut i de forskjellige stadiene av simuleringen og hvilke statuser skip og kontainere hadde på forskjellige tidspunkt i simuleringen. Mye av denne informasjonen er i vår simulering lagret i lister som forteller posisjonen på de forskjellige skipene på de gitte tidspunktene. For å gjøre at brukeren ikke skal kunne ødelegge historien i simuleringen for seg selv gjorde vi disse listene om til ReadOnlyCollections. Fordelen her er at brukeren får tilgang til dataene i listene, men kan ikke gjøre noe for å endre selve listene. Listene som har blitt gjort om fra IList til ReadOnlyCollection er som følger:

Ship klassen:

```
public ReadOnlyCollection<StatusLog> History { get; }
```

Container klassen:

```
public ReadOnlyCollection<StatusLog> History { get; }
```

DailyLog klassen:

```
public ReadOnlyCollection<Ship> ShipsInAnchorage { get; }

public ReadOnlyCollection<Ship> ShipsInTransit { get; }

public ReadOnlyCollection<Ship> ShipsDockedInLoadingDocks { get; }

public ReadOnlyCollection<Ship> ShipsDockedInShipDocks { get; }public
ReadOnlyCollection<Container> ContainersInHarbour { get; }
```

Namespaces

Namespaces har også vært noe som ikke var satt opp ideelt i det tidligere utkastet av APIet. Namespacene som ble brukt var HarbNet og HarbFramework og de ble brukt litt omhverandre uten mål og mening. Dette gjør det vanskelig for brukeren å skjønne hvilke namespace de burde lete i og hva namespacene faktisk inneholder. Læreboken anbefaler å bruke så få namespaces som mulig. Bruker man for mange namespaces blir det fort uoversiktig for brukeren å vite hvor det er de skal lete etter klasser og variabler. For vår del gir det så langt ikke mening å bruke mer en ett namespace ettersom vi ikke har klassenavn som krasjer med hverandre eller andre namespaces i .net rammeverket. Vi har derfor samlet alt under ett namespace. Læreboken anbefaler: "DO prefix namespace names with a company name to prevent namespaces from different companies from having the same name. For example, the Microsoft Office automation APIs provided by Microsoft should be in the namespace Microsoft.Office." (Cwalina; Jeremy; Brad. et al., 2020 s. 63). Vårt filmanavn er Gruppe8 og navnet på vårt rammeverk er HarbNet. Vi opprettet derfor namespacet "Gruppe8.HarbNet" og sørget for at alle klassene våre tilhørte dette namespacet. Dette namespacet følger også bokens krav for pascalcasing, navngivning og at namespace navnet ikke skal krasje med navn på klasser som namespacet inneholder.

Event

Vi har nå "abstrahert" Simulation til å kun ta seg av simuleringslogikken.
Det vil si at Simulation ikke tenker på eller bryr seg *om* noe eller *hvordan* noe skal gjøres på bakgrunn av det Simulation har gjort, for eksempel formidling/presentering videre til brukeren. Vi har altså fjernet writeLines (utskrifter) fra Simulation, og byttet disse ut med Event-kall. Noe vi tenkte ville være en god idé hvis brukeren ikke har en konsoll.

Istedefor at simulering skriver ut til konsoll hver gang noe skjer, lar vi simulerings klassen bruke events til å sende informasjon ut fra APIet. Dette gjør at brukeren selv kan velge hvordan hendelsene i simuleringen blir representert og ikke lenger er tvunget til å

lese consol loggen for å få oversikt over statusen til simuleringen som gjøres.

Til å starte med brukte vi en event metode som var på denne måten:
Public delegate void shipLoadinContainerHandler(Ship ship);
public event shipLoadingContainerHandler? ShipLoadingContainer;

Vi var først fornøyd med dette oppsettet, men valgte å gjøre litt mer undersøkelser på event oppsett. Vi fant ut at dette var en gammel metode å gjøre det på og at det var mulig å slå de to sammen til en linje istedenfor. Vi endte derfor med dette oppsettet:
Public EventHandler? ShipLoadingContainer.
Denne metoden ga en mer lettlest og forståelig kode.

I program.cs filen og måten subscribe fungerte på ble også endret med dette tidligere hadde vi : simulation.ShipDockingtoLoadingDock += (ship) => {}
etter endringen byttet vi dette til: simulation.ShipDockingtoLoadingDock += (sender, e) =>

Når vi navnga eventene har vi passet på å bruke verb som beskriver hva det er eventen har gjort. "DO name events with a verb or a verb phrase. Examples include Clicked, Painting, and DroppedDown." (Cwalina; Jeremy; Brad. et al., 2020 s. 77.) Events er handlinger som skjer i simuleringen så det gir mening å gi dem navn som beskriver handlingen. Det er også viktig at brukeren forstår om eventet som ble raised har skjedd før eller etter hendelsen den dokumenterer. Vi har forsøkt etter beste evne å sørge for at dette blir tydelig i navngivningen vår ved å bruke navngivning som indikerer om hendelsen har skjedd eller kommer til å skje. "DO name event handlers (delegates used as types of events) with the “EventHandler” suffix, such as ClickedEventHandler. (Cwalina; Jeremy; Brad. et al., 2020 s. 77.). For eksempel eventene som indikerer at skip legger til og fra kai har fått navnene: shipDockingToLoading og shipDockedToLoadingDock for å indikere at den første skjer før skipet har lagt til kai og den sistnevnte skjer etter.

I tillegg har vi prøvd så godt det lar seg gjøre å returnere så mye relevant informasjon ut fra eventene som er hensiktsmessig. Alle eventene våre returnerer variabler med informasjon og bruker subklasser relatert til eventet for å definere argumentene som returneres. CONSIDER using a subclass of EventArgs as the event argument, unless you are absolutely sure the event will never need to carry any data to the event handling method, in which case you can use the EventArgs type directly. (Cwalina; Jeremy; Brad. et al., 2020 s. 178.) Dette gjør at brukeren står mere fritt til å velge hva de vil gjøre med eventene de abonerer til og at dette ikke blir låst til bare å skrive informasjonen ut til konsoll slik vi gjorde i den tidligere versjonen av APIet.

Vi har også holdt oss til språkbruk som er vanlig for .Net rammeverket når det kommer til XML dokumentasjon. " DO use the term “raise” for events rather than “fire” or “trigger.” When referring to events in documentation, use the phrase “an event was raised” instead of “an event was fired” or “an event was triggered.” (Cwalina; Jeremy; Brad. et al., 2020 s. 176.). Dette vil forhåpentligvis hjelpe til å gjøre dokumentasjonen forståelig og lettleselig.

Vi har implementert Event-kall for hver gang et skip tar for seg noe og når noe grunnleggende i simuleringer skjer, f.eks dagen er over eller simuleringen avslutter.

SimulationStarting – Simuleringen starter

I SimulationStart eventet får brukeren tilbake en String, i denne stringen får brukeren beskjed om at simulationen startet. Bruker får også Id’en til havnen som skal simuleres samt startDatoen på simuleringen.

SimulationEnded – Simuleringen har nådd endTime, og simuleringen er ferdig

I SimulationEnd eventet blir det gitt en string tilbake til bruker, I stringen får brukeren beskjed om at simuleringen er over og historien til simuleringen.

DayEnded – tiden er blitt 00:00, og dagen er over.

I DayOver eventet får brukeren tilbake en String og et DateTime objekt. I stringen får brukeren beskjed om at dagen er over og tiden dagen er over, i tilfellet til simuleringen blir det klokken 00:00.

DayLoggedToSimulationHistory – skips status forandres

I dayLoggedToSimulationHistory eventet får brukeren tilbake ett skips objekt og et log |objekt. Brukeren får da beskjed om hva som har skjedd med hvert enkelt skip de siste 24 timene så lenge noe har blitt gjort med dem.

ShipUndocking – skip undocker

I shipUndocking eventet får brukeren tilbake tidspunktet hendelsen tok sted, navnet på skipet og hvilken dock den docker til med en ID

ShipDockingToShipDock – skip docker til ship dock

I ShipDockingToShipDock får brukeren tilbake et Shipname, hvilken dockId dock'en har og tidspunktet hendelsen tar sted.

ShipDockedToShipDock – skip har docket til ship dock

I ShipDockedToShipDock får brukeren tilbake et Shipname, hvilken dockId dock'en har og tidspunktet hendelsen tar sted.

ShipDockingToLoadingDock – skip docker til loading dock

I ShipDockingToLoadingDock eventet får bruker tilbake tidspunktet hendelsen tar sted, shipname og hvilken Dock som blir brukt ved bruk av ID

ShipDockedToLoadingDock – skip har docket til loading dock

I ShipDockedToLoadingDock eventet får bruker tilbake tidspunktet hendelsen tar sted, shipname og hvilken Dock som blir brukt ved bruk av ID

ShipLoadedContainer – skip har lastet på en container

I ShipLoadedContainer eventet får bruker først tidspunktet hendelsen tar sted, shipname og til slutt får brukeren vite hvilken størrelse på kontaineren det er når den

blir lastet på skipet. Deretter får brukere navnet på skipet, et kontainer objekt og tidspunktet.

ShipUnloadedContainer – skip laster av en kontainer

I ShipUnloadedContainer eventet får bruker tilbake tiden hendelsen skjer på, shipname og beskjed om størrelsen på kontaineren som ble lastet

ShipAnchored – skip har ankret

I ShipAnchored eventet får bruker tilbake tidspunktet hendelsen tar sted, shipName og beskjed om hvilken Id anchorage har

ShipAnchoring – skip ankrer

I ShipAnchored eventet får bruker tilbake tidspunktet hendelsen tar sted, shipName og beskjed om hvilken Id anchorage som skipet skal ankres til.

ShipInTransit – skip er i transit

I shipInTransit eventet får brukeren først vite tidspunktet hendelsen skjer, deretter navnet på skipet og en beskjed om at skipet er i transit og transitId

Slik sier Simulation i fra at f.eks “nå har Skip O’hoi forlatt havnen” ved å kalle ShipUndocked?.Invoke
Det er nå opp til den mer ytre logikken (Program i vårt tilfelle) å kommunisere dette til brukeren, om ønskelig.
Program abonnerer (“ønsker beskjed når …”) på ShipUndocked, og får dermed beskjed når Simulation kaller den.
I vårt tilfelle er Program et console-program som skriver ut informasjon i terminal, og har derfor valgt å skrive “Ship {ship} has undocked” i terminalen når den får melding om ShipUndocked fra Simulation.

I tillegg velger Program å “unsubscribe”/”melde seg av” på eventer den ikke lenger trenger eller vil ha beskjed om. Som eksempel viser vi i Program at Program melder seg av eventene etter at simulation.run() er kjørt, da det nå ikke lenger vil komme flere beskjeder om hendelser fra simulation.

Slik følger Simulation også “Single Responsibility” prinsippet, ved å kun forbeholde Simulation til simuleringslogikken.

Exceptions

Exceptionsene vi har implementert går ut på å sjekke om inputen til parameterne følger kravene, objekter eksisterer i lister, sammenligning, sjekk om objekter holder på containere og om parametere har oversteget satte grenser. Skulle ikke kravene bli oppfylt blir en exception kastet som ønsket av følgende regel;

- ‘DO report execution failures by throwing exceptions.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 256.).

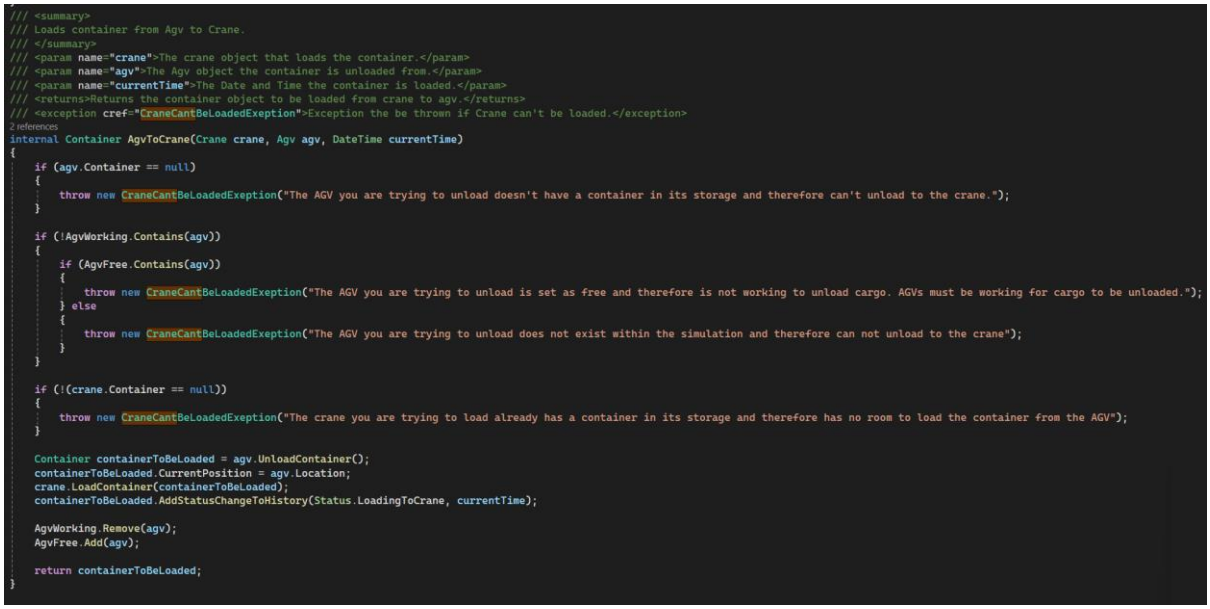
Vi har også sørget for at exceptionsene returnerer en String som en error message, som forklarer hvorfor en exception ble kastet og hvordan man kan fikse det, framfor de standard error kodene. For eksempel kan en exception bli kastet fordi skipet du prøver å hente ut fra havnen i simuleringen ikke eksisterer i listen med alle skip opprettet.

- ‘’DO NOT return error codes.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 255.).

Vi implementerte exceptionsklassene våre for å hovedsakelig sjekke om et objekt holder på en container eller ikke, utenom dette tilfelle bruker vi exceptions som allerede eksisterer i frameworket.

- ‘’DO NOT create a new exception type if the exception would not be handled differently than an existing framework exception.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 262.).

Vi har implementert egne exception klasser og brukt exceptions som allerede eksisterer i frameworket.
Under starten av implementasjoner av exceptions, så vi ikke noe behov for å opprette egne exceptions klasser før vi kom videre i utviklingen av rammeverket.
Nedenunder er et eksempel fra koden hvor en exception klasse vi opprettet blir brukt i en metode:



```
/// <summary>
/// Loads container from Agv to Crane.
/// </summary>
/// <param name="crane">The crane object that loads the container.</param>
/// <param name="agv">The Agv object the container is unloaded from.</param>
/// <param name="currentTime">The Date and Time the container is loaded.</param>
/// <returns>Returns the container object to be loaded from crane to agv.</returns>
/// <exception cref="CraneCantBeLoadedExeption">Exception the be thrown if Crane can't be loaded.</exception>
2 references
internal Container AgvToCrane(Crane crane, Agv agv, DateTime currentTime)
{
    if (agv.Container == null)
    {
        throw new CraneCantBeLoadedExeption("The AGV you are trying to unload doesn't have a container in its storage and therefore can't unload to the crane.");
    }

    if (!AgvWorking.Contains(agv))
    {
        if (AgvFree.Contains(agv))
        {
            throw new CraneCantBeLoadedExeption("The AGV you are trying to unload is set as free and therefore is not working to unload cargo. AGVs must be working for cargo to be unloaded.");
        } else
        {
            throw new CraneCantBeLoadedExeption("The AGV you are trying to unload does not exist within the simulation and therefore can not unload to the crane");
        }
    }

    if (!(crane.Container == null))
    {
        throw new CraneCantBeLoadedExeption("The crane you are trying to load already has a container in its storage and therefore has no room to load the container from the AGV");
    }

    Container containerToBeLoaded = agv.UnloadContainer();
    containerToBeLoaded.CurrentPosition = agv.Location;
    crane.LoadContainer(containerToBeLoaded);
    containerToBeLoaded.AddStatusChangeToHistory(Status.LoadingToCrane, currentTime);

    AgvWorking.Remove(agv);
    AgvFree.Add(agv);

    return containerToBeLoaded;
}
```

Exception klassene vi selv opprettet blir hovedsakelig brukt på metoder som innebærer container objekter, hvor de diverse kjøretøyene i rammeverket blir sjekket om de holder på en container. Ellers blir det sjekket om parametere brukeren skriver inn eksisterer i relaterte lister til metoden, hvis det satte parameteret ikke eksisterer blir en exception vi har implementert kastet.

Vi opprettet egne exceptionsklasser etter implementasjonen av kjøretøyene AGV, Crane og Truck kom, da det var behov for å transportere containere gjennom loading og unloading mellom de diverse kjøretøyene. Det ble nødvending å sjekke om f.eks et Crane objekt allerede holdt på en container i en loading prosess, eller om en Truck var i Transit med en container.

Utenom de nevnte tilfellene bruker vi exceptions rammeverket allerede har implementert.

- ‘’DO NOT create a new exception type if the exception would not be handled differently than an existing framework exception. Throw the existing framework exception in such a case.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 262.)
- ‘’DO create a new exception type to communicate a unique program error that cannot be communicated using an existing framework exception.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 263.)
- ‘’DO NOT create and throw new exceptions just to have your feature name in the exception type or namespace name.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 263.)

Vi har ikke implementert eller brukt exceptions som System.Exception eller System.SystemException.

- ''DO NOT throw System.Exception or System.SystemException.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 274.)

Vi har gått over exception error messages og sett over at de følger gramatikken boken ønsker vi følger. Error messagene gir kun utdypende informasjon om hvorfor exception ble kastet og hvordan det kan løses.

- ''DO ensure that exception messages are grammatically correct.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 264.)
- ''DO ensure that each sentence of the message text ends with a period.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 264.)
- ''DO NOT disclose security-sensitive information in exception messages.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 265.)
- "DO provide a rich and meaningful message text targeted at the developer when throwing an exception. The message should explain the cause of the exception and clearly describe what needs to be done to avoid the exception." (Cwalina; Jeremy; Brad. et al., 2020 S.262).

Egne exceptions klasser implementert:

Egne exceptions klasser vi har opprettet med metoder som oppretter et nytt exception objekt. Vi oppfyller minstekravet av metoder boka ønsket man implementerte i en exception klasse, og tar utgangspunkt av eksempelet vist i boka og det foreleser har gjennomgått i foreleser.

- ''Do provide (at least) these common constructors on all exceptions.'' (Cwalina; Jeremy; Brad. et al., 2020 S.262).

***AgvCantBeLoadedException*:**

AgvCantBeLoadedException()

AgvCantBeLoadedException(string message) : base(message)

AgvCantBeLoadedException(String message, Exception innerException) :
base(message, innerException)

CraneCantBeLoadedException:

CraneCantBeLoadedException()

CraneCantBeLoadedException(string message) : base(message)

CraneCantBeLoadedException(String message, Exception innerException) :
base(message, innerException)

TruckCantBeLoadedException:

TruckCantBeLoadedException()

TruckCantBeLoadedException(string message) : base(message)

TruckCantBeLoadedException(String message, Exception innerException) :
base(message, innerException)

Exceptions implementasjoner i DailyLog:

HistoryToString - exception hvis input til parameter ikke er riktig

Exceptions implementasjoner i Harbor:

Harbor konstruktør - exception hvis mengden av single-trip ships av spesifikk størrelse er større enn antall loadingDocks av samme størrelse

Initialize - exception hvis prosenten av containers lastet fra ship direkte til truck er utenfor den satte rekkevidden

CraneToTruck – exception hvis truck allerede er i transit og ikke kan loade container fra ship, eller hvis truck ikke eksisterer

SendTruckOnTransit - exception hvis truck ikke eksiterer

CraneToAgv – exception hvis AGV'en allerede er i bruk og ikke kan loade container fra crane, eller hvis AGV'en ikke eksisterer

AgvToCrane – exception hvis AGV'en ikke holder på en container, hvis den er satt som free og ikke kan ''jobbe'', hvis den ikke eksiterer eller den allerede holder på en container

ContainerRowToCrane – exception hvis Crane allerede holder på en container

GetShipFromAnchorage – exception hvis ship ikke eksisterer i shipsInAnchorage listen

GetShipFromLoadingDock - exception hvis ship ikke eksisterer i shipsInLoadingDock listen

GetShipFromShipDock - exception hvis ship ikke eksisterer i shipsInShipDock listen

GetLoadingDockContainingShip - exception hvis ship ikke eksisterer i shipsInLoadingDock listen

NumberOfFreeContainerSpaces - exception hvis containere av spesifisert størrelse som er free (ledige) ikke eksistrerer i freeContainerSpaces listen

GetContainerRowWithFreeSpace - exception hvis containere av spesifisert størrelse som er free (ledige) ikke eksistrerer i freeContainerSpaces listen

GetContainerStatus - exception hvis container ikke eksisterer i storedContainers listen

Exceptions implementasjoner i Ship:

SetBaseShipInformation – exception hvis shipSize ikke har riktig input

CheckForValidWeight – exception hvis skipets currentWeightInTonn overstiger maxWeightInTonn, shipSize ikke er angitt riktig og containersOnBoard overstiger containerCapacity.

GetContainer – exception hvis container av spesifisert størrelse ikke eksisterer i containersOnBoard listen

Exceptions implementasjoner i Simulation:

PrintShipHistory – exception hvis det spesifiserte skipet ikke eksisterer i AllShips listen i Harbor

HistoryToString - exception hvis det spesifiserte skipet ikke eksisterer i AllShips listen i Harbor

3 Iterasjon 2

Delinlevering 3

3.1 Kravsliste

Med den nye oppgaven ble det nødvendig å oppdatere APIet til å tilpasse seg spesifikasjonene gitt i oppgaveteksten. Vi har prøvd fra starten av å holde APIet vårt så generelt som mulig slik at det er hardkodet så få spesifikke detaljer om havnen som mulig. Vi ønsker isteden at så langt det lar seg gjøre at det er brukeren selv som sender inn detaljene om havnen sin til rammeverket og rammeverket heller bare tar seg av simuleringen av havnen ut ifra de detaljene brukeren sender inn. Havnen i oppgaveteksten kunne derfor i stor grad allerede simuleres ved hjelp av HarbNet rammeverket, men det var noen nye elementer i simuleringen som vi fremdeles måtte ta hensyn til.

De nye elementene lagt til for å tilpasse de nye kravene om transport av containere fra skip og opprettelse av lagrinskolonner var:

Introduksjon av Lastebiler

I den tidligere versjonen av rammeverket var det ikke tatt hensyn til at lastebiler kunne komme og hente kontainere fra havnen. Det var heller antatt at alle kontainere skulle fraktes med skip.

Introduksjon av Kraner

I den tidligere versjonen av rammeverket tok vi ikke hensyn til kranene som lastet kontainere av og på skip. Slik det var implementert var det skipene selv som tok seg av lasting og avlasting av kontainere. I den nye implementasjonen skulle denne jobben heller bli gjort av kraner.

Introduksjon av AGVer

Hvordan kontainere ble flyttet rundt på kontainerområdet ble ikke tatt hensyn til i det gamle rammeverket. Denne jobben er nå blitt gitt til kjøretøyene representert av en ny AGV klasse.

Kontainerstørrelser av type Half og Full størrelse.

Slik vi hadde satt opp kontainerstørrelser så kunne kontainere være av Small, Medium og Large størrelse. Dette skiftes nå ut til å bestå av Half og Full størrelser for å være bedre tilpasset til lagrinskolonnene som blir implementert. I tillegg har det blitt tatt hensyn til at en lagringsplass for en kontainer skal kunne holde 2 Half størrelse kontainere eller 1 Full størrelse kontainerne.

Ytterligere detaljer i lagringsområdet av kontainere på havna.

For å bedre representere havnas lagringsplass for kontainere i større detalj ville det bli nødvendig å gjøre endringer på hvordan denne plassen representeres i koden. Slik vi hadde det frem til nå var det en liste med ContainerSpace objekter hvor hver ContainerSpace representerte ett areal hvor en kontainer kunne lagres på havnen. Det ble ikke feil å ha det sånn i forhold til oppgaveteksten ettersom den totale flaten kontainere kunne lagres på til slutt bare blir ett mattestykke på hvor mange slike flater med ledig plass det er på havnen. Men vi ønsket å legge til mer detalj på hvordan denne plassen representeres. Og hvilke av disse ContainerSpace flatene som henger sammen i større områder.

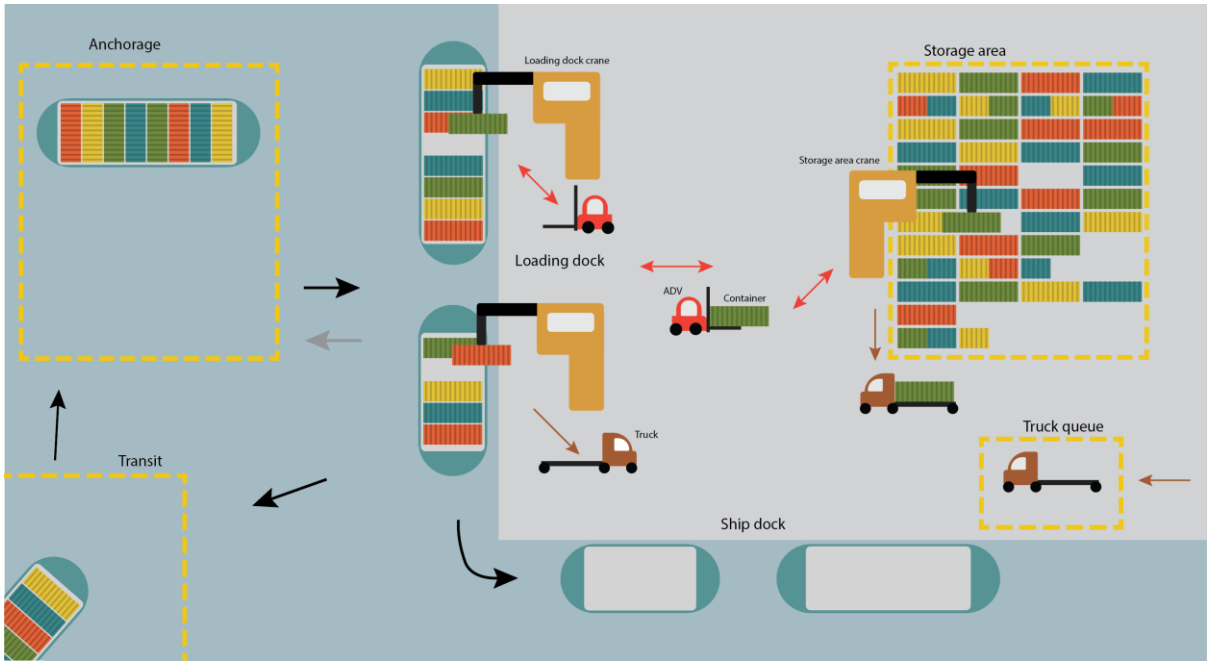
Andre implementasjoner

I tillegg til å introdusere disse nye elementene i rammeverket ønsket vi også å benytte muligheten til å forbedre rammeverket på noen områder som kanskje ikke var implementert helt optimalt fra de tidligere innleveringene.

For eksempel var det ett krav om at skip som bare gjorde et seilas, såkalte singleTripShips skulle starte i loading docks når simuleringen startet. Dette gjorde brukeren var tvunget til å ha like mange loading docks som det var singleTripShips i simuleringen. Denne begrensingen ønsket vi å komme oss bort fra.

Det var også en begrensning som sa at det måtte være like mange kaiplasser for skip som det var singleTripShips ettersom singleTripShips la seg til kai og ble liggende der ut simuleringen etter at de var ferdig med seilasen sin. Denne begrensningens hadde vi også lyst til å komme oss bort fra.

En annen begrensing i havnen var at kontainere i simuleringen aldri ble lastet av noe annet sted en havn. Dette gjorde at de essensielt bare gikk på rundgang fra lasteplassen på havnen, til å bli sendt på havet i ett skip til å bli lastet på havnen igjen når skipet kom tilbake. Vi ønsket at skipene og lastebilene som forlatte havnen skulle laste av kontainerne sine ved destinasjonen sin for deretter å få helt nye kontainere som de kom tilbake med. Noe som antageligvis er mer riktig en at kontainerne bare gikk på rundgang.



Visualisering av hvordan havnen fungerer etter de nye implementasjonene.

Nye overloads på gamle metoder

For å gjøre det litt enklere for brukere å hente ut informasjon fra simuleringen bestemte vi oss også for at vi ville lage noen flere overloads på metodene PrintShipHistory og HistoryToString slik at de også kunne ta inn Guid til skipet som det skulle hentes historien til.

Nye Objekter:

Kran objekter mellom LoadingDocks og AGV/Lastebiler, og mellom AGV og container spaces.

Lastebil objekter mellom Kraner og Destination liste.

AGV objecter mellom Kraner og ContainerSpaces (Frakte containere fra kran til kran).

Container:

Daycounter (daytime) objekt som teller antall dager containeren har vært i storage.

Nye metoder/klasser:

Truck

I oppgaven står det at en Truck skal laste og laste av kontainere hos en kran, derfor valgte vi å lage en egen klasse for truck. I klassen fant vi ut av at det måtte inneholde en unik ID, den nåværende statusen til trucken og den måtte kunne ha et Container objekt som var knyttet til den. Vi må vite hvor truck objektet befant seg, så vi har også med location variabel. Samt måtte Trucken kunne loade og unloade kontainere.
Med disse kravene planla vi at Truck måtte minst ha disse propertisene og metodene:

- **ID** – Guid objekt
- **Location** – Guid til et objekt eller lokasjon.
- **Status** – Status objekt, Enum klasse.
- **Container** – Holder på container objektet som er plassert på trucken.
- **LoadContainer()** – Fjerner Container objektet fra Trucken.

- **UnloadContainer()** – Legger til Container objekt til Trucken.

AGV

I oppgaven står det at AGV'er skal transportere kontainere fra storage til kran og omvendt, derfor valgte vi å lage en egen klasse for AGV. Klassen fant vi ut av at måtte inneholde en unik ID, den nåværende statusen til AGV objektet, vi måtte også vite hvor AGV objektet befant seg, så vi har også med location variabel. Samt måtte AGV'en kunne loade og unloade kontainere.

Vi kom med dette fram til at AGV klassen måtte minimum ha disse propertisene og metodene:

- **ID** – Guid objekt
- **Location** – Guid til et objekt eller lokasjon.
- **Status** – Status objekt, Enum klasse.
- **Container** – Holder på container objektet som er plassert på AGVen.
- **UnloadContainer()** – Fjerner Container objektet fra AGVen.
- **LoadContainer()** – Legger til Container objekt til AGVen.

Kran

I oppgaven skulle vi også ha med kraner i harbor. Kran valgte vi å lage en klasse ut av, den skulle inneholde en unik ID, vi var også nødt til å ha location ettersom at vi måtte ha kraner ved containerspace og loading dock, mengden kontainere et kran objekt kan loade hver time, container object som var knyttet til kranen, samt måtte de kunne unloade og loade kontainere.

Kran klassen må med dette ha minimum disse propertisene og metodene:

- **ID** – Guid objekt
- **Location** – Guid til et objekt eller lokasjon.
- ContainersLoadedPerHour – antall containere kranen kan loade per time.
- **Container** – Holder på container objektet som er plassert på AGVen.
- **UnloadContainer()** – Fjerner Container objektet fra AGVen.
- **LoadContainer()** – Legger til Container objekt til AGVen.

Harbor

I harbor var vi nødt til å legge til en god del metoder, ettersom at havnen er den som styrer det meste som skjer i API-et. Vi fant ut av at harbor måtte ha mulighet til å generere trucker selv, alt etter behov, den måtte også ha mulighet til å se hvilke AGV-er, Trucker og StorageCrane som var ledige. Harbor skulle også ta seg av Objekt to Objekt metoder som shipToCrane, CraneToShip og CraneToTruck etc. Metodene håndterer flytting av kontainere mellom objektene.

Harbor klassen ble dermed planlagt til å få disse nye metodene i startfasen:

- **GenerateTruck()** – Lager ett nytt Truck objekt
- **GetFreeAGV()** – Returnerer en ledig AGV.
- **GetFreeTruck()** – Returnerer en ledig Truck.
- **GetFreeStorageAreaCrane()** – Returnerer en ledig kran ved havnens storage area.
- **ShipToCrane()** – Tar en kontainer fra ett ship, fjerner den fra egen storage og legger den til kranens storage.
- **CraneToShip()** – Tar en kontainer fra en crane, fjerner den fra kranens storage og legger det til skipets storage.
- **CraneToTruck()** – Tar en kontainer fra en kran, fjerner den fra kranens storage og legger den til en trucks storage.
- **CraneToAGV()** – Tar en kontainer fra en kran, fjerner den fra kranens storage og legger den til AGV'ens storage.
- **AGVToCrane()** – Tar en kontainer fra en AGV, fjerner den fra AGV'ens storage og legger den til kranens storage.
- **ContainerRowToCrane()** – Tar en kontainer fra en ContainerRow, fjerner den fra ContainerRowen og legger den til kranens storage
- **CraneToContainerRow()** – Tar en kontainer fra en ContainerRow, fjerner den fra ContainerRowen og legger den til kranens storage

I Simulation skjønte vi at det ville kreve forandringer i logikken som tok for seg flyttingen av containere. Spesielt ville det kreve forandringer i de overordnede metodene UnloadingShips() og LoadingShips() som tok i bruk de mindre container-flytt metodene i Harbor.

3.2 Oppdatert design av API

ContainerSize

Oppgaven spesifiserte at kontainere kan komme i størrelsene Half og Full. Vi endret derfor den public enumen ContainerSize for å representere disse nye størrelsene.

De nye verdiene i enumen er nå:

- None = 0
- Half = 10
- Full = 20

Der tallverdien representer vekten på kontaineren i tonn. Container objekter bruker nå disse nye verdiene i tillegg til den nye public ContainerStorageRow klassen.
Brukere kan dermed bruke denne enumen til å sjekke og gi størrelse på kontainere som brukes i simuleringen.

ContainerStorageRows

I oppgaven spesifiseres det at havnen som kunden er ute etter skal containere lagres på en tredimensjonal flate som har rad etter rad etter rad med kontainerplasser for lagring av kontainere.

Med slik vårt API håndterer posisjoner containere kan befinne seg, nemlig i form av lister, ble det da nødvendig å vurdere bruken av en todimensjonal liste for å lagre posisjonen til hver enkelt lagringsplass for en kontainer. Todimensjonale lister kan fort bli forvirrende både for utviklere og brukere av systemet så istedenfor valgte vi å opprette en ny klasse ContainerStorageRow som definerer en rad med kontainer lagringsplasser.

Listen for hele lagringsområde i havnen kunne dermed forbli endimensjonal og holde på flere slike ContainerStorageRow objekter for at lagerplassen skulle representere et helt areal for lagring av kontainere. For å minimere antall attributter i konstruktøren til harbor og for å gi brukeren mest mulig kontroll over hvor mange lagringsplasser havnen har valgte vi å gjøre denne nye klassen public. Brukere vil dermed selv opprette rader og

angi antall lagringsplasser for hver rad for så å sende en liste med dette inn til harbor som brukes i simuleringen. Siden ContainerStorageRow nå er en del av det offentlige API-et definerte vi også ett interface IContainerStorageRow for å spesifisere de public tilgjengelige variablene og metodene i klassen. Oppgaven sier også at hvis en kontainer av halv størrelse blir lagret i raden så må resten av kontainerne i raden være av hav størrelse. Vi gjorde derfor implementasjon for å forsikre oss om at alle kontainere i raden alltid er av samme størrelse.

De public variablene og metodene er som følger:

- **public Guid ID { get; }**
  Returnerer en unik ID i form av en Guid for storageRowen.
- **public int numberOfFreeContainerSpaces(ContainerSize size)**
  Returnerer ett tall i form av int på antallet ledige containerplasser i raden.
- **public ContainerSize SizeOfContainersStored();**
  Returnerer en ContainerSize enum som forteller hvilke størrelse det er på kontainerne som er lagret i raden.
- **public IList<Guid> GetIDOfAllStoredContainers();**
  Returnerer unike IDer i form av Guid på alle kontainere som er lagret i containerStorageRow
- **public String ToString();**
  Siden den standard ToString metoden ikke returnerer veldig brukbar data har vi i likhet med alle andre public klasser valgt å gi en override på tostring. Læreboken anbefaler også dette når det er mulig:
  - "DO override ToString whenever an interesting human-readable string can be returned." (Cwalina; Jeremy; Brad. et al., 2020 S.316).

Harbor konstruktør

Det er en del nye elementer å ta hensyn til i den nye versjonen av rammeverket. Simuleringen skal nå ta hensyn til AGVer, Trucks og Cranes. For å holde rammeverket enkelt å bruke ønsket vi at brukeren skulle trenge å forholde seg til så lite som mulig av logikken til hvordan disse elementene bruktes i simuleringen. Vi valgte derfor å legge

opprettelsen av enkelte av disse objekttypene i Harbor konstruktøren istedenfor å gi brukeren direkte tilgang til disse klassene. I enkelte tilfeller som ContainerRow ble det for lite intuitivt fra brukersiden å gjøre dette som argumenter i konstruktøren, men for Crane, Trucks og AGV gikk dette fint.
Vi introduserte derfor parameterne:

- **Int numberOfCranesNextToLoadingDocks**
  Antall kraner vedsiden av laste kaiene som kan laste kontainere av og på skip.
- **Int numberOfCranesOnHarborStorageArea**
  Antall kraner vedsiden av lager området til havnen der det lagres kontainere.
- **Int LoadsPerCranePerHour**
  Antall omlast en kran kan gjøre i timen i simuleringen.
  Denne variabelen valgte vi for at rammeverket fremdeles skulle være så generelt som mulig og at disse verdiene ikke skulle trenges å hard kodes inn. Vi synes det er bedre at brukeren selv kan angi effektiviteten av kranene og at simuleringen forholder seg til tallet brukeren gir.
- **Int NumberOfAgv**
  Antall AGVer havnen har til rådighet til å frakte kontainere mellom laste kaiene og havnens lagringsplass.
- **Int LoadsPerAgvPerHour**
  Hvor mange omlastninger en AGV klarer per time. Dette er igjen for å holde rammeverket så generelt som mulig.
- **Int numberOfTrucksArriveToHarborPerHour**
  Antallet lastebiler som ankommer havnen i timen. Dette tallet vil ha effekt på effektiviteten til kaien så det er fint å la brukeren selv angi antallet.

I tillegg la vi til to attributter hvor brukeren kunne angi prosenten av kontainere som lastes til lastebiler fra skip og kaias lasteplass:

- **Int percentageOfContainersDirectlyLoadedFromShipToTrucks**
  Antallet i prosent av kontainere som skal lastes direkte fra skip og til lastebiler.
  Her kaster vi også en ArgumentOutOfRangeexeption hvis brukeren prøver å gi en

prosentverdi som er under 0 eller over 100. Disse verdiene gir ikke mening og det er fint om koden tar hensyn til dette.

- **Int percentageOfContainersDirectlyLoadedFromHarborStorageToTrucks**
  Antallet i prosent av hvor mange kontainere som skal lastes rett fra havnas lagringsplass til lastebiler. Denne kaster også en ArgumentOutOfRangeExeption hvis tallet er under 0 eller over 100.

Til slutt la vi til ett argument hvor brukeren sender inn en liste av ContainerStorageRow:

- **IList<ContainerStorageRow> listOfContainerStorageRows**
  Denne listen inneholder ContainerStorageRow objekter som havnen skal bruke for å representere sin egen lagringsplass. Grunnen til at vi valgte å la brukeren opprette disse objektene selv og sende det inn som en liste er for å gi brukeren mer kontroll over hvor stor lagringsplass hver rad med lagringsplasser skulle ha. En kan tenke seg at forskjellige rader har forskjellige antall lagringsplasser så det er mer fleksibelt å la brukeren opprette lagringsradene selv og selv lage listen som skal brukes i simuleringen.

Overloads av PrintShipHistory og HistoryToString metodene i Simulation klassen.

Vi implementerte overloads av disse metodene slik at de kunne ta inn Guid til skipet som man ønsket historien til. Dette gjorde det nødvendig å i tilfellene hvor skipet ikke fantes i havnen som simuleringen benyttet å kaste en exception for å gi informasjon om dette til brukeren.

Vi vurderte å lage en ny ShipNotFound exception som disse metodene kunne kaste, men igjen gikk vi bort fra ideen om å lage en unik exception klasse for dette.
Lære boken sier:

- “DO NOT create a new exception type if the exception would not be handled differently than an existing framework exception. Throw the existing framework exception in such case.” (Cwalina; Jeremy; Brad. et al., 2020 S.262).

Det er antageligvis mer intuitivt for en bruker å forholde seg til allerede kjente exceptions istedefor å måtte forholde seg til exception typer som er unike for dette rammeverket. Boken sier

- “DO create a new exception type to communicate a unique program error that cannot be communicated using an existing framework exception” (Cwalina; Jeremy; Brad. et al., 2020 S.262),

men ettersom exceptionen i dette tilfelle vil bli kastet hvis brukeren sender inn ett argument som ikke gir mening i forhold til det metoden skal gjøre er dette allerede dekket av .NET rammeverkets egen ArgumentException. Det gir derfor mer mening å kaste denne exceptionen enn å kaste en egen ny exception type.

Vi prøver alltid så godt det lar seg gjøre å gi beskrivende tekst i exceptionene vi kaster. Det er viktig at exception teksten ikke bare beskriver hva som er feil, men også gjør det tydelig hvordan dette kan rettes opp i.

- “DO provide a rich and meaningful message text targeted at the developer when throwing an exception. The message should explain the cause of the exception and clearly describe what needs to be done to avoid the exception.” (Cwalina; Jeremy; Brad. et al., 2020 S.262).

Events forandringer og nye events

I Eventene vi hadde så vi en gjengang av visse parametere/argumenter, og så da muligheten for å opprette en Base som inneholdt disse argumentene. BaseShipEventArgs heter denne, og inneholder Ship, CurrentTime og Description argumentene. Denne arver fra EventArgs.
EventArgs hvor et skip er involvert, for eksempel ShipUndockingEventArgs eller ShipInTransitEventArgs, arver så fra BaseShipEventArgs.

Vi har også passet på at EventArgs-ene inneholder konstuktør, slik at disse tas i bruk ved opprettelse.

I tillegg har vi laget noen flere EventArgs, både for skip-aktivitet men også en ny en for trucker som lastes ved lagringsplass: TruckLoadingFromStorageEventArgs.

3.3 Dokumentasjon av implementasjon

I den nye implementasjonen av API-et var vi nødt til å endre hvordan en kontainer ble lastet til og fra skip. Tidligere tok harbor seg av dette automatisk hvor den hentet fra et skip og la det direkte til en lagringsplass. Nå skulle vi implementere kraner som skulle laste fra og til skip.

Vi valgte derfor å legge en kran "imellom" Ship og lagringsplass.
I kran konstruktøren har brukeren mulighet til å velge antall kontainere en kran kan laste i timen. Vi tenkte at dette var en god løsning slik at brukeren kan styre hastigheten på kranene selv. Kranen skal da laste kontainere til enten en AGV eller en Truck.
Trucken har kun plass til en container, og når den har fått denne containeren kjører den ut av havnen, hvis det allerede er trucker som venter på containere har de mulighet til å stille seg i kø.
AGV klassen har derimot en større jobb en hva trucken har, i oppgaven fikk vi vite at det skulle være 20 av de som var på havnen til enhver tid. Den ble det neste objektet som skulle være imellom en kran og lagringsplass. AGV-en laster enten til lagringsplassen eller til en skipskran.

Vi var også nødt til å endre hvordan containere ble lagret. Tidligere ble de kun lagret, men nå fikk vi spesifikasjoner på hvordan de skulle lagres.

Member design

I designet av vårt api har vi gitt mulighet til bruker å overloade konstruktører og noen utvalgte metoder, dette gjorde vi fordi at vi fant ut igjennom brukertestingen vår at forskjellige brukere har forskjellige tanker rundt metoder og konstruktører og hva de skal inneholde. For å reflektere dette la vi til ett par overloads på metoder og konstruktører vi selv synes det var naturlig å legge det.
For eksempel i Ship-klassen:

- Ship(string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip, int roundTripInDays, int numberOfHalfContainersOnBoard, int numberOfFullContainersOnBoard)
- Ship(string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip, int roundTripInDays,IList<Container> containersToBeStoredInCargo, int directDeliveryPercentage)

For overload metodene våre har vi satt de samme navnene til enhver parameter som er felles. Dette har vi gjort fordi vi ønsker at det skal være lett å forstå at de parameterne holder på de samme verdiene.
I boken sies dette om parameter navn i overloads

- “Do try to use descriptive parameter names to indicate the default used by shorter overloads.” (Cwalina; Jeremy; Brad. et al., 2020 s. 137).

I API’et vår har også mange properties.
I Ship klassen har vi for eksempel ID, ShipSize, Name, og StartDate. Vi diskuterte tidlig i oppgaven hvordan de skulle håndteres med tanke på get og set. Flere av de var det naturlig at en bruker skulle kunne endre selv og hente når det var ønskelig. Det var også noen av de som det ikke var gunstig at kunne endres.
Hvis vi ser på ID property som et eksempel: ID property er av typen GUID, det vil si at det er en tilfeldig string med tall og bokstaver, noe som er fint for å få unike identifikatorer. Problemet oppstår hvis en bruker har mulighet til å endre dette til en vilkårlig streng, da vil muligheten for at den er unik bli lavere. Vi valgte derfor å fjerne set metoden fra denne property’en.
I boken blir dette sagt om get og set til properties

- “Do create get-only properties if the caller should not be able to change the value og the property. If the type off the property is mutable refrence type, the property value can be changed even if the property is get-only"
  (Cwalina; Jeremy; Brad. et al., 2020 s. 158).

For å følge det nye kravet hvor et visst antall prosent av containerne fra skip som kommer til harbor skal direkte på trucker, og resten lagres i storage.

Vi fant også ut av at vi kunne gjøre Dock klassen abstrakt å eller la shipdock og loadingdock arve fra denne klassen "base". Vi fant ut av at det var mye felles mellom de klassene og derfor valgte vi arv. I boken sier den dette om base klasser "Consider making base classes abstract even if they don't contain any abstract members. This clearly communicates to the user that the class is designed solely to be inherited from." (Cwalina; Jeremy; Brad. et al., 2020 s. 243).

DateTime and DateTimeOffset

I simulation klassen har vi en variabel som holder på "currentTime" som sier hvilke tid og dato det er i simuleringen. Vi har også mulighet til å finne ut når en dag er slutt vet å se når tiden er 00:00.
I følge boken "Do use DateTime with a 00:00:00 time component rather than DateTimeOffset to represent whole dates, such as a date of birth" (Cwalina; Jeremy; Brad. et al., 2020 s. 307).

ContainerSpace

ContainerSpace er en intern klasse i rammeverket som representerer en lagringsplass for en kontainer. Oppgaven definerer andre størrelser på kontainerne en det vi hadde i den opprinnelige utgaven av rammeverket. Vi fjernet derfor de gamle størrelsene og implementerte de nye størrelsene i form av Full og Half størrelse kontainere. Oppgaven spesifiserte også at to halfsize kontainere tar opp like mye plass som en fullsize kontainer. Vi implementerte derfor muligheten for en ContainerSpace til å holde på enten en Full kontainer eller to Half størelse kontainere. Dette gjør at ContainerSpace. Dette er I midlertidig ikke noen brukere av APIet trenger å forholde seg til siden denne klassen ikke er en del av APIet og all logikken for det som brukeren må ha ett forhold til er definert i public metodene og variablene i klassen ContainerStorageRow.

Simulation

Simulation er under større forandringer, som også jobbes aktivt med. Logikk har og blir delt opp mindre metoder med mindre ansvarsområder, som skal forbedre både leselighet og oversikt, men også lage et bedre grunnlag for utvidelse og forandringer.

Vi har separert ut if-sjekker i boolske metoder, for å gjøre if-sjekkene mer menneskelig leselig. Ved navngivingen av disse metodene har vi fulgt DO angående "affirmative phrase":

- "DO name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value." (Cwalina; Jeremy; Brad. et al., 2020 s. 76).

De største logikk-forandringene i Simulation ligger i LoadingShips() og UnloadingShips(). Det er her mesteparten av simulasjon av container-flytt skjer. Med nye krav og objekter som skal involveres i et container-flytt, tilrettelegger nå disse metodene for introduksjonen av de nye objektene (Crane, Adv, Truck).

LoadingShips() har nå fått to overordnede undermetoder:
LoadShipForOneHour() og LoadContainerOnShip().

I LoadShipForOneHour() regnes antall repetisjoner i timen (antall containere som lastes på et skip, per time) ut ifra maks flytt for kran og AGV.
Det vil si at hvis en AGV kan laste 10 i timen, og kran kan laste 20, så vil antall repetisjoner basere seg på det minste tallet; 10.

Dette tallet ganges så med antall (ledige) instanser av "det svakeste ledd".
For eksempel så har AGV maks 10 last i timen, og vi har 20 AGV-er. Vi kan dermed gjøre 10 * 20 repetisjoner i timen.
I tillegg tas antallet containere som skal på skip fra lagerplass med i beregningen, ved bruk av harbor sin interne metode NumberOfContainersInStorageToShips().

LoadContainerOnShip() står for det overordnede flyttet av container fra lagringsplass til skip. Her inngår det oppdelte metoder som tar for seg de mindre flyttene ved de forskjellige lokasjonene: En for container-flytt fra lagringsplass til AGV, og en fra AGV til

skip. Intern logikk her tar i bruk Harbor sine metoder for container-flytt, som ContainerRowToCrane og CraneToAdv.

UnloadingShips() har også fått den samme oppstykkingen som LoadingShips().

Vi har også implementert LoadingTrucksFromStorage(), som laster containere fra lagringsplass til trucker.
Hvor mange containere konstateres med en ny metode, CalculateNumberOfStorageContainersToTrucks().
Med Harbor sin metode, NumberOfContainersInStorageToTrucks(), settes riktig antall containere til variabelen numberOfStorageContainersToTrucksThisRound. Dette var viktig fordi det var tilfeller hvor harbor var tom for skip (alle skip var i Transit eller parkert), og trucker ville da fortsette å nærmest tømme lagringsplassen for containere. Denne metoden stopper dette, og kun oppdaterer numberOfStorageContainersToTrucksThisRound hvis det er skip på loadingDock. Disse skipene laster enten på eller av containere, så utregningen blir mer riktig.

I tillegg har vi tatt høyde for at containere som forlater havn, enten ved skip eller truck, kommer til en destinasjon. I InTransitShips() tas harbor sin RestockContainers() i bruk når skipet er på vei inn i havnen igjen. Slik kommer det nye containere til havnen, og de som dro ut, blir lagret i harbor sin ArrivedAtDestination liste.
ContainersOnTrucksArrivingAtDestination() håndterer dette for containere som reiser med trucker.
Slik får Harbor da en oversikt over alle containere som har vært på havnen og nå kommet seg til neste destinasjon.

Truck

Truck klassen vi har laget er en public klasse, Hovedoppgaven til truck klassen er å transportere kontainere til og fra havnen. En truck har kun mulighet til å bære på en kontainer uansett størrelse. Harbor har også mulighet til å legge truck'er i kø hvis det allerede er en truck som får lastet på eller av en kontainer, den vil også sende trucks i

transit, som vil si at trucken blir sendt ut av havnen til en ny destinasjon.

Siden en Truck kun har plass til en kontainer om gangen så tenkte vi at det ville være smart at havnen skulle kunne generere trucks etter antall kontainere som var nødt til å hentes ut av en truck. Oppgaven sier at 10% av alle kontainere skal hentes av truck'er. Vi tenkte at det kunne være tider hvor det var få kontainere og derfor var det ikke like stort behov for å ha mange truck'er inn i havnen, imens er det andre tider hvor behovet er større.

Client prosjektet

Siden vi prøver å lage ett API som er så generelt som mulig blir den konkrete implementasjonen av havnen som det spørres om i oppgaven gjort som ett eget prosjekt som bare bruker APIet til rammeverket istedenfor å ha selve rammeverket gjøre implementasjonen.

Implementasjonen av den konkrete kundehavnen er gjort i en separat Solution: ClientHarbor som oppretter en havn ifra spesifikasjonene gitt i oppgaven i klassen Program.

Det er også definert en klasse ShipNames i ClientHarbor solutionen som holder på en liste med navn på skip. Denne klassen er heller ikke en del av APIet og er bare laget for å gjøre navngivning av mange skip så enkelt som mulig for vår del i implementasjonen av kundehavnen. Det er ingen krav i APIet at denne klassen skal brukes eller at brukere av APIet skal trenge å opprette en egen klasse for skipsnavn, det er bare tilfeldigvis slik vi valgte å gjøre det i dette ene tilfelle. Ettersom navn bare er en string verdi som sendes inn til konstruktøren på skip er det opp til brukeren selv å velge hvordan de vil håndtere navngivning. De kan gjøre det på samme måte som vi har gjort det, skrive navnene inn manuelt når de oppretter skipene eller velge en helt annen måte å generere skipsnavn på.

For å generere skipsnavn i ShipNames klassen brukte vi ChatGPT for å lage en liste med 200 navn. Prompten som er brukt er "Give me a list of 200 unique ship names of famous ships from fiction or real life in the form of string values in C#. Each value in the list must be unique and not a duplicate of another value in the list." og er vist i ett

skjermbilde i del 5 Literatur i dette dokumentet. Vi har i tillegg lagt til noen av våre egne navn til listen som ChatGPT genererte.

Dette er igjen ikke noe som er en del av APIet og ikke noe det er nødvendig for brukere å gjøre.

3.4 Dokumentasjon av Nuget-pakking av rammeverk

Når vi skulle lage en Nuget pakke av rammeverket vårt brukte vi den tildelte dokumentasjonen som var i oppgaven: (https://learn.microsoft.com/en-us/nuget/quickstart/create-and-publish-a-package-using-visual-studio?tabs=netcore-cli)

Nuget CLI, path environment variables og nuget.org konto

Det første vi gjorde var å laste ned Nuget CLI og la nuget.exe filen på en valgfri plass på datamaskinen. Denne filstien la vi så inn til "path environment variables" på maskinen.

Vi opprettet også en konto på nuget.org, som skulle stå for publisering av Nuget-pakken.

Package properties

Siden vi allerede har et prosjekt, hoppet vi over "Create a class library project" og gikk videre til "Configure package properties".

Vi lot ID stå som "$(AssemblyName)", da dette ga oss ID-en "Gruppe8.HarbNet".
Vi lot også Package Version stå som "$(VersionPrefix)", da dette er versjon 1.0.0 av prosjektet. Utover det lot vi også resten stå, da det ikke var viktig informasjon for å få teste publiseringen av Nuget-pakke.

Vi fulgte deretter "Run the pack command" og bygget og deretter pakket prosjektet, så vi fikk en *.nupkg* fil.

Publisere Nuget-pakke på nugettest.org

Så var det på tide å publisere Nuget-pakken.

Vi fikk tak i API-nøkkel fra nuget.org profilen vår og tok i bruk .NET CLI kommandoen for publisering.

Det som var viktig her var å bytte ut "https://api.nuget.org/v3/index.json" med linken for nugettest, som vi fant ut var `https://int.nugettest.org`

```
dotnet nuget push Gruppe8.HarbNet.1.0.0.nupkg --api-key <nøkkel-her> --
source https://int.nugettest.org
```

Etter publisering fant vi også ut at Nuget-pakken må ha en readme-fil for å bli tilgjengelig. Dette la vi til via Nuget-pakken sin side.
Deretter var versjon 1.0.0 av API-et vårt publisert!

Linken til vår publiserte Nuget-pakke:
https://int.nugettest.org/packages/Gruppe8.HarbNet

Vær oppmerksom på at Nuget pakken bare inneholder selve rammeverket og at den konkrete implementasjonen av kundens havn ikke er med i den.

Kilder:

(Microsoft. (2023, 21 August). *Quickstart: Create and publish a NuGet package using Visual Studio (Windows only)*. https://learn.microsoft.com/en-us/nuget/quickstart/create-and-publish-a-package-using-visual-studio?tabs=netcore-cli)

Delinnlevering 4

3.5 Plan for brukertesting

Hva vi ønsket å oppnå fra brukertestene

Vi har hele veien hatt fokus på intuitiv navngiving og så brukervennlig oppsett som mulig, uten å måtte lese ekstra dokumentasjon.

- Vi fokuserer derfor i denne brukertesten på navngiving og oppsettets flyt fra start til slutt.

- Vi fokuserer også på “Ship”-konstuktørens parametere “roundTripInDays” og “isForASingleTrip”, da vi kan se at disse er noe mer domene-konkrete parametere, og kan være vanskeligere å komme opp med intuitive og brukervennlige navn. Brukertestingen i delinnlevering 2 tilsa at disse var intuitive og fine navn. Vi fortsetter likevel fokuset av disse inn i brukertesting i delinnlevering 4, for å få enda mer data om dette stemmer.
- Vi ville også se hvor intuitivt og forståelig bruken og behovet av “ContainerStorageRow” var: Er det intuitivt for bruker at en sammensetting av “ContainerStorageRow”-objekter vil lage lagringsplassen til havnen?
- Med nye parametere relatert til nye krav fra delinnlevering 3, ønsket vi også brukers input om navngivingene vi har kommet opp med er intuitive.
  Dette gjelder for eksempel Harbor-konstruktør parameterne “loadsPerAgvPerHour”, “numberOfCranesNextToLoadingDocks” og “percentageOfContainersDirectlyLoadedFromShipToTrucks”.

Oppsettet av alle deler som simuleringen trenger fra bruker, er viktig å se, og like viktig er det å se om bruker intuitivt kan starte og ikke minst ta i bruk simuleringen.

- Vi ønsker derfor å se om det er noen mangler i “Simulation”-konstruktøren, om bruker savner noe som parameter.
- I tillegg vil vi se om navngivingen av Eventene i API-et er intuitive og brukbare. Fokuset her er altså navngivingen, og ikke syntaks, da selve syntaksen rundt abonnering og slikt av Eventer ikke omhandler vårt API direkte. Vi er derfor åpne for at bruker skriver pseudokode, og i tilfeller hjelper dem noe på vei for å få opp hjelpe-funksjoner som IntelliSense, så de kan både bruke dokumentasjonen vår og få oversikt over hva som er tilgjengelig.

Planlegging av oppgaver

Vi tar utgangspunktet i et ferdig oppsett, og deler oppgavene i mindre deler, som vil bygge opp de siste oppgavene.
Oppgave 1 og 2 lager grunnlaget for oppgave 3, og sammen lager de videre grunnlaget for oppgave 4 og 5. Tanken bak dette er å gi brukeren mulighet for suksess og mestring fra starten av.

Vi tar også høyde for tilfeller hvor de relaterte oppgavene ikke blir ferdigstilt, ved å ha ferdigkodet delene som brukeren allerede har vært igjennom.
Får de til tidligere oppgaver, tar vi i bruk deres egen besvarelse fra de andre oppgavene.

Valget bak å bygge opp oppgavene på hverandre er også gjort med tanke på at det vi er opptatt av å se i hver oppgave, ikke blir overskygget av frustrasjon over oppgavens arbeidsmengde. Med en større samlet oppgave ville kanskje navngivingen bli ansett som uforståelig fordi bruker er sliten og lei av å ikke bli ferdig med oppgaven. Når en oppgave bygger på en de allerede har gjort, får bruker også følelse av at det de har gjort har betydning, og med det mer eierskap til oppgavene.

Oppgavene som ble planlagt var følgende:

1. **Opprettelse av skip:**
   Opprett ett stort skip med navn "Titanic", med 30 halv-containere og 70 full-size containere. Skipet bruker 7 dager på en tur.

   <u>Mål med oppgaven:</u>
   Med denne oppgaven ønsker vi å se hvor intuitivt hvilken klasse som må tas i bruk for disse type objektene er. Denne oppgaven vil også teste om parameterne i “Ship”-konstruktøren er intuitivt og meningsfylt for brukerne.
   Vi ser også ekstra etter forståelsen av “isForSingleTrip” og “roundTripInDays” parameterne, da disse er ganske så sentrale i selve simuleringen, og i tillegg noe vanskeligere å finne gode navngivninger til.

2. **Opprettelse av lagringsplass:**
   Du skal lage lagringsplassen til en havn. Lag 15 container rader med plass til 10 containere per rad.

   <u>Mål med oppgaven:</u>
   I denne oppgaven ser vi etter om det er intuitivt for brukeren å lage en og en rad selv, og om det var intuitivt at disse sett sammen skaper lagringsplass.
   Det var også viktig å se om navnet “ContainerStorageRow” er forståelig og

intuitivt å velge, da vi var fram og tilbake med å utelate enten Container eller Storage fra klassenavnet. *(ContainerStorageRow vs ContainerRow vs StorageRow)*

3. **Opprettelse av en havn:**

   En havn har disse detaljene:

   - 6 Loading Docks – 2 små, 3 medium, 1 stor
   - 3 Ship Docks – 2 medium, 1 stor
   - 4 loading dock kraner
   - 2 lagringsplass-kraner
   - 30 AGV-er
   - En kran tar 20 lastinger per time
   - En AGV tar 15 lastinger per time
   - 10 Trucker kommer til havnen hver time
   - 10% av containerne lastes direkte på trucker fra skip
   - 20% av containerne lastes på trucker fra lagringsplass

   Lag en havn med de spesifiserte detaljene ovenfor.

   <u>Mål med oppgaven:</u>
   Denne oppgaven introduserer en sentral konstuktør i vårt rammeverk, som krever mer informasjon fra bruker enn de foregående.

   Den vil også eksponere om de to foregående oppgavene/konstruktørene er intuitive i det større bilde av API-et, med tanke på at denne "Harbor"-konstruktøren ber om det brukeren lagde i oppgave 1 og 2 (liste med Ship og liste med ContainerStorageRows).

   Med denne oppgaven ønsker vi å se om "Harbor"-klassen er et intuitivt valg, og om flyten i bruken av konstruktøren er god.
   Med tanke på antall parametere konstruktøren har, ser vi det som ekstra viktig at disse parameterne er forståelige og så lite kompliserte som mulig.

   Vi har øynene åpne for bruken av alle parametere, men kanskje også et ekstra fokus rundt parameterne som er relatert til de nye kravene fra delinnlevering 3,

som for eksempel “loadsPerAgvPerHour”, “numberOfCranesNextToLoadingDocks” og “percentageOfContainersDirectlyLoadedFromShipToTrucks”

Vi ser i tillegg på om måten prosent føres i parameteret er intuitivt.
Vi ser det som mer naturlig å skrive 10 enn 0.10 for å presentere 10%.
Vi har også med dette valget gått for Int som datatype, som gjør at 0.10 ikke er mulig å skrive, uten å forårsake en TypeError.
Dette håper vi oppgaven vil vise oss om er riktig tenkt eller ikke, da en annerledes skrivemåte her vil ha stor betydning, både implementasjons- og simuleringsmessig.

4. **Opprettelse og kjøring av en simulering:**
   Opprett en simulering med startdato 10.04.2024 kl 00:00, og sluttdato 10.05.2024 kl 00:00. Kjør deretter simuleringen.

   <u>Mål med oppgaven:</u>
   I denne oppgaven vil vi se om klassen “Simulation” er intuitiv å velge, og om bruker ser noen manglende parametere som de intuitivt tenker på kunne vært til stede. I tillegg vil vi se hvordan de mener at man intuitivt kjører denne simuleringen.

5. **Bruk og abonnering av Eventer:**
   Abonner på eventer for følgende hendelser:
   a. Når et skip ankrer (anchoring)
   b. Når et skip laster av en kontainer
   c. Når en truck laster på en container fra lagringsplass
   d. Når en time har passert
   e. Når en dag er passert og logget i simuleringens historie

   Riktig syntaks av Event-håndteringer er ikke viktig her, og koden trenger ikke fungere.

   <u>Mål med oppgaven:</u>
   I denne oppgaven fokuserer vi på hva brukeren vil skrive for å abonnere på de gitte hendelsene – det vil si hva som er intuitivt å kalle hvert Event.

Vi ser også om brukerne forstår navngivingen av de med følgende Args-ene, og om de er intuitive og passer med navnet til eventet de tilhører.

Alle oppgavene er planlagt å bli gitt muntlig, og deretter gi bruker oppgaven som tekst, slik at de kan referere og prosessere oppgaven på egen tid.

3.6 Gjennomføring og evaluering av brukertesting

Plan for gjennomførelse

Vårt første ønske var å gjennomføre 3 tester: 2 tester med én person og 1 test med to personer. Vi fikk vite at den andre gruppen hadde 3 deltakere, så vi forandret denne planen til å være 3 tester med én person, slik at vi fikk mer data.

Vi planla skjermopptak med OBS, uten lyd og uten video. Dette vil gi brukeren mer komfort, og mindre følelse av overvåkelse. Det var dermed viktig at observatørene fikk notert ned så mange detaljer som mulig fra brukerens reaksjoner, ikke bare reaksjoner når noe var vanskelig, men også der det var lett, naturlig og intuitivt.

Resultater

Vi gjennomførte våre brukertester med den andre gruppen 10. April 2024 fra kl 12:00 i et grupperom på Høgskolen i Østfold.

Oppgave 1 – Opprettelse av skip:

Brukerne lagde på eget initiativ Ship objekter og fylte inn parameterne basert på oppgaven. Ingen hadde problemer eller tanker med oppsettet her.

Fasilitator stilte spørsmål om hva testerne tenkte “isForASingleTrip” og “roundTripInDays” sto for og betydde. Alle svarte mer eller mindre selvsikkert på betydningene. “Drar kun en vei” og “Hvor lang tid det tar fram og tilbake”.

Oppgave 2 – Opprettelse av lagringsplass:

En bruker begynte å skrive ordet “storage” intuitivt og så på hva som dukket opp mens de skrev. En annen tester så på oversikten over alle tilgjengelige klasser. Alle valgte

ContainerStorageRow, og når de så den, var de ganske klare på at denne var den de skulle bruke.

spurte om hva brukeren tenkte om navngivingen på klassen “ContainerStorageRow”, og om de ville foretrukket ContainerRow eller StorageRow i stedet.
Alle brukerne syns ContainerStorageRow var intuitive og forklarende. En bruker poengterte at sammensettingen av Container og Storage var bra for han, fordi det understreket at det var kontainere (Container) som ble lagret (Storage) et sted.

Oppgave 3 – Opprettelse av havn:

Som de andre oppgavene gikk brukerne intuitivt for Harbor klassen.

Det var også intuitivt for dem hva listOfShips og listOfContainerStorageRows var og hvor de fikk tak i denne.

Flertallet av brukerne var verbale om at Harbor-konstruktøren hadde mange parametere. De kom seg lett gjennom den, selv om de syns det var mange. Ingen ga uttrykk at dette var spesifikt noe negativt, selv ikke når spurt, men de kommenterte det. Brukerne ble også spurt av fasilitator om hva de syns om parameternavnenes lengder. Alle brukerne sa de ikke så noe problem i det, da de var skrevet med CamelCase som gjorde det leselig og var beskrivende.

Parameterne relatert til AGV, Cranes og Trucks, som “loadsPerAgvPerHour”, “numberOfCranesNextToLoadingDocks” og “percentageOfContainersDirectlyLoadedFromShipToTrucks”, forklarte og forsto hver bruker uten noen store usikkerheter.

Med tanke på hvordan brukerne førte inn prosent i prosent-parameterne, valgte alle brukerne intuitivt å skrive 10 i stedet for 0.10. Fasilitator spurte om brukerne tenkte over hvordan de førte inn prosenten. Alle brukerne sa det var bare var et intuitivt valg, og en av de grublet videre og konkluderte med at datatypen som har blitt satt for parameteret er en Int, og det understreker videre at det er 10 og ikke 0.10.

Oppgave 4 – Opprettelse og kjøring av selve simuleringen:

Alle testerne var raske på å ta i bruk "Simulation" klassen. Run() funksjonen var også veldig intuitiv, der noen av testerne til og med tenkte høyt før de skrev metoden "Det er vel "run" det da".

Fasilitator spurte brukerne om de tenkte på noe de skulle ønske de kunne føre inn i "Simulation"-konstruktøren, men ingen av brukerne hadde noen umiddelbare tanker om mangler.

Oppgave 5 – Bruk og abonnering av Eventer:

Det var viktig å få poengtert til testerne at vi her ikke tester hvordan de initialiserer og faktisk abonnerer syntaksmessig riktig. De var velkomne til å pseudokode eller kun skrive Event-navnet.

Event ser ut til å være et mer usikkert område, men dette ser ut til å være med tanke på riktig syntaks og hvordan Event funker, og ikke med API-et vårt å gjøre.

Alle brukerne var fornøyd med navngivingene på Eventene. Args-ene som fulgte med var også forståelig, da de hadde samme navn som Eventet, med suffix EventArgs som boken tilsier:

- "DO name event argument classes with the "EventArgs" suffix" (Cwalina; Jeremy; Brad. et al., 2020 s. 78).

Refleksjon

En svakhet med fremgangsmåten vår er at brukeren lærer seg API-et underveis gjennom oppgave-progresjon. Testene vil dermed ikke nødvendigvis avdekke problemer med rammeverkets oppsett i sin helhet, ved å for eksempel avsløre hvor naturlig og intuitivt start til slutt er, hvis gitt et tomt dokument som første oppgave.
Er det intuitivt at bruker må lage egne skip objekter og egne lagringsplassobjekter? Vil de finne ut av behovet for dette, *uten* dokumentasjon?

På en annen side er oppgave-progresjon og suksess viktig for brukerens motivasjon og velvære gjennom en brukertest. (M. Geitle, personlig kommunikasjon, 11. februar 2024.) Når brukeren får til en oppgave, gir dette motivasjon og driv til videre oppgaver. En stor

første oppgave kan da virke stressende og overveldende.

Konklusjon

Sesjonene viste ikke noe tegn til stussing/pausing fra brukerne ved tanken om å opprette skip eller lagringsplasser manuelt - det virket som en “naturligvis” del av oppsettet.
Brukertestene har ikke avslørt store problemer med intuitiviteten ved API-et, og heller gitt oss god tilbakemelding om at det står mye bra til med navngivinger.

- **isForASingleTrip og roundTripInDays er fortsatt fine og intuitive navngivinger.** Problematikken vi føler rundt disse, er nok mer knyttet til usikkerhet rundt domenekunnskap og ikke navngivingen selv.
- De nyeste parameterne relatert til prosenter, Trucks, AGVs og Cranes var også forståelig for testerne. Alle testerne har skjønt og intuitivt forklart på egenhånd hva disse parameterne er og betyr.
- **Harbor-konstruktøren har mange parametere.**
  Dette har vi adressert i en senere versjon, ved å i tillegg ha en “simpel” konstruktør, som også retter seg mot en regel i boken som sier:
  - “DO provide simple overloads of constructors and methods. A simple overload has a very small number of parameters, and all parameters are primitives” (Cwalina; Jeremy; Brad. et al., 2020 s. 26).

  Denne konstruktøren vil gi en alternativ metode å opprette en havn på, med mindre konfigurasjon og enda mer “plug-and-play" for brukeren.

3.7 Implementasjon av grafisk grensesnitt

Fra delinnlevering 4.

Vi valgte å bruke MAUI grensesnittet til Microsoft.

Vi valgte først å dele de forskjellige delene av konfigureringen på forskjellige sider for å få en side som kun skulle kjøre simuleringen, men vi fant ut av det kunne bli litt strevsomt for en bruker å måtte hoppe frem og tilbake på forskjellige sider for å kunne starte opp en simulering. Vi valgte derfor å legge alt på hovedsiden.

Måten vi gikk frem for å løse dette var å dele forsiden på tre like store deler. Opprettelse av harbor skulle ta 1/3 av siden imens skulle konsollen ta 2/3 av siden. På harbor delen av forsiden har vi valgt å gå for entry bokser som ligger i rader. På denne 1/3 av siden er det også satt opp en scroll mulighet slik at de kan scrolle ned kun på den delen av forsiden. I entry boksene har en placeholder text i seg slik at brukeren vet hva som skal legges inn i entry. Vi tenkte at dette ville være en bedre løsning en hvis man f.eks hadde 2 entry bokser på en rad, hvis vi hadde hatt 2 kunne det oppstå problemer ved å trykke på feil boks og lignende.

I entry boksene må bruker legge inn alle parameterne for en havn, samt skal brukeren velge hvor mange skip som ønskes å brukes i simuleringen. Hvis brukeren skriver inn ugyldige tegn, som f.eks bokstaver får de opp et display boks som sier ifra om dette når de ikke lenger har den entry'en fokusert. Når alle parameterne er skrevet inn har brukeren en knapp de skal trykke på som heter Create harbor, når denne trykkes på kommer det en tekst som sier at brukeren har laget en havn med GUID: *** ***** ******
Når ship og harbor er laget kan brukeren trykke på run simulation knappen og simuleringen starter.

I konsoll delen får brukeren se hva som skjer igjennom simuleringen. Det første de blir møtt med er at simuleringen starter og fra dagen den starter og til dagen den slutter.
Deretter får de se hva som har skjedd for hver dag:
Cotainers stored in harbor
Ships Docked To Loading docks
Ships Docked to ship docks
Ships anchored in anchorage og
ships in transit.

Det neste brukeren får se er hva de individuelle skipene gjør i denne rekkefølgen ship 0 anchored-> ship 0 docked to loading dock-> unload container-> loaded-> undocking-> transit.

Til slutt får brukeren se hva som skjedde igjennom hele simuleringen, Hvor mange containere som har vært igjennom havnen, hvor mange som forlot havnen ved bruk av lastebiler og skip og hvor mange skip som er docked.

4 Iterasjon 3 - utvikling av endelig versjon

4.1 Endelige krav

Ny for mappeinnleveringen.

Kravene til det endelige rammeverket er som følger.

Funksjonelle krav

Skip

Brukere skal kunne få informasjon om navn til hvert enkelt skip som finnes i en simulering

Brukere skal kunne hente ut informasjon om den unike IDen til hvert enkelt skip som finnes i en simulering

Brukere skal kunne hente ut informasjon om størrelsen til hvert enkelt skip som finnes i en simulering

Brukere skal kunne hente ut informasjon til tidspunktet skipene i simuleringen først starter sine tokt.

Brukere skal kunne hente ut informasjon om hvor lang tid ett skip bruker på en rundtur fra brukerens havn til sin destinasjon og tilbake.

Brukeren skal kunne hente ut informasjon om den nåværende lokasjonen til hvert enkelt skip som finnes i en simulering.

Brukeren skal kunne hente ut historisk data om lokasjonsendringer og lastendringer til skipene som finnes i en simulering.

Brukeren skal kunne hente ut informasjon om lasten, hvilke kontainere skipet har ombord og størrelsene på disse kontainerne, til hvert enkelt skip som finnes i en simulering.

Brukeren skal kunne hente ut informasjon om kontainerkapasiteten til hvert enkelt skip som finnes i en simulering.

Brukere skal kunne hente ut informasjon om maksvekten til hvert enkelt skip som finnes i en simulering.

Brukere skal kunne hente ut informasjon om base vekten, vekten som skipet har når det ikke har noen last ombord, til hvert enkelt skip som finnes i en simulering.

Brukere skal kunne hente ut informasjon om den nåværende vekten til hvert enkelt skip som finnes i en simulering.

Brukeren skal kunne lage skip objekter som kan brukes i simuleringen av en havn.

Brukeren skal kunne sette navnet til hvert enkelt skip i simuleringen når brukeren oppretter skipet.

Brukere skal kunne sette størrelsen til hvert enkelt skip i simuleringen når brukeren oppretter skipet.

Brukeren skal kunne sette hvor vidt skipet bare skal brukes til en enkeltseilas når brukeren oppretter skipet.

Brukere skal kunne sette starttidspunktet for det første seilasen skipet skal utføre når brukeren oppretter skipet.

Brukere skal kunne sette hvor lang tid en seilas for ett skip vil ta når brukeren oppretter skipet.

Brukeren skal kunne bestemme hvor mange kontainere og størrelsen til disse kontainerne som ett skip har i sitt lasterom når brukeren oppretter skipet.

Brukere skal kunne sette prosentverdien av hvor stor del av lasten som blir direkte levert til destinasjonen når brukeren oppretter skipet.

Container

Brukere skal kunne hente ut informasjon om den unike IDen til hver enkelt kontainer som finnes i en simulering.

Brukere skal kunne hente ut informasjon om størrelsen til hver enkel hver enkelt kontainer som finnes i en simulering.

Brukere skal kunne hente ut informasjon om vekten hver enkelt kontainer som finnes i en simulering.

Brukeren skal kunne hente ut historisk data om endringer i hver enkelt kontainers lokasjon gjennom en simulering.

Brukere skal kunne hente ut informasjon om nåværende posisjon til hver enkelt kontainer som finnes i en simulering.

Brukere skal kunne hente ut informasjon om den nåværende statusen til hver enkelt kontainer som finnes i en simulering.

Brukeren skal kunne sette størrelsen til en kontainer som skal brukes som finnes i en simulering.

Truck

Brukere skal kunne hente ut informasjon om den unike IDen til hver enkelt truck som finnes i en simulering.

Brukere skal kunne hente ut informasjon om den nåværende lokasjonen til hver truck som finnes i en simulering.

Brukere skal kunne hente ut informasjon om den nåværende statusen til hver truck som finnes i en simulering.
Brukere skal kunne hente ut informasjon om containeren som hver truck lager i sitt lasterom for trucks som finnes i en simulering.

Harbor

Brukere skal kunne opprette sitt eget objekt som representerer havn som skal brukes i en simulering.

Brukere skal kunne hente ut informasjon om den unike IDen til en havn som finnes i en simulering

Brukere skal kunne Hente ut informasjon om hvilke brygger som er ledige i havnen som simuleres.

Brukere skal kunne hente ut informasjon om hvilke skip som har ankommet sin destinasjon når en simulering er ferdig.

Brukere skal kunne hente ut den unike IDen til ankerplass lokasjonen til en havn.

Brukere skal kunne hente ut den unike IDen til lasten til AGVer som brukes i en simulering.

Brukere skal kunne hente ut den unike IDen til lokasjonen som representerer Trucks som er i transit i en simulering.

Brukere skal kunne hente ut informasjon om den unike IDen til lokasjonen som representerer Trucks som står i kø for å komme inn til havnen.

Brukere skal kunne hente ut informasjon om den unike IDen til lokasjonen som representerer kontainer lagringsplassen til Havna.

Brukere skal kunne hente ut informasjon om den unike IDen til lokasjonen som representerer arealet hvor bryggene til havna er lokalisert.

Brukere skal kunne hente ut informasjon om den unike IDen til lokasjonen som representerer skip som brukes i simuleringen sin destinasjon.

Brukere skal kunne angi hvilke skip som skal brukes i en simulering.

Brukere skal kunne angi hvor mange kontainerplasser som er tilgjengelig i havner de oppretter.

Brukere skal kunne angi hvor mange brygger for å laste av og på kontainere fra skip som er tilgjengelig i havner de oppretter.

Brukere skal kunne angi størrelsen til alle bryggene i havnene de oppretter.

Brukere skal kunne angi hvor mange kraner som bryggene har tilgjengelig for lasting av kontainere.

Brukere skal kunne angi effektiviteten, altså hvor mange last en kran kan gjøre i timen, for alle kraner som brukes i en simulering.

Brukere skal kunne angi hvor mange kraner som er tilgjengelig for lasting av kontainere i havnen sitt lagringsområde for kontainere.

Brukere skal kunne angi hvor mange brygger som er tilgjengelig for lagring av skip etter at de er ferdig med sine seilaser.

Brukere skal kunne angi hvor mange trucks som ankommer havnen per time.

Brukere skal kunne angi prosentverdien av hvor mange kontainere som lastes direkte fra skip ombord på trucks.

Brukere skal kunne angi prosentverdien av hvor mange kontainere som lastes direkte fra havnas lagringsplass til trucks.

Brukere skal kunne angi hvor mange AGV en havn har tilgjengelig i en simulering.
Brukere skal kunne angi effektiviteten til hver AGV, altså hvor mange last en AGV kan gjøre per time, for AGVer som brukes i en simulering.

Brukere skal kunne hente ut informasjon om den unike IDen til lagringsplassen for hver rad av kontainere som lagres i en havn.

Brukere skal kunne hente ut antall ledige lagringsplasser for kontainere i hver kontainer rad.

Brukere skal kunne hente ut informasjon om antall kontainere som er lagret i hver kontainer rad sin lagringsplass.

Simulation

Bruker skal kunne sette en valgfri starttid på simuleringen i formen YYYY/MMMM/DDDD

Bruker skal kunne sette en valgfri avslutningstid på simuleringen i formen YYYY/MMMM/DDDD

Bruker skal kunne legge til en valgfri havn til simuleringen

Bruker skal kunne se hva som har skjedd i løpet av en simulering i form av en dagslogg.

Bruker skal kunne se hvor et skip har vært i løpet av en dag.

bruker skal kunne se om skipet har lastet av kontainere.

Bruker skal kunne se om skipet har lastet på kontainere.

Bruker skal kunne se hvor skipet er til ett gitt tidspunkt.

brukeren skal kunne se historikken til et valgfritt skip.

Brukeren skal kunne se hvilke skip som ligger i Anchorage.

Brukeren skal kunne se hvilke skip som er i transit.

Brukeren skal kunne se hvilke skip som har lagt seg til en Loading Dock.

Brukeren skal kunne se hvilke skip som har lagt seg til ShipDock.

Brukeren skal kunne se hvilke containere som ligger i havnen.

Det skal være begrensninger i systemet som forhindrer brukere i å overskrive informasjon om den historiske dataen de henter ut om skipene og kontainerene som brukes i en simulering.

Bruker skal kunne se hvor en container befinner seg.

Bruker skal kunne se hvor en container har vært tidligere.

bruker skal kunne se ID'en til havnen containeren har vært på.

ToString skal returnere lettlest informasjon til bruker.

ToString skal returnere Informasjon om alle skip.

ToString skal returnere informasjon om ett enkelt skip i simuleringen.

Ikke funksjonelle krav

Rammeverket skal skrives i programmeringspråket C#.

Rammeverket skal følge retningslinjene gitt i boken Framework Design Guides Third Edition.

Rammeverket skal bruke Engelsk som standardspråk.

Rammerket skal kunne brukes av en erfaren utvikler uten å måtte slå opp i ekstern dokumentasjon.

Alle klasser skal ha beskrivende xml kommentarer som beskriver hva formålet med klassen er.

Alle metoder skal ha beskrivende xml kommentarer som beskriver hva formålet med metoden er.

Alle parametere skal ha beskrivende xml kommentarer beskriver som hva formålet med parametere er.

Alle medlemmer skal ha beskrivende xml kommentarer som beskriver hva formålet med medlemmet er.

Alle XML kommentarer skal være beskrivende nok til at erfarende utviklere ikke skal trenge å slå opp i ekstern dokumentasjon for å forstå hvordan medlemmet xml kommentaren beskriver skal brukes.

Alle exceptions skal ha en beskrivende exception tekst som beskriver feilen som oppsto og hva brukeren kan gjøre for å rette opp i feilen.

4.2 Endelig API-design

Container

Objekter av Container klassen representerer en container i simuleringen.
Denne klassen arver fra den abstrakte klassen StorageUnit.

Den holder på relevant informasjon om én container slik som IDen, historikken til containeren, størrelsen, vekten og containerens nåværende lokasjon.

Bruker kan opprette Container-objekter selv, for innsending som en liste av Containere, i en av Ship-konstruktørene. Ellers opprettes objektene typisk i skips lasterom samtidig som nye skip opprettes og på nytt når skip er på havet og skal returnere til Harbor, med nye Containere.

Følgende metoder er gitt i Container klassen:

public Guid ID { get; }
Denne metoden returnerer ett Guid-objekt som holder på den unike IDen til containeren. IDen kan brukes til å identifisere og finne spesifikke kontainere i andre metodekall.

public override ReadOnlyCollection<StatusLog> History { get; }
Denne metoden returnerer en ReadOnlyCollection med StatusLog-objekter. Denne samlingen inneholder ett StatusLog objekt for hver gang containeren har endret Status (se beskrivelse av Status enum). Listen er organisert fra første status endring til siste, der første statusendring har index 0 i listen og den siste status endringen har den siste indexen i listen. Man kan slik få en komplett oversikt over hva en container har gjort og hvor den har befunnet seg til enhver tid gjennom hele simuleringen.

public ContainerSize Size { get; }
Denne metoden returnerer en ContainerSize enum (se beskrivelse av Container Size enum for nærmere informasjon) som definerer størrelsen på kontaineren. Kontainerens

vekt er gitt ut ifra størrelsen og forskjellige container størrelser krever forskjellig størrelse på lagerplassen den kan få tildelt på havnen.

public int WeightInTonn { get; internal set; }

Denne metoden returnerer en int med vekten til containeren gitt i tonn. Vekten på en kontainer kan variere basert på størrelsen og skip har en samlet maksvekt på kontainere de kan laste ombord.

public Guid CurrentLocation { get; internal set; }

Denne metoden returnerer ett Guid objekt som holder på IDen til objektet som i dette øyeblikket lagrer containeren. Dette kan være IDen til ett skip kontaineren er lagret ombord hos, eller det kan være en lagerplass på havnen som kontaineren er lagret på mens den venter på at ett skip skal laste den ombord.

public Status GetCurrentStatus();

Denne metoden returnerer en status enum (Se enum status for mer informasjon) som indikerer statusen til skipet på det tidspunktet metoden blir kalt.

public override void PrintHistory()

Printer ut containerens historikk til konsollen, som Dato og status.
Informasjonen som blir printet inkluderer Statusendringen som har oppstått hos objektet og dato + klokkeslett.

public override String HistoryToString()

Henter en string verdi som inneholder informasjon om en containers historikk.
Inkludert i denne informasjonen er Statusendringen som har oppstått hos objektet og dato + klokkeslett.

public override String ToString()

Gir bedre og relevant ToString som inkluderer: kontainerens ID, størrelse og vekten i tonn.

ContainerStorageRow

Denne klassen representerer en lagringsplass for kontainere i havnen. Hver enkelt rad tilsvarer en rad med lagringsplasser hvor kontainere kan bli lagret. Hvor mange kontainere som kan lagres per rad velger brukeren når konstruktøren kalles. Klassen ContainerStorageRow arver fra den abstrakte klassen StorageArea.

I oppgaven spesifiseres det at havnen som kunden ønsker så skal containere lagres på en tredimensjonal flate som har rad etter rad etter rad med kontainerplasser for lagring av kontainere.

Siden vårt API håndterer posisjonene til kontainere i lister ble det da nødvendig å bruke en todimensjonal liste for å lagre posisjonen til hver enkelt lagringsplass for en kontainer. Todimensjonale lister kan fort bli forvirrende både for utviklere og brukere av systemet så istedenfor valgte vi å opprette en ny klasse ContainerStorageRow som definerer en rad med kontainer lagringsplasser.

Listen for hele lagringsområde i havnen kunne dermed forbli endimensjonal og holde på flere slike ContainerStorageRow objekter for at lagerplassen skulle representere et helt areal for lagring av kontainere. For å minimere antall attributter i konstruktøren til harbor og for å gi brukeren mest mulig kontroll over hvor mange lagringsplasser havnen har valgte vi å gjøre denne nye klassen public. Brukere vil dermed selv opprette rader og angi antall lagringsplasser for hver rad for så å sende en liste med dette inn til harbor som brukes i simuleringen. Siden ContainerStorageRow nå er en del av det offentlige API-et definerte vi også ett interface IContainerStorageRow for å spesifisere de public tilgjengelige variablene og metodene i klassen. Oppgaven sier også at hvis en kontainer av halv størrelse blir lagret i raden så må resten av kontainerne i raden være av hav størrelse. Vi gjorde derfor implementasjon for å forsikre oss om at alle kontainere i raden alltid er av samme størrelse.

ContainerStorageRow definerer følgende metoder og properties:

public Guid ID { get; }

Returnerer en unik ID i form av en Guid for storageRowen.

public int numberOfFreeContainerSpaces(ContainerSize size)

Returnerer ett tall i form av int på antallet ledige containerplasser i raden.

public ContainerSize SizeOfContainersStored();

Returnerer en ContainerSize enum som forteller hvilke størrelse det er på kontainerne som er lagret i raden.

public IList<Guid> GetIDOfAllStoredContainers();

Returnerer unike IDer i form av Guid på alle kontainere som er lagret i containerStorageRow

public String ToString();

Gir en forbedret string verdi som inneholder relevant informasjon om raden. Denne stringen inneholder StorageRow sin ID, antall containerStorageSpaces og antall kontainere som blir lagret

```
public ContainerStorageRow(int numberOfContainerStorageSpaces)
```

- NumberOfContainerStorageSpaces
  -En int verdi som representerer antall lagringsplasser som skal bli lagd for en containerStorageRow. Hvis en bruker legger inn verdien 10 vil det bli laget 10 lagringsplasser for kontainere

Harbor

Harbor definerer API-et for havn-objekter.
Klassen arver fra den abstrakte klassen Port.

Ett harbor objekt representerer en enkelt havn som kan brukes i simuleringen. Hver simulering bruker kun en havn. Harbor er objektet som driver det meste av det som skjer i simuleringen. Den holder på informasjon om skipene som brukes i simuleringen og hvor skipene befinner seg. Den har også loading docks, hvor skip laster av og på containere, ship docks der skip kan ligge til havn over lengere perioder og en

lagringsplass satt sammen av rader med containerplasser. Skip vil ankomme havnen og laste av og på kontainere før de drar ut på havet igjen, eller kaier på en ship-dock resten av simuleringen. Harbor objektet holder på all denne informasjonen om hvilke skip som er med i simuleringen og hvor de befinner seg.

Harbor objekter har en public konstruktør som ikke er gitt i interfacet.
Denne konstruktøren ser slik ut:

```
public Harbor(IList<Ship> listOfShips, int numberOfSmallLoadingDocks,
int numberOfMediumLoadingDocks, int numberOfLargeLoadingDocks,
int numberOfSmallShipDocks, int numberOfMediumShipDocks,
int numberOfLargeShipDocks, int numberOfSmallContainerSpaces,
int numberOfMediumContainerSpaces, int numberOfLargeContainerSpaces)
```

Denne konstruktøren tar 10 attributter som er som følger:

- **listOfShips:** Denne attributtene er en liste med Ship objekter av type IList<Ship> som skal brukes i simuleringen. Alle skip vil starte på ankerplassen mens de venter på å få komme til en ledig kaiplass for å kunne laste av lasten sin.
- **numberOfSmallLoadingDocks:** Denne attributtene tar en int verdi som angir hvor mange laste-kai-plasser som havnen har tilgjengelig for sip av størrelse Small (se Enum Shipsize for mer info). LoadingDocks eller laste-kai-plasser er kaiplasser der skip kan legge til og laste av og på kontainere før de drar ut på havet igjen. Hver LoadingDock kan ta ett skip av tilsvarende størrelse som seg selv.
- **numberOfMediumLoadingDocks:** Tilsvarende numberOfSmallLoadingDocks bare for loading docks av medium skipstørrelse i stedet.
- **numberOfLargeLoadingDocks:** Tilsvarende numberOfSmallLoadingDocks bare for loading docks av Large skipstørrelse i stedet.
- **numberOfSmallShipDocks:** Denne attributten tar en int verdi som angir hvor mange kaiplasser havnen skal ha tilgjengelig for skip av størrelse Small (see Enum ship size for mer info) for å kunne ligge til kai når de ikke laster av og på kontainere. Det er typisk sett skip som bare gjør en enkeltseilas som benytter seg av denne typen kaiplasser. Ett skip av denne typen vil legge seg til en ledig ShipDock som matcher sin egen størrelse når den er ferdig med å laste av lasten sin. Der vil skipet ligge til simuleringen er ferdig.

- **numberOfMediumShipDocks:** Tilsvarende numberOfSmallShipDocks bare for docks av størelse medium istede.
- **numberOfLargerShipDocks:** Tilsvarende numberOfSmallShipDocks bare for docks av størelse large istede.
- **numberOfSmallContainerSpaces:** Denne attributten tar imot en int verdi som definerer hvor mange lagerplasser (container space) for kontainere av størrelse Small (see Enum ContainerSize for mer info) havnen skal ha tilgjengelig. En containerSpace kan lagre en kontainer av tilsvarende størelse som seg selv. For eksempel kan en smal container space lagre en smal container.
- **numberOfMediumContainerSpaces:** tilsvarende numberOfSmallContainerSpaces bare for Container Spaces av størrelse medium i stedet.
- **numberOfLargeContainerSpaces:** tilsvarende numberOfSmallContainerSpaces bare for Container Spaces av størrelse large i stedet.

```
public Harbor(int numberOfShips, int numberOfHarborContainerStorageRows, int
containerStorageCapacityInEachStorageRow, int numberOfLoadingDocks,  int
numberOfCranesNextToLoadingDocks, int numberOfCranesOnHarborStorageArea, int numberOfAgvs,
int loadsPerCranePerHour = 35,
int numberOftrucksArriveToHarborPerHour = 10,
int loadsPerAgvPerHour = 25,
int percentageOfContainersDirectlyLoadedFromShipToTrucks = 10,
int percentageOfContainersDirectlyLoadedFromHarborStorageToTrucks = 15)
```

Denne konstruktøren har 12 parametere, Her er det også satt default verdier.

- **NumberOfShips**

  -Int verdi som representerer antall skip om skal bli konstruert i simuleringen hvor skipsstørrelsen vil være tilfeldig.
- **NumberOfContainerStorageRows**

  -En int verdi som representerer antall container storage rows som er tilgjengelig i havnen. Hver “row” representerer en lagringsplass hvor kontainere kan lagres.

- **ContainerStorageCapacityInEachStorageRow**

  -en int verdi som representerer hvor mange kontainere det er plass til i hver rad.
- **NumberofLoadingDocks**

  -En int verdi som representere antall loading docks som er på en gitt havn. Loadingdocks er docks hvor et skip kan laste av eller på kontainere til eller fra havnen.
- **NumberOfCranesNextToLoadingDocks**

  -En int verdi som representerer antall kraner som er plassert ved havnens loadingdock plass. Kranene vil bli brukt til å laste av og på skip
- **NumberOfCranesOnHarborStorageArea**

  -En int verdi som representere antall kraner som er plassert ved havnens lagringsplass.
- **NumberOfAgvs**

  -En int verdi som representerer antall AGV'er som er konstruert i havnen. AGV er "automated guided vehicles" som kan frakte kontainere fra plass A til plass B
- **LoadsPerCranePerHour**

  -En int verdi som representerer antall kontainere en kran kan flytte i løpet av en time. I denne konstruktøren er verdien satt til 35. Altså én kontainer kan frakte 35 kontainere i timen.
- **NumberOfTrucksArriveToHarborPerHour**

  -En int verdi som representerer antall Trucks som ankommer havnen per time. I denne konstruktøren er verdien satt til 10. Altså vil 10 trucker komme til havnen for å hente kontainere hver time.
- **LoadsPerAgvPerHour**

  -En int verdi som representerer antall kontainere en AGV kan frakte i løpet av en time. I denne konstruktøren er verdien satt til 25. Altså en AGV kan frakte 25 kontainere per time.
- **PercentageOfContainersDirectlyLoadedFromShipToTrucks**

  -En int verdi som representerer prosentdelen av kontainere som skal lastes fra et skip direkte på en truck. I denne konstruktøren er verdien satt til 10. Altså vil 10% av skipets last bli lastet på truck'er.

- **PercentageOfContainersDirectlyLoadedFromHarborStorageToTrucks**
  -En int verdi som representerer antall prosent av kontainere som ligger på havnens lagringsplass som skal hentes av Trucks. I denne konstruktøren er denne verdien satt til 15. Altså vil 15% av alle kontainere på havnens lagringsplass bli hentet av Trucks.

I tillegg er disse metodene gitt i klassen:

public Guid ID { get; }
Denne metoden returnerer ett Guid objekt som definerer den unike IDen til Harbor objektet. IDen kan brukes for å identifisere og skille harbor objekter fra hverandre.

public IList<Container> ArrivedAtDestination { get; }
Denne metoden returnerer en IList med alle containere som har ankommet den endelige destinasjonen deres.

public Guid TransitLocationID { get; }
Denne metoden returnerer ett Guid objekt som holder på den unike IDen til Transit lokasjonen. Transit er en lokasjon som ett skip kan befinne seg i. Den representerer når skip er på havet.

public Guid AnchorageID { get; }
Denne metoden returnerer ett Guid objekt som holder på den unike IDen til Anchorage. Anchorage er en lokasjon som skip kan befinne seg i. Den representerer når skip ligger til anker og venter på å få en ledig kaiplass ved havnen.

public Guid AgvCargoID { get; }
Denne metoden returnerer ett Guid objekt som representerer lokasjonen en container befinner seg i når den er under transport av en AGV.

public Guid TruckTransitLocationID { get; }
Denne metoden returnerer ett Guid objekt som representerer lokasjonen en truck er når den er under transit.

public Guid TruckQueueLocationID { get; }

Denne metoden returnerer ett Guid objekt som holder på den unike IDen til truck-køen ved havnen. Den representerer lokasjonen en truck er når den er i kø og venter på en ledig truck-loading-spot ved loading-dock.

public Guid HarborStorageAreaID { get; }
Denne metoden returnerer ett Guid objekt som representerer den unike IDen til havnens lagringsplass. Den representerer for eksempel lokasjonen til en container når den er lagret i havnens lagringsplass.

public Guid HarborDockAreaID { get; }
Denne metoden returnerer ett Guid objekt som representerer den unike IDen til kaiområdet på havnen. Den representerer for eksempel lokasjonen til en kran, om den er satt på kai-området (loading-dock området).

public Guid DestinationID { get; }
Denne metoden returnerer ett Guid objekt som representerer lokasjonen som containere har når de er på sin endelige destinasjon. Dette skjer når for en container er under transport av enten en truck eller skip, etter å ha vært på havnens lagringsplass.

public IDictionary<Guid, bool> GetAvailabilityStatusForAllLoadingDocks();
Denne metoden returnerer en liste sortert i key-value par der Key er ett Guid objekt som representerer den unike IDen til en enkel loading dock (kaiplass der skip ligger til kai for å laste av og på containere), og value er en bool verdi som har verdien True hvis loading-docken er tilgjengelig og False hvis den er opptatt. Listen inneholder slike key-value par for alle loading-dockene på havnen og kan brukes for å få en oversikt over hvilke loading-docker som er ledig eller opptatt på tidspunktet metoden blir kalt.

public IDictionary<Guid, bool> GetAvailabilityStatusForAllShipDocks();
Denne metoden returnerer en liste sortert i key-value par der Key er ett Guid objekt som representerer den unike IDen til en enkel ship-dock (kaiplass der skip ligger til kai uten å kunne laste av og på containere), og value er en bool verdi som har verdien True hvis ship-docken er tilgjengelig og False hvis den er opptatt. Listen inneholder slike key-value par for alle ship-dockene på havnen og kan brukes for å få en oversikt over hvilke ship-docker som er ledig eller opptatt på tidspunktet metoden blir kalt.

public Status GetShipStatus(Guid ShipID);

Denne metoden tar imot ett Guid objekt med IDen til ett enkelt ship objekt og returnerer en Status enum (se enum ship status for mer informasjon) som representerer statusen skipet har når metoden blir kalt.

Ett skip kan ha følgende statuser:

- **Transit**
  – Skipet er på havet.
- **Anchoring**
  – Skipet ligger til anker og venter på å få en ledig kaiplass til å kunne laste av lasten.
- **LoadingDock**
  – Skipet ligger til en lastekai hvor den laster av og/eller på kontainere.
  – Hver enkelt loadingDock har sin unike Guid.
- **ShipDock**
  – Skipet er ferdig med enkelt-seilasen sin og ligger nå til kai.
  – Hver enkelt shipDock har sin unike Guid.
- **Unloading**
  -En kran begynner å unloade skipets kontainere som er ombord.
  -hver kontainer og kran har sin unike Guid.
- **Loading**
  -En kran begynner å laste på nye kontainere på skipet.
  - hver kontainer og kran har sin unike Guid.
- **DockingToLoadingDock**
  -Skipet dock'er til en ledig loading dock.
- **DockingToShipDock**
  -skipet dock'er til en ledig ship dock

public IDictionary<Ship, Status> GetStatusAllShips();

metoden returnerer en key-value liste av typen IDictionary<Ship, Status> der key verdien er ett Ship objekt og value verdien er en Status enum (se Enum status for mer info) som beskriver statusen skipet har i øyeblikket metoden blir kalt. Man kan på denne

måten få oversikt over hva hvert enkelt skip i simuleringen gjør på tidspunktet metoden blir kalt.

public IDictionary<Ship, Status> GetStatusAllLoadingDocks();
dette metoden er av typen IDictionary og inneholder skipsobjekter og deres statuser. Ved å vite hvor de befinner seg (statusen) kan metoden finne statusen til alle Loadingdocks.

public string GetContainerStatus(Guid ContainerID);
En string verdi som inneholder informasjon om en spesifikk kontainers Status.

public string GetAllContainerStatus();

String verdi som har informasjon om alle containere i port sin status.

public IDictionary<Ship, Status> GetStatusAllShips();

En Idictionary som inneholder Skip og deres status. Gir mulighet til å hente alle statusene til skipene

public string ToString();

Returnerer en optimalisert string verdi som har relevant informasjon om havnen.

Skip

Ship klassen definerer APIet til skip objekter.
Klassen arver fra den abstrakte klassen CargoVessel.

Ett skip objekt definerer ett enkelt skip som skal brukes i simuleringen. Alle skip i simuleringen er lasteskip som kan laste kontainere. Skip kan komme i flere størrelser (se ShipSize enum for mer informasjon). Størrelsen til skipet vil definere maksvekten av lasten ett skip kan ha ombord, antall kontainere den har plass til i lageret sitt og hvilke størrelse på havneplass skipet krever for å kunne legge til kai. Et skip har også et antall dager det tar fra det drar fra havnen til skipet er tilbake igjen.

Skip kan enten opprettes for en enkeltseilaser eller for kontinuerlige seilaser.

For enkeltseilaser vil skipet forlate havnen ved startdato, komme tilbake igjen til og legge seg til en ledig loading-dock, laste av lasten sin for deretter og flytte seg til en ship-dock. Der vil den ligge ut simuleringen uten videre aktivitet.

For ett skip med kontinuerlig seilas vil skipet komme til havnen ved startdato og legge seg til en ledig loading dock, laste av lasten sin, laste på ny last for deretter og legge fra kai og dra ut på havet igjen. Skipet vil repetere denne syklusen for hele lengden av simuleringen.

```
public Ship (string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip, int roundTripInDays, IList<StorageUnit> containersToBeStoredInCargo)
```

Denne konstruktøren tar imot 6 attributter:

- **ShipName**
  -tar imot en string verdi som representerer skipets navn.
- **shipSize:** tar imot en enum ShipSize (se seksjon om Enum Shipsize for mer informasjon) som bestemmer størrelsen på skipet. Størrelsen på skipet vil definere maksvekten på den totale lasten ett skip kan ha ombord og antallet kontainere ett skip har plass til i lagerrommet, i tilleg til hvor stor kaiplass skipet trenger når det legger til kai.
- **startDate**: tar imot ett DateTime objekt som definerer tidspunktet hvor skipet først ankommer havnen. Skipet vil ankomme havnen for å laste av sin første last på datoen gitt i attributtet.
- **isForASingleTrip:** Tar imot en boolsk verdi (true/false) som sier om skipet er et enkeltseiling-skip eller ikke.
- **roundTripInDays:** tar imot en int verdi som definerer hvor mange dager det tar fra ett skip legger fra kai, drar ut på havet til det er tilbake igjen med ny last.
- **ContainersToBeStoredInCargo:** En IList som inneholder kontainer objekter. container objektene vil bli plassert i shipets last når den kommer til havnen. Hvis bruker ikke har lagt noe til i listen så er skipet tomt

```
public Ship (string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip, int roundTripInDays, int numberOfHalfContainersOnBoard, int numberOfFullContainersOnBoard)
```

Denne konstruktøren tar imot 7 attributter

- **ShipName**
  -tar imot en string verdi som representerer skipets navn.
- **shipSize:** tar imot en enum ShipSize (se seksjon om Enum Shipsize for mer informasjon) som bestemmer størrelsen på skipet. Størrelsen på skipet vil definere maksvekten på den totale lasten ett skip kan ha ombord og antallet kontainere ett skip har plass til i lagerrommet, i tilleg til hvor stor kaiplass skipet trenger når det legger til kai.
- **startDate**: tar imot ett DateTime objekt som definerer tidspunktet hvor skipet først ankommer havnen. Skipet vil ankomme havnen for å laste av sin første last på datoen gitt i attributtet.
- **isForASingleTrip:** Tar imot en boolsk verdi (true/false) som sier om skipet er et enkeltseiling-skip eller ikke.
- **roundTripInDays:** tar imot en int verdi som definerer hvor mange dager det tar fra ett skip legger fra kai, drar ut på havet til det er tilbake igjen med ny last.
- **NumberOfHalfContainersOnBoard**
  -en int verdi som sier hvor mange half size kontainere som er ombord
- **NumberOfFullContainersOnBoard**
  -en int verdi som sier hvor mange fullsize kontainere det er om bord.

I tillegg er disse propertisene og metodene gitt i Ship klassen:

public Guid ID { get; }

Denne metoden returnerer et Guid objekt som holder på den unike IDen til skipet. Denne IDen kan brukes til å identifisere det spesifikke skipet i andre metodekall.

public ShipSize ShipSize { get; }

Denne metoden returnerer ett ShipSize enum (se Enum Shipsize for mer informasjon) som definerer størrelsen på skipet. Størrelsen på skipet vil definere maksvekten på den

totale lasten ett skip kan ha ombord og antallet kontainere ett skip har plass til i lagerrommet, i tilleg til hvor stor kaiplass skipet trenger når det legger til kai.

public String Name { get; }

Denne metoden returnerer en String som henter navnet på skipet.

public DateTime StartDate { get; }

Denne metoden returnerer ett DateTime objekt som gir tidspunktet for første gang skipet ankom havnen.

public TransitStatus TransitStatus { get; }

Denne metoden returnerer en transitStatus enum som informerer om skipets transit status, de 4 verdiene er:

none = 0;

arriving = 1,

leaving = 2,

anchoring = 3

public int RoundTripInDays { get; }

Denne metoden returnerer en int verdi som sier hvor mange dager det tar fra skipet legger fra kaien med til den er tilbake igjen til havnen. Altså tiden det tar for skipet å dra ut på havet for så å komme tilbake til havnen igjen.

public Guid CurrentLocation { get; }

Denne metoden returnerer Guid til den nåværende posisjonen til skipet.

Posisjonene ett skip kan befinne seg er:

- **Transit**
  – Skipet er på havet.
- **Anchorage**
  – Skipet ligger til anker og venter på å få en ledig kaiplass til å kunne laste av lasten.
- **LoadingDock**
  – Skipet ligger til en laste kai hvor den laster av og/eller på kontainere.
  – Hver enkelt loadingDock har sin unike Guid.

- **ShipDock**
  – Skipet er ferdig med enkelt-seilasen sin og ligger nå til kai.
  – Hver enkelt shipDock har sin unike Guid.

public override ReadOnlyCollection<StatusLog> History { get; }
Denne metoden returnerer en ReadOnlyCollection med StatusLog-objekter (se beskrivelse av StatusLog for nærmere forklaring av disse objektene). Denne listen inneholder ett StatusLog objekt for hver gang skipet har endret Status (se beskrivelse av Status enum). Listen er organisert fra første status endring til siste, der første statusendring har index 0 i listen og den siste status endringen har den siste indexen i listen. Man kan slik få en komplett oversikt over hva skipet har gjort og hvor det har befunnet seg til enhver tid gjennom hele simuleringen.

public IList<Container> ContainersOnBoard { get; }
Denne metoden returnerer en IList med Container objekter som representerer containerene som skipet har i sitt lasterom i nåværende øyeblikk.

public int ContainerCapacity { get; }
Denne metoden returnerer en int verdi som representerer hvor mange containere skipet har plass til i lasterommet sitt. En kontainer vil ta opp 1 plass i lasterommet uavhengig av størrelsen på kontaineren.

public int MaxWeightInTonn { get; }
Denne metoden returnerer en int verdi som representerer hva maksvekten i tonn skipet kan være før den synker. Denne maksvekten inkluderer vekten på selve skipet pluss vekten på lasten som skipet har ombord. Hvor stor maksvekt ett skip kan ha er basert på skipets størrelse. Større skip vil ha en større maksvekt. Maksvekten til skipet kan ikke overskrides.

public int BaseWeightInTonn { get; }
Denne metoden returnerer en int verdi som representerer vekten i tonn skipet er når den ikke har noen last ombord. Altså når lasterommet er tomt. Denne vekten medgår i den nåværende vekten til skipet (CurrentWeightInTonn). Base vekten til skipet er basert på størrelsen til skipet. Større skip vil ha en større dødvekt.

public int CurrentWeightInTonn { get; }

Denne metoden returnerer en int verdi som representerer vekten i tonn som skipet i øyeblikket metoden kalles. Vekten kalkuleres med grunnlag på vekten skipet har når det er tomt (BaseWeightInTonn) + summen av vekten til alle kontainerne som skipet har i sitt lasterom. Dette tallet kan aldri overskride maksvekten skipet kan ha (MaxWeightInTonn).

public void PrintHistory()

Printer skipets hele historie til console. Informasjonen som blir printet inkluderer skipets navn, ID, dato og tid av alle status endringer og følgende status skipet hadde til de tidene.

public void HistoryToString()

Gir en String som inneholder informasjon om skipets hele historie. Informasjonen i Stringen inkluderer skipets navn, ID, dato og tid av alle status endringer og følgende status skipet hadde til de tidene.

public void ToString()

Returnerer en optimalisert string verdi som har relevant informasjon om skipet. Dette inkluderer skipets ID, navn, størrelse, startdato, round trip tid, antall containere av de forskjellige containerSizes ombord, base weight i tonn, current weight i tonn og max weight i tonn som skipet kan takle.

Truck

En truck har som oppgave å skal laste av og på containere fra en kran. I klassen fant vi ut av at det måtte inneholde en unik ID, den nåværende statusen til trucken og den måtte kunne ha et Container objekt som var knyttet til den. Vi må vite hvor truck objektet befant seg, så vi har også med location variabel. Samt måtte Trucken kunne loade og unloade kontainere.

Truck definerer følgende properties og metoder:

public Guid ID { get; internal set; }

En unik Guid som representerer et truck objekt.

public Guid Location { get; internal set; }

En unik Guid som representerer den nåværende lokasjonen truck objektet befinner seg på.

public Status Status { get; internal set; }

En Status enum som representerer den nyligste registerte status objektet som blir den nåværende statusen til truck objektet.

public Container? Container { get; internal set; }

Henter container objektet som ligger i lasteplanet til truck objektet.

SimpleSimulation

SimpleSimulation definerer API-et til Simulation klassen. SimpleSimulation klassen brukes for å lage SimpleSimulation-objekter som brukes for å kjøre simuleringen til en havn.

Simulation klassen har en konstruktør som ser slik ut:

```
public Simulation (Harbor harbor, DateTime simulationStartTime,
DateTime simulationEndTime)
```

Konstruktøren har 3 attributter:

- **harbor**: tar inn et Harbor-objekt som er havnen som skal brukes i simuleringen.
- **simulationStartTime**: tar imot ett DateTime objekt som angir tidspunktet som er første dag simuleringen skal starte på.
- **simulationEndTime**: tar imot ett DateTime objekt som angir tidspunktet som simuleringen skal kjøre til. Altså tidspunktet simuleringen skal stoppe.

I tillegg angir klassen følgende properties og metoder:

public ReadOnlyCollection<DailyLog> History { get; }

Denne metoden returnerer en ReadOnlyCollection med DailyLog-objekter (Se DailyLog for mer informasjon om disse). Denne samlingen inneholder ett DailyLog-objekt for hver dag simuleringen har pågått. Nye logg objekter opprettes og legges til listen klokken 12 om natten hver 24 time. Listen er organisert sekvensielt slik at det første log objektet som blir opprettet ligger først på index 0 og det siste log objektet som ble opprettet ligger på den høyeste indexen i listen. Man kan slik ved hjelp av denne listen få en komplett oversikt over posisjonen og statusen til alle skip og kontainere gjennom hele forløpet til simuleringen.

public IList<DailyLog> Run();

Denne metoden brukes til å starte simuleringen. Når metoden kalles vil simuleringen starte og kjøre fra tidspunktet gitt i konstruktørens start time og til tidspunktet gitt i konstruktørens end time. Den returnerer så en IList<DailyLog> med log objekter som forteller om historien til hvordan simuleringen har utspilt seg.

public void PrintShipHistory();

Denne metoden printer til konsoll hele historien til alle skip for den forrige simuleringen som har blitt utført.

public void PrintShipHistory(Ship shipToBePrinted);

Denne metoden printer til konsoll hele historien til ett enkelt skip for den forrige simuleringen som har blitt utført, basert på Ship-objektet til skipet.

public void PrintShipHistory(Guid shipID);

Denne metoden printer til konsoll hele historien til ett enkelt skip for den forrige simuleringen som har blitt utført, basert på Guid-en til skipet.

public void PrintContainerHistory();

Denne metoden printer til konsoll hele historien til alle kontainere for den forrige simuleringen som har blitt utført.

public String HistoryToString();

Denne metoden returnerer en string som inneholder historikk om alle skip i en ferdig simulering. Returnerer en tom string hvis en simulering ikke har blitt kjørt.

public String HistoryToString(Guid shipID);
Denne metoden returnerer en string som inneholder historikken til ett enkelt skip i en ferdig simulering, basert på dens Guid. Returnerer en tom string hvis en simulering ikke har blitt kjørt.

public String HistoryToString(Ship ship);

Denne metoden returnerer en string som inneholder historikken til ett enkelt skip i en ferdig simulering, basert på Ship-objektet. Returnerer en tom string hvis en simulering ikke har blitt kjørt.

public String HistoryToString(String ShipsOrContainers);

Denne metoden returnerer en string som inneholder historikken til enten alle skip eller alle containere i en ferdig simulering, basert på string parameteret fra bruker. Returnerer en tom string hvis en simulering ikke har blitt kjørt.

public String ToString();

Returnerer en optimalisert string verdi som har relevant informasjon om simuleringen.

StatusLog

StatusLog definerer API-et for log-objekter for skip og containere, og holder informasjon om historikken til til disse gjennom i simuleringen.

Hvert objekt holder informasjon om statusen til ett skip eller container på ett gitt tidspunkt. Objektene opprettes hver gang en container eller skip endrer status (arbeidsoppgave) fra en oppgave til en annen. For eksempel vil ett event objekt bli opprettet når ett skip går fra å laste av kontainere (Unloading) til å laste på nye kontainere (Loading). Disse event objektene vil bli lagt til hvert enkelt skips eller containers historie slik at de kan hentes ut for å få en oversikt over hva skipet eller kontaineren har gjort gjennom forløpet til simuleringen.

StatusLog definerer følgende metoder og properties:

public Guid Subject { get; }

Denne metoden returnerer Guid til den spesifikke containeren eller skipet som loggen er knyttet til. Hvert skip og container har en unik ID som identifiserer det skipet eller containeren. Variabelen subjekt identifiserer dette skipet eller containeren slik at vi kan vite hvilke skip eller container loggen handler om.

public Guid SubjectLocation { get; }

Denne metoden returnerer Guid til lokasjonen som Subjectet til statusloggen befinner seg i det øyeblikket event objektet ble opprettet. SubjectLocation kan for eksempel være skipet som en container befinner seg på. En havnplass som ett skip befinner seg på eller en kontainerplass på havna som en kontainer befinner seg på. De forskjellige lokasjonene er som følger:
For skip: Anchorage, Transit, LoadingDock, ShipDock.
For containere: Ship, ContainerSpace, Truck, Agv, Crane
Alle disse forskjellige lokasjonene har sin egen ID. Hvert unike skip, loading dock, ShipDock vil ha sin egen unike ID.

public DateTime Timestamp { get; }
Denne metoden returnerer ett DateTime objekt med tidspunktet som loggen ble opprettet. StatusLog-er opprettes når Subjectet de omhandler endrer status.
For eksempel hvis en container går fra å bli lastet av ett skip (Unloading) til å bli lagret på et containerspace på havnen (InStorage). Du kan dermed se ut ifra Timestamp variabelen og Statusen når Subjectet fikk sin nåværende status.

public Status Status { get; }
Denne metoden returnerer en Status enum som beskriver statusen Subjectet har fått idet StatusLog objektet opprettes. Status enumen gir informasjon om hva som skjer med, eller hva subjectet gjør i det øyeblikket Event objektet blir opprettet.
Les mer om Status enumen og de mulige statusene under Status.

public String ToString();

Returnerer en optimalisert string verdi som har relevant informasjon om statusloggen.

DailyLog

DailyLog definerer APIet til log-objekter for simuleringer per dag, og arver fra den abstrakte klassen HistoryRecord.

DailyLog klassen brukes til å lage log objekter som simuleringen bruker for å bokføre statusen til alle skip og kontainere i simuleringen på ett gitt tidspunkt, og med det ta et "snapshot" av harbor og simuleringen på slutten av den endte dagen. Simuleringen vil hvert døgn opprette et nytt DailyLog objekt som forteller om status og plassering som alle skip og containere har i det øyeblikket DailyLog-objektet ble opprettet.

DailyLog definerer følgende metoder og properties:

public DateTime Timestamp { get; }
Denne metoden returnerer ett DateTime objekt som angir tidspunktet i simuleringen logobjektet ble opprettet. Denne verdien angir altså tidspunktet som alle skip og containere hadde status og plassering som er angitt i log objektet.

public ReadOnlyCollection<Ship> ShipsInAnchorage { get; }
Denne metoden returnerer en liste av type ReadOnlyCollection<Ship> som inneholder alle skip som lå til anker på det tidspunktet hvor Log objektet ble opprettet.

Samlingen er ReadOnly, slik at bruker ikke kan manipulere den faktiske tilstanden på tidspunktet.

public ReadOnlyCollection<Ship> ShipsInTransit { get; }

Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som var på havet på det tidspunktet Log objektet ble opprettet.

Samlingen er ReadOnly, slik at bruker ikke kan manipulere den faktiske tilstanden på tidspunktet.

public ReadOnlyCollection<Ship> ShipsDockedInLoadingDocks { get; }

Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som lå til en laste-kai (Loading dock) og lastet av og på containere på det tidspunktet Log objektet ble opprettet.

Samlingen er ReadOnly, slik at bruker ikke kan manipulere den faktiske tilstanden på tidspunktet.

public ReadOnlyCollection<Ship> ShipsDockedInShipDocks { get; }
Denne metoden returnerer en liste av type IList<Ship> som inneholder alle skip som lå til en kaiplass for skip som ikke laster av eller på containere på det tidspunktet Log objektet ble opprettet.

Samlingen er ReadOnly, slik at bruker ikke kan manipulere den faktiske tilstanden på tidspunktet.

public ReadOnlyCollection<Container> ContainersInHarbor { get; }
Denne metoden returnerer en liste av type IList<Container> som inneholder alle containere som lå lagret til land på havnen i på det tidspunktet Log objektet ble opprettet.

Samlingen er ReadOnly, slik at bruker ikke kan manipulere den faktiske tilstanden på tidspunktet.

public void PrintInfoForAllShips();
Denne metoden printer informasjon om alle skip og hvor de befinner seg til konsoll.

public void PrintInfoForAllContainers);
Denne metoden skriver ut informasjon om alle containere og hvor de befinner seg til konsoll.

Enumer

Enumer er en klasse som brukes i flere programmeringsspråk. En enum inneholder konstanter, konstantene er “read only” noe som betyr at de ikke kan endres på. Hvis man har en Enum som inneholder størrelsene “small, medium, large” kan man da ha faste størrelser på en klasse. Hvis man tar skips klassen vår så bruker denne Enumen ShipSize som sier noe om størrelsen på skipet.

Enum ShipSize

Enumen ShipSize er tilgjengelig gjennom API-et. Enumen angir størrelsene skip kan ha. Alle skip har en ShipSize enum som definerer størrelsen til skipet.

Tallverdien i denne enumen representer vekt i tonn. Bruker kan dermed velge base weight og container capacity som passer til skipstørrelsen de har valgt.

Størrelsene er som følger:

- None = 0,
  – Ingen størrelse, standard enum verdi.
- Small = 5000,
  – Lite skip. Base weight: 5000 tonn, container capacity: 20.
- Medium = 50000,
  – Medium skip. Base weight: 50 000 tonn, container capacity: 50.
- Large = 100000

Enum ContainerSize

Tallverdien i denne enumen representerer vekt i tonn. Bruker kan dermed velge den vekten som passer til kontainervekten de har valgt.

- None = 0,
- Half = 10,
- Full = 20,

Enum Status

I denne enumen ligger de forskjellige statusene ett skip, truck, crane, agv og container kan ha.

- **None = 0**

  – Standardverdi for Enum
- **Anchoring**

  – Skipet holder på å ankre ved havnen
- **Anchored**

  – Skipet ligger til anker hvor det venter på å få en ledig kaiplass.
- **DockingToLoadingDock**

  – Skipet holder på å docke til en loading-dock, hvor skipet kan laste av og på kontainere.
- **DockedToLoadingDock**

  – Skipet har docket til en loading-dock, hvor skipet kan laste av og på kontainere
- **DockingToShipDock**

  – Skipet holder på å docke ved en skip-dock, hvor skipet vil bli værende uten videre aktivitet.
- **DockedToShipDock**

  – Skipet har docket til en skip-dock, hvor skipet vil bli værende uten videre aktivitet.
- **Unloading**

  – Skipet holder på å laste av (unloade) containere fra skipets lasterom.
- **UnloadingDone**

  – Skipet er ferdig med å laste av (unloade) containere fra skipets lasterom.
- **Loading**

  – Skipet holder på å laste på (loade) containere fra havnens lagringsplass til skipets lasterom.
- **LoadingDone**

  – Skipet er ferdig med å laste på (loade) containere fra havnens lagringsplass til skipets lasterom.
- **Undocking**

  – Skipet holder på å undocke fra havnen.
- **Transit**

  – For skip: I transit på havet.

  – For container: Ombord et skip som er i Transit på havet.

- **InStorage**
  – Container er lagret på havnens lagringsplass.
- **LoadingToCrane**
  – Container holder på å bli lastet på en kran.
- **UnloadingFromCraneToShip**
  – Container holder på å bli lastet fra kran og ombord et skip.
- **LoadingToTruck**
  – Container holder på å bli lastet på en Truck.
- **LoadingToAgv**
  – Container holder på å bli lastet på en AGV.
- **Queuing**
  – Truck er i truck-kø for å få en loading-dock-spot for pålasting av container.
- **ArrivedAtDestination**
  – Container har kommet fram til endelig destinasjon.

Abstrakte klasser

En abstrakt klasse er en klasse som ikke kan instansieres direkte men isteden inneholder metoder og atributter som kan arves. Når de arves av andre klasser må de implementeres i de klassene, det er da mulig å ha en abstrakt klasse som definerer forskjellige metoder og atributter for en gruppe med klasser. For eksempel et Skip. Et skip vil som regel alltid ha de samme metodene og attributtene og det er derfor en god måte å gjenbruke kode ved å lage en abstrakt klasse.

CargoVessel

Cargovessel er en abstrakt klasse som definerer public APi for cargovessels som Ship klassen. I denne abstrakte klassen er det metoder og atributter som ikke kan instansieres i klassen selv men som heller skal implementeres av underklassene.

Cargovessel krever følgende properties og metoder fra arvende klasser:

public abstract Guid ID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel ID i form av en unik GUID for et cargovessel objekt

public abstract ShipSize ShipSize { get; internal set; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel l ShipSize i form av en ShipSize enum som representerer størrelsen til CargoVessel.

public abstract string Name { get; internal set; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel Name i form av string verdi som representerer navnet til en CargoVessel.

public abstract DateTime StartDate { get; internal set; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha ha en variabel StartDate i form av Et DateTime objekt som representerer start datoen til en CargoVessel.

public abstract int RoundTripInDays { get; internal set; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel RoundTripInDays i form av En int verdi som representerer hvor mange dager det tar for en CargoVessel å dra frem og tilbake, altså en rundtur.

public abstract Guid CurrentLocation { get; internal set; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel l GetCurrentLocation i form av En unik GUID for den nåværende lokasjonen til en CargoVessel.

public abstract ReadOnlyCollection<StatusLog> History { get; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel History i form av En ReadOnlyCollection som inneholder StatusLog objekter som gir informasjon om endringer i statusen til en CargoVessel.

public abstract IList<Container> ContainersOnBoard { get; }

Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha

en variabel ContainersOnBoard i form av En Ilist som inneholder ContainerObjekter som er en liste med containere som CargoVessel har ombord.

public abstract int ContainerCapacity { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel ContainerCapacity i form av En int verdi som representerer antall Containere en CargoVessel har plass til ombord.

public abstract int MaxWeightInTonn { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel MaxWeightInTonn i form av en Int verdi som representerer maksvekten til en CargoVessel i Tonn.

public abstract int BaseWeightInTonn { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel BaseWeightInTonn i form av en Int verdi som representerer vekten til en CargoVessel uten last ombord (dødvekt)

public abstract int CurrentWeightInTonn { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra Cargovessel klassen skal ha en variabel CurrentWeightInTonn i form av en int verdi som representerer den nåværende vekten til en CargoVessel.

public abstract void PrintHistory();
Definerer en kontrakt som sier at alle Klasser som arver fra Cargovessel klassen skal ha en metode som Printer historikken til en CargoVessel i konsoll

public abstract string HistoryToString();
Definerer en kontrakt som sier at alle Klasser som arver fra Cargovessel klassen skal ha en metode som skal ha en string verdi som inneholder en CargoVessel sin historikk.

public abstract override string ToString();
Definerer en kontrakt som sier at alle Klasser som arver fra Cargovessel klassen skal ha en metode som skal ha En string verdi som inneholder informasjon om en CargoVessel.

internal CargoVessel()

Konstruktøren til en CargoVessel

Port

den abstrakte klassen port definerer public api for harbor klassen. Port inneholder metoder og egenskaper som ikke kan instansieres men istedenfor skal implementeres av underklassen harbor.

Port krever følgende properties og metoder fra arvende klasser:

public abstract Guid ID { get; internal set; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel ID i form av ett Guid objekt som definerer den unike IDen til klassen.

public abstract IDictionary<Guid, bool>
GetAvailabilityStatusForAllLoadingDocks();

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en metode som sjekker hvilke loadingdocks som er tilgjengelige og ikke. til dette blir en IDictionary som inneholder en GUID for en dock og en bool brukt.

public abstract IList<Container> ArrivedAtDestination { get; internal set; }

Definerer en kontrakt som sier at alle metoder som arver fra port klassen skal ha en variabel ArrivedAtDestination i form av en Ilist som inneholder Container objekter som har ankommet sin destinasjon.

public abstract IDictionary<Ship, Status> GetStatusAllShips();

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en metode som skal hente statusen til alle skip, ved dette blir en Idictionary som inneholder Skip og deres status brukt.

public abstract Guid AnchorageID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel AnchorageID i form av en unik GUID for en anchorage plass.

public abstract Guid TransitLocationID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel TransitLocationID i form av en unik GUId for en Transitlocation.

public abstract Guid AgvCargoID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel AgvCargoID i form av en unik GUID for en AGV'en sin cargo.

public abstract Guid TruckTransitLocationID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel TruckTransitLocationID i form av en unik GUID som representerer en truck sin transit lokasjon.

public abstract Guid TruckQueueLocationID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel TruckQueueLocationID i form av en unik GUID som representerer plassen hvor en Truck stiller seg i kø.

public abstract Guid HarborStorageAreaID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel HarborStorageID i form av en unik GUID som representerer en Port sin lagringsplass.

public abstract Guid HarborDockAreaID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en variabel HarborDockAreaID i form av en unik GUID som representerer en Port sitt kai område

public abstract override string ToString();

Definerer en kontrakt som sier at alle klasser som arver fra port klassen skal ha en

Metode som skal kunne hente all informasjonen om en port i form av en string.

internal Port() { }
Konstruktøren til en port

HistoryRecord

Den abstrakte klassen definerer et public api for HistoryRecord som for eksempel Dailylog klassen. Den inneholder informasjon og data om en havn til en gitt tid.

HistoryRecord krever følgende properties og metoder fra arvende klasser:

public abstract DateTime Timestamp { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en variabel TimeStamp i form av Et DateTime objekt som representerer Dato og klokkeslett HistoryRecord ble opprettet.

public abstract ReadOnlyCollection<Ship> ShipsInAnchorage { get; }
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en variabel ShipsInAnchorage i form av En ReadOnlyCollection med skip som inneholder skipene som var i Anchorage når HistoryRecord ble laget.

public abstract ReadOnlyCollection<Ship> ShipsInTransit { get; }
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en variabel ShipsInTransit i form av En ReadOnlyCollection med skip som inneholder skipene som var i Transit når HistoryRecord ble laget

public abstract ReadOnlyCollection<Ship> ShipsDockedInLoadingDocks { get; }
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en variabel ShipsDockedInLoadingDocks i form av En ReadOnlyCollection med skip som inneholder skipene som var i LoadingDocks når HistoryRecord ble laget

public abstract ReadOnlyCollection<Ship> ShipsDockedInShipDocks { get; }
Definerer en kontrakt som sier at alle metoder som arver fra HistoryRecord klassen skal ha en variabel ShipsDockedInShipDocks i form av En ReadOnlyCollection med skip som inneholder skipene som var i ShipDock når HistoryRecord ble laget

public abstract ReadOnlyCollection<Container> ContainersInHarbor { get; }
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en variabel ContainersInHarbor i form av En ReadOnlyCollection med container objekter som representerer kontainerne som lagres på havnens lagringsplass. når HistoryRecord ble laget

public abstract ReadOnlyCollection<Container> ContainersArrivedAtDestination { get;}
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en variabel ContainersArrivedAtDestination i form av En ReadOnlyCollection med skip som inneholder Containerne som hadde ankommet destinasjonen sin når HistoryRecord ble laget

public abstract void PrintInfoForAllShips();
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en Metode PrintInfoForAllShips som printer informasjon om alle skip til konsollen.

public abstract void PrintInfoForAllContainers();
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en Metode PrintInforForAllContainers som printer informasjon om alle kontainere i konsollen.

public abstract string HistoryToString();
Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en Metode HistoryToString som lagrer En String verdi som inneholder informasjon om alle skip på en gitt dag i simuleringen.

public abstract string HistoryToString(string ShipsOrContainers);

Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en Metode HistoryToString(String ShipsOrContainers) som lagrer En string verdi som inneholder informasjon om Skip eller kontainere, en bruker velger selv med å skrive "Ship" eller "Container"

public abstract override string ToString();

Definerer en kontrakt som sier at alle klasser som arver fra HistoryRecord klassen skal ha en Metode toString som som inneholder informasjonen som er lagret i HistoryRecord Objektet i en string verdi.

internal HistoryRecord() { }

Konstruktøren til HistoryRecord.

Simulation

Simulation klassen definerer kontrakten for det public APIet til klasser som kan brukes for å simmulere havner. Slik som for eksempel SimpleSimulation klassen. Klassen har en rekke abstrakte medlemmer og metoder som definerer hvilke av barneklassens metoder og medlemmer som skal være public tilgjengelig. Denne abstrakte klassen er også der for å gjøre det lettere for rammeværket å testes, ved for eksempel at denne klassen kan brukes til å lage fakes for Simuleringsklasser.

Simulation krever følgende properties og metoder fra arvende klasser:

public abstract ReadOnlyCollection<DailyLog> History { get; }

Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en variabel History i form av En ReadOnlyCollection med DailyLog objekter som inneholder informasjon om en dag fra simuleringen. Når alle blir lagt sammen gir det hele historikken til en simulering

public abstract IList<DailyLog> Run();

Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode Run som inneholder informasjon som går fra dag til dag i simuleringen i form av En IList med DailyLog objekter.

public abstract void PrintShipHistory();
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode PrintShipHistory som Printer historikken til et skip til konsoll

public abstract void PrintShipHistory(Ship shipToBePrinted);
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode PrintShipHistory(shipToBePrinted) som Printer historikken til et gitt skip til konsollen hvor man bruker Ship objekt

public abstract void PrintShipHistory(Guid shipID);
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode PrintShipHistory(ShipId) som Printer historikken til et gitt skip til konsoll hvor man bruker GUID'en til et skip

public abstract void PrintContainerHistory();
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode PrintContainerHistory som Printer historikken til alle containere i simuleringen til konsollen.

public abstract override string ToString();
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode toString som inneholder informasjon om starttid, slutt tid og Id'en til Harbor objektet som ble brukt i form av En string verdi.

public abstract string HistoryToString();
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode HistoryToString som inneholder all historikk for hver skip i den tidligere

simuleringen i form av en string verdi som.

public abstract string HistoryToString(string ShipsOrContainers);
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode HistoryToString(ShipsOrContainers) som inneholder historikken til et ship eller container. Bruker velger selv hvilket objekt ved å legge inn "ship" eller "container" i parameteret i form av en String verdi.

public abstract string HistoryToString(Ship ship);
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode HistoryToString(ship) som inneholder historikken til alle skip. Når man legger inn ett skipsobjekt vil historikken til dette skipet bli lagret i stringen i form av en string verdi.

public abstract string HistoryToString(Guid shipID);
Definerer en kontrakt som sier at alle klasser som arver fra Simulation klassen skal ha en Metode HistoryToString(shipID) som inneholder historikken til alle skip. Når man legger inn en shipID vil historikken til dette skipet bli lagret i stringen i form av en string verdi.

StorageArea

StorageArea klassen definerer kontrakten for det public APIet til klasser som kan brukes for å lage lagringsplasser. Slik som ContainerStorageRow klassen. Klassen har en rekke abstrakte medlemmer og metoder som definerer hvilke av barneklassens metoder og medlemmer som skal være public tilgjengelig. Denne abstrakte klassen er også der for å gjøre det lettere for rammeverket å testes, ved for eksempel at denne klassen kan brukes til å lage fakes for ContainerStorageRow.

StorageArea krever følgende properties og metoder fra arvende klasser:

public abstract Guid ID { get; }

Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en variabel ID i form av En unik GUID for et StorageArea

public abstract int NumberOfFreeContainerSpaces(ContainerSize size);

Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode NumberOfFreeContainerSpaces(size) som representerer antall ledige ContainerSpaces av typen som blir spesifisert i parameteret i form av en int verdi.

public abstract ContainerSize SizeOfContainersStored();

Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode SizeOfContainersStored som Gir informasjon om størrelsen på containerne som er lagret i et storageArea i form av ContainerSize.

public abstract IList<Guid> GetIDOfAllStoredContainers();

Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode GetIdOfAllStoredContainers som inneholder GUId for alle de containere som er lagret i StorageArea i form av en IList

public abstract override string ToString();

Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode toString som inneholder informasjonen til en StorageArea i form av En string verdi

internal StorageArea() { }

Konstruktøren til en StorageArea

StorageUnit

StorageUnit klassen definerer kontrakten for det public APIet til klasser som kan brukes for å lage StorageUnits. Slik som Containerklassen. Klassen har en rekke abstrakte medlemmer og metoder som definerer hvilke av barneklassens metoder og

medlemmer som skal være public tilgjengelig. Denne abstrakte klassen er også der for å gjøre det lettere for rammeverket å testes, ved for eksempel at denne klassen kan brukes til å lage fakes for StorageUnit.

StorageUnit krever følgende properties og metoder fra arvende klasser:

public abstract Guid ID { get; }
Definerer en kontrakt som sier at alle klasser som arver fra StorageUnit klassen skal ha en variabel ID i form av En Unik GUID for en StorageUnit

public abstract ReadOnlyCollection<StatusLog> History { get; }
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en variabe History i form av en ReadOnlyCollection som inneholder StatusLog objekter for en StorageUnit, de inneholder endringene i Statusen til en storage unit.

public abstract ContainerSize Size { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en variabel Size i form av ContainerSize Enum som tilsvarer størrelsen til en StorageArea

public abstract int WeightInTonn { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en variabel WeightInTonn i form av en int verdi som representererer en StorageUnit's vekt i tonn.

public abstract Guid CurrentLocation { get; internal set; }
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en variabel CurrentLocation i form av En unik GUID som representerer lokasjonen til en storage unit

public abstract Status GetCurrentStatus();
Definerer en kontrakt som sier at alle klasser som arver fra StorageUnit klassen skal ha

en Metode GetCurrentStatus som henter En Status enum som sier noe om den nåværende statusen til Storage Unit

public abstract void PrintHistory();
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode PrintHistory som Printer hele historikken til en StorageUnit til konsoll.

public abstract string HistoryToString();
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode HistoryToString som lagrer En string verdi som representerer Hele historikken til en StorageUnit.

public abstract override string ToString();
Definerer en kontrakt som sier at alle klasser som arver fra StorageArea klassen skal ha en Metode toString som inneholder En string Verdi som inneholder informasjon om en StorageUnit

internal StorageUnit() { }
Konstruktøren til StorageUnit

Annet

Events

Events er en av de mest bruke formen av callback. Det lar brukere ‘’subscribe’’ til en event, og events er knyttet til metoder som blir kjørt når en event blir ‘’raised’’. Når man er ferdig med å bruke en event så er det også smart å “unsubscribe” på de for å unngå problemer som kan oppstå, som minnelekkasjer og forhindrindre uventet oppførsel av programmet.

SimulationEnded
Event som raises når simuleringen når endedatoen satt, og simuleringen er ferdig. *SimulationEndedEventArgs* er det tilhørende EventArgs-et.

SimulationEndedEventArgs

EventArgs tilhørende SimulationEnded.
Inkluderer:

- ReadOnlyCollection<DailyLog> SimulationHistory
  – en collection av alle DailyLogs gjennom simuleringens kjøretid
- string Description

  – En beskrivelse av Eventet.

SimulationStarting

Event som raises når simuleringen starter, og kjører fra startdatoen satt.

SimulationStartingEventArgs er det tilhørende EventArgs-et.

SimulationStartingEventArgs

EventArgs tilhørende eventet SimulationStarting.
Inkluderer:

- Harbor HarborToBeSimulated

  – Harbor-objektet som er under simulering, og som ble sendt inn i Simulation-konstruktøren.
- DateTime StartDate

  – Dato og tidspunkt som simuleringen simulerer fra.
- string Description

  – En beskrivelse av Eventet.

OneHourHasPassed

Event som raises når simuleringen starter, og kjører fra startdatoen satt.

OneHourHasPassedEventArgs er det tilhørende EventArgs-et.

OneHourHasPassedEventArgs

EventArgs tilhørende eventet OneHourHasPassed.

Inkluderer:

- DateTime CurrentTime

  – Dato og tidspunktet når eventet ble raised.

- string Description
  – En beskrivelse av Eventet.

DayEnded

Event som raises når simuleringen starter, og kjører fra startdatoen satt.

DayEndedEventArgs er det tilhørende EventArgs-et.

DayEndedEventArgs

EventArgs tilhørende eventet DayEnded.

Inkluderer:

- DailyLog TodaysLog
  – DailyLog-objektet som inneholder informasjon om den siste dagen (24 timer).
- Dictionary<Ship, List<StatusLog>> DayReviewAllShipLogs
  – En dictionary med Ship-List par, hvor Listen er alle DailyLog objektene fra skipet historie, fra den siste dagen.
- DateTime CurrentTime
  – Dato og tidspunktet når eventet ble raised.
- string Description
  – En beskrivelse av Eventet.

BaseShipEventArgs

Base EventArgs som Ship-relaterte EventArgs arver fra.

Inkluderer:

- Ship Ship
  – Skipet som var involvert i eventet.
- DateTime CurrentTime
  – Dato og tidspunktet når eventet ble raised.
- string Description
  – En beskrivelse av Eventet.

Følgende Eventer har tilhørende EventArgs som arver fra BaseShipEventArgs.

ShipUndocking

Event som raises når et skip "undocker" fra Harbor.

ShipUndockingEventArgs er det tilhørende EventArgs-et.

ShipUndockingArgs

EventArgs tilhørende eventet ShipUndocking.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID
  – Guid-en til docken som skipet docket fra.

ShipInTransit

Event som raises når et skip er ferdig med undocking, og er nå i Transit

ShipInTransitEventArgs er det tilhørende EventArgs-et.

ShipInTransitEventArgs

EventArgs tilhørende eventet ShipInTransit.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid TransitLocationID
  – Guid-en til Transit-lokasjonen.

ShipDockingToShipDock

Event som raises når et skip har begynt å docke til en skip-dock.

ShipDockingToShipDockEventArgs er det tilhørende EventArgs-et.

ShipDockingToShipDockEventArgs

EventArgs tilhørende eventet ShipDockingToShipDock.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID
  – Guid-en til skip docken som skipet docker til.

ShipDockedToShipDock

Event som raises når et skip er ferdig med å docke til en skip-dock, og er nå helt docket.

ShipDockedToShipDockEventArgs er det tilhørende EventArgs-et.

ShipDockedToShipDockEventArgs

EventArgs tilhørende eventet ShipDockedToShipDock.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID
  – Guid-en til skip-docken som skipet nå er docket til.

ShipDockingToLoadingDock

Event som raises når et skip har begynt å docke til en loading-dock.

ShipDockingToLoadingDockEventArgs er det tilhørende EventArgs-et.

ShipDockingToLoadingDockEventArgs

EventArgs tilhørende eventet ShipDockingToLoadingDock.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID
  – Guid-en til loading-docken som skipet docker til.

ShipDockedToLoadingDock

Event som raises når et skip er ferdig med å docke til en skip-dock, og er nå helt docket.

ShipDockedToLoadingDockEventArgs er det tilhørende EventArgs-et.

ShipDockedToLoadingDockEventArgs

EventArgs tilhørende eventet ShipDockedToLoadingDock.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID
  – Guid-en til loading-docken som skipet nå er docket til.

ShipStartingLoading

Event som raises når et skip begynner prosessen med pålasting (loading) av containere.

ShipStartingLoadingEventArgs er det tilhørende EventArgs-et.

ShipStartingLoadingEventArgs

EventArgs tilhørende eventet ShipStartingLoading.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Container Container
  – Guid-en til loading-docken som skipet er docket til, og gjør pålasting fra.

ShipLoadedContainer

Event som raises når én (1) container har blitt lastet på et skip.

ShipLoadedContainerEventArgs er det tilhørende EventArgs-et.

ShipLoadedContainerEventArgs

EventArgs tilhørende eventet ShipLoadedContainer.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Container Container
  – Container objektet som har blitt lastet på skipet.

ShipDoneLoading

Event som raises når et skip er ferdig med pålasting (loading).

ShipDoneLoadingEventArgs er det tilhørende EventArgs-et.

ShipDoneLoadingEventArgs

EventArgs tilhørende eventet ShipDoneLoading.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID
  – Guid-en til loading-docken som skipet er docket til, og har gjort pålasting fra.

ShipStartingUnloading

Event som raises når et skip begynner prosessen med avlasting (unloading) av containere.

ShipStartingUnloadingEventArgs er det tilhørende EventArgs-et.

ShipStartingUnloadingEventArgs

EventArgs tilhørende eventet ShipStartingUnloading.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Guid DockID

  - Guid-en til en loading-dock som skipet er docket til, og skal avlaste til.

ShipUnloadedContainer

Event som raises når én (1) container har blitt lastet av et skip.

ShipUnloadedContainerEventArgs er det tilhørende EventArgs-et.

ShipUnloadedContainerEventArgs

EventArgs tilhørende eventet ShipUnloadedContainer.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- Container Container

  – Container objektet som har blitt lastet av skipet.

ShipAnchoring

Event som raises når et skip begynner ankring ved Harbor.

ShipAnchoringEventArgs er det tilhørende EventArgs-et.

ShipAnchoringEventArgs

EventArgs tilhørende eventet ShipAnchoring.

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- AnchorageId

  – GUID-en til en anchorage plass.

ShipAnchored

Event som raises når et skip har fullstendig ankret ved Harbor.

ShipAnchoringEventArgs er det tilhørende EventArgs-et.

ShipAnchoredEventArgs

EventArgs tilhørende eventet ShipAnchored:

Arver fra BaseShipEventArgs.

Inkluderer alle argumentene fra BaseShipEventArgs, i tillegg til:

- AnchorageId

  – GUID-en til en anchorage plass.

TruckLoadingFromHarborStorage

Event som raises når en Truck laster på container fra Harbor sin lagringsplass.

TruckLoadingFromStorageEventArgs er det tilhørende EventArgs-et.

TruckLoadingFromHarborStorageEventArgs

EventArgs tilhørende eventet TruckLoadingFromStorage

Inkluderer argumentene :

- Truck Truck

  - Truck objektet som laster på Container fra Harbor sin lagringsplass.
- DateTime CurrentTime

  – Dato og tidspunktet når eventet ble raised.
- string Description

  – En beskrivelse av Eventet.

Exceptions

Exceptions er kode som håndterer spesifike errorer som kan oppstå. Skulle en exception bli kastet av en error som har oppstått, får man en beskrivende tilbakemelding på hvorfor erroren oppsto og hvordan man kan fikse det, framfor en vanlig error melding som for eksempel ‘’undefined reference’’.
Det finnes innebygde exceptions i rammeverk som dekker de grunnleggende erroer hvor exceptions kan bli brukt, men det er også mulig å lage egne exceptions klasser skulle man ha behov for det.

CraneCantBeLoadedException

En exceptions klasse opprettet for å håndtere exceptions relatert til feil som kan oppstå ved Crane objekter. Den inneholder alle de grunnleggende konstruktørene med parameterne som boken ønsket skulle være med i egendefinerte exceptions klasser, og som foreleser har gått gjennom i forelesning.

public CraneCantBeLoadedException() { }
Oppretter nytt CraneCantBeLoadedException.

public CraneCantBeLoadedException(string message) : base(message) { }
Oppretter nytt CraneCantBeLoadedException.

public CraneCantBeLoadedException(string message, Exception innerException) : base(message, innerException) { }
Oppretter nytt CraneCantBeLoadedException.

AgvCantBeLoadedException

En exceptions klasse opprettet for å håndtere exceptions relatert til feil som kan oppstå ved AGV objekter. Den inneholder alle de grunnleggende konstruktørene med parameterne som boken ønsket skulle være med i egendefinerte exceptions klasser, og som foreleser har gått gjennom i forelesning.

public AgvCantBeLoadedException() { }
Oppretter nytt AgvCantBeLoadedException objekt.

public AgvCantBeLoadedException(string message) : base(message) { }
Oppretter nytt AgvCantBeLoadedException objekt.

public AgvCantBeLoadedException(String message, Exception innerException) :
base(message, innerException) { }
Oppretter nytt AgvCantBeLoadedException objekt.

TruckCantBeLoadedException

En exceptions klasse opprettet for å håndtere exceptions relatert til feil som kan oppstå ved Truck objekter. Den inneholder alle de grunnleggende konstruktørene med parameterne som boken ønsket skulle være med i egendefinerte exceptions klasser, og som foreleser har gått gjennom i forelesning.

public TruckCantBeLoadedException() { }
Oppretter nytt TruckCantBeLoadedException objekt.

public TruckCantBeLoadedException(string message) : base(message)
Oppretter nytt TruckCantBeLoadedException objekt.

public TruckCantBeLoadedException(String message, Exception innerException) :
base(message, innerException)
Oppretter nytt TruckCantBeLoadedException objekt.

4.3 Endelig implementasjon

Ny for mappeinnlevergitingen.

Agv

AGV klassen representere AGV kjøretøy. Det er Automated Guided Vehicles som kan levere kontainere fra punkt A til punkt B på havnområdet. For eksempel mellom kaiområdet på havnen og havnens lagerområde for kontainere.

internal Guid ID { get; set; } = Guid.NewGuid();

Henter den unike ID-en til AGVen.

internal Container? Container { get; set; }

Holder på Container objektet som er i AGV-ens last. Har mulighet til å være null.

internal Guid Location { get; set; }

Unike ID-en til AGV-ens gjeldende lokasjon.

internal Agv(Guid location)

Lager et nytt AGV-objekt, med ingen container (null) og lokasjon gitt fra parameter.

internal Guid LoadContainer(Container containerToBeLoaded)

Laster parameter containeren på AGV-en og returnerer så containerens ID.

internal Container? UnloadContainer()

Laster av AGV-ens gjeldende container ved å returnere null og og sette AGV-ens Container til null. Returnerer så Containeren.

Container

Container klassen representerer Kontainere som skal brukes i havnsimuleringen.

internal IList<StatusLog> HistoryIList { get; } = new List<StatusLog>();

Henter en IList med StatusLog objekter som inneholder status-forandringer containeren har gjort gjennom simuleringen.

internal int DaysInStorage { get; set; }

Henter antall dager Containeren har vært lagret på havnen sin lagringsplass.

internal Container(ContainerSize size, int weightInTonn, Guid currentLocation)

Lager et nytt Container-objekt, med størrelse, vekt og lokasjon gitt fra parametere.

internal Container(ContainerSize size, int weightInTonn, Guid currentLocation, Guid id, IList<StatusLog> containerHistory)

Lager et nytt Container-objekt, med størrelse, vekt, lokasjon, ID og en IList med historien til containeren.

internal void AddStatusChangeToHistory(Status status, DateTime currentTime)

Legger til et StatusLog objekt til Containerens historie (History). Metoden brukes til å loggføre informasjon om én status-forandring for Containeren.

internal void AddAnotherDayInStorage()

Legger til én dag i Containerens 'DaysInStorage'. Brukes etter endt 24 timer i simuleringen, på alle containere som er i havnen sin lagringsplass.

LoadingDock

LoadingDocks som brukes i simuleringen er docks hvor skipene kan laste på eller laste av lasten deres til en havn.

Arver fra den abstrakte klassen Dock.

internal IDictionary<Guid, Truck?> TruckLoadingSpots { get; set; } = new Dictionary<Guid, Truck?>();

Henter en oversikt over pålastningsplasser for lastebiler på kaiplassen og lastebilen som er på parkert på denne plassen. Hver pålastningsplass har sin egen ID.

internal LoadingDock(ShipSize shipSize) : base(shipSize)

Lager et nytt LoadingDock-object, som har plass til skipstørrelse gitt av parameteret. Tar i bruk klassen den arver fra, Dock, sin konstruktør.

internal Truck? AssignTruckToTruckLoadingSpot(Truck truck)

Plasserer en lastebil på en pålastningsplass på kaien, ved å legge til lastebilen i 'TruckLoadingSpots' oversikten, på en ledig plass.

internal Truck? RemoveTruckFromTruckLoadingSpot(Truck truck)

Fjerner en lastebil fra pålastningsplassen, ved å sjekke om lastebilen som er sendt inn i parameter, matcher en av lastebilene som allerede er plassert i en pålastningsplass.

ShipDock

Shipdock som brukes i simuleringen. En shipdock er en dock hvor skipene blir plassert når de er ferdige med seillaset sitt og ikke skal på noen andre seillas. Hvis ShipDock er full blir de heller sendt til Anchorage.
ShipDock er en kai som lagrer skip når de ikke har noen flere turer å gjennomføre.

internal ShipDock(ShipSize shipSize) : base(shipSize)

Lager ShipDock-objekter med plass til skipstørrelser gitt av parameteret.

Dock

En abstrakt klasse som holder på basisverdier som er felles for de forskjellige variantene av docker som finns i en havn. Det er flere typer docker som arver fra denne klassen slik som ShipDock og LoadingDock. Denne klassen er ikke ment for konkret implementasjon av en havn simulering, til dette burde en av klassene som arver fra Dock brukes istede.

internal Guid ID { get; }

Returnerer eller setter ett Guid objekt som holder på den unike IDen til docken. IDen blir brukt av simuleringen for å identifisere og skille de forskjellige dockene i havnen fra hverandre.

internal ShipSize Size { get; set; }
Returnerer eller setter en ShipSize enum som representerer størelsen på docken. En dock kan bare ta imot skip av tilsvarende størelse som seg selv. For eksempel kan ShipSize.Large docks ta imot ShipSize.Large ships.

internal bool Free { get; set; }
Returnerer eller setter en boolean verdi som indikerer hvor vidt docken er ledig for å ta imot ett nytt skip. Hvis verdien er true er docken ledig for å ta imot ett nytt skip, ellers er verdien false.

internal Guid DockedShip { get; set; }
Setter eller returnerer ett Guid objekt som holder på IDen til skipet som er fortøyd til docken. Denne verdien brukes av simuleringer for å holde rede på hvor skip befinner seg i havnen.

internal Dock(ShipSize shipSize)
Konstruktør for den abstrakte klassen Dock.
ShipSize shipSize tar en ShipSize enum som indikerer størelsen på docken som skal oprettes. Konstruktøren setter også variabelen Free til true som indikerer at docken er Free til å ta imot nye skip. Merk at denne klassen ikke er ment for konkret implementasjon. For konkret implementasjon av en dock burde en av klassene som arver fra Dock klassen benyttes istede. Slik som LoadingDock eller ShipDock.

Ship

En klasse som brukes til å definere skip som skal brukes i en simulering av en havn. Ett skip er en lastefartøy som brukes for transport av kontainere på havet. Den arver fra den

abstrakte klassen CargoVessel som definerer det offentlige APIet som skips skal implementere.

public override ShipSize ShipSize {internal set; }
Setter en ShipSize enum som bestemmer skipets størrelse. Skipets størelse bestemmer hvor mange kontainere skipet har plass til i lasterommet og hvor tungt skipets grunnvekt og maksvekt er.

public override string Name { internal set; }
Setter en string som indikerer skipets navn.

public override DateTime StartDate {internal set; }
Setter ett DateTime objekt som brukes til å angi starttidspunktet for skipets første ankomst til havnen.

internal TransitStatus TransitStatus { get; set; }
Setter en TransitStatus enum som brukes til å angi hvilken del av skipets transit til leveringsmålet den befinner seg i. Denne brukes av simuleringsklassen til å holde orden på hvor i transitløpet skipet befinner.

public override int RoundTripInDays { internal set; }
Setter en int som angir hvor mange dager det vil ta for skipet å foreta en full reise til leveringsdestinasjonen sin og tilbake igjen til havnen.

public override Guid CurrentLocation { internal set; }
Setter ett Guid objekt som holder på lokasjonsIDen til skipets nåværende possisjon. Denne brukes av simmuleringsklassen til å sette og finne ut hvor skipet befinner seg.

internal IList<StatusLog> HistoryIList { get; }
Henter en IList med StatusLog objekter som holder på informasjon om skipets statusendringer gjennom simmuleringen. Hvert StatusLog objekt holder på informasjon

om en statusendring som skipet gjennomgikk. Tilsammen holder hele listen på hele historien til skipet. Denne listen brukes av simuleringsklassen til å oppdatere skipets historie etterhvert som simmuleringen gjør endringer på skipets status.

public override int ContainerCapacity { internal set; }
Setter en int verdi som indikerer hvor mange kontainere skipet maksimalt kan ha i sitt lasterom. Dette er det rene antallet konteinere det er plass til og er uavhengig av vekten til konteinerne.

public override int BaseWeightInTonn {internal set; }
Setter en int verdi som indikerer skipets basevekt. Basevekten til skipet er hvor tongt det er når skipets lasterom er helt tomt for last. Vekten er angitt i tonn.

public override int CurrentWeightInTonn { internal set; }
Setter en int verdi som indikerer skipets nåværende vekt. Denne vekten inkluderer basevekten til skipet pluss vekten av alle konteinerne i skipets lasterom. Vekten er angitt i tonn.

internal int ContainersLoadedPerHour { get; set; }
Setter eller henter en int verdi som indikerer hvor mange kontainere skipet kan laste ombord i sitt lasterom per time. Denne brukes av simmuleringsklassen for å beregne hvor effektivt det er å laste konteinere ombord på skipet.

internal int BaseDockingTimeInHours { get; set; }
Setter eller henter en int som angir hvor mange timer det tar for skipet å legge til eller fra kai. Denne brukes av simmuleringsklassen for å beregne hvor lang tid det tar fra skipet begynner å legge til kai til den kan begynne å laste av lasten sin.

internal int BaseBerthingTimeInHours { get; set; }
Setter eller henter en int verdi som indikerer hvor mange kontainere skipet kan laste ut av lasterommet sitt hver time. Denne brukes av simmuleringsklassen til å beregne hvor effektivt det er å laste konteinere av skipet.

internal bool IsForASingleTrip { get; set; } ;

Setter eller henter en boolean verdi som indikerer om skipet bare skal på en enkelt seilas før den legger seg til kai permanent. Ett skip som er for en enkeltseilas vil permanent legge seg til en ledig skipdock som matcher skipets egen størelse når den er ferdig med seilasen. Hvis ingen slik skipdock er tilgjengelig vil skipet ankre i anchorage istede. Verdien her er True hvis skipet er ment for en enkeltseilas og false ellers.

internal bool HasBeenAlteredThisHour = false;

Henter eller setter hvor vidt skipet har gjort noe i denne timen av simmuleringen eller ikke. Denne brukes av simmuleringsklassen til å holde rede på om skipet har gjort handlingene sine denne timen i simuleringen. Verdien er true hvis skipet har utført handlinger denne timen og false ellers.

internal Ship(string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip,int roundTripInDays, Guid id, IList<Container> containersOnboard, IList<StatusLog> currentHistory)

private void SetBaseShipInformation(ShipSize shipSize)

Setter basisinformasjonen om skipet basert på skipets størrelse. Basisinformasjonen er kontainerkapasitet, basevekt, maksvekt, docking time og berthing time. Alle disse verdiene er avhengig av størrelsen på skipet.

internal void GenerateContainer(DateTime time)

Tar ett DateTime objekt som indikerer når i simuleringen handlinger ble utført. Genererer deretter ett nytt konteinerobjekt og legger det til skipets lasterom. Denne metoden brukes av simmuleringsklassen for å fylle lasterommet til skipet med nye kontainere etter at skipet har lastet av lasten sin hos leveringstedet. Dette er konteinerne som skipet vil ta med tilbake til brukerens havn.

internal int GetNumberOfContainersToTrucks(double percentTrucks)

Tar en double verdi som indikerer hvilken prosentandel av kontainerne ombord på skipet som skal direkte lastes på lastebiler.

Metoden regner deretter ut hvor mange av skipets konteinere som skal lastes direkte på trucks basert på prosentverdien den fikk inn og returnerer dette antallet i form av en intverdi.

internal int GetNumberOfContainersToStorage(double percentTrucks)
Tar en double verdi som indikerer hvilken prosentandel av kontainerne ombord på skipet som skal direkte til havnens lagringsplass.
Metoden regner deretter ut hvor mange av skipets konteinere som skal til havnens lagringsplass og returnerer antallet i form av en intverdi.

private void AddContainersOnBoard(ContainerSize containerSize, int numberOfContainersToBeAddedToStorage)
ContainerSize containerSize tar en ContainerSize enum som representerer størelsen på kontainerne metoden skal generere.
Int numberOfContainersToBeAddedToStorage tar en int som indikerer hvor mange kontainere metoden skal generere av den gitte størelsen.
Metoden bruker deretter disse verdiene til å opprette nye kontainere av den gitte størelsen og legger dem til skipets lagerom. Denne metoden brukes av simuleringsklassen til å generere nye kontainere som skipet skal ta med seg tilbake fra leveringstedet tilbake til brukerens havn.

private void CheckForValidWeight()
Sjekker om vekten til skipet ikke overskrider skipets maksvekt. Med andre ord at skipet ikke er overlastet med kontainere. Denne klassen brukes av skipets konstruktører for å sjekke at brukeren ikke overlaster skiper når skipet blir opprettet. Hvis skipet er blitt overlastet kaster den en ArgumentOutOfRange exeption som kan kastes videre av konstruktøren som kaller denne klassen.

internal StatusLog AddStatusChangeToHistory(DateTime currentTime, Guid currentLocation, Status status)
DateTime currentTime tar ett DateTime objekt som indikerer hvilket tidspunkt en statusoppdatering skjedde med skipet.

Guid currentLocation tar ett Guid objekt som inneholder lokasjons IDen til lokasjonen skipet befant seg da statusendringen intraff.
Status status tar en Status enum som representerer den nye statusen til skipet.
Metoden bruker disse verdiene til å opprette ett nytt StatusLog objekt som logfører statusendringen og legger dette objektet til skipets historie liste. Denne metoden brukes av simuleringsklassen for å loggføre skipets historie etterhvert som simmuleringen kjører.

internal Container? GetContainer(ContainerSize containerSize)
ContainerSize containerSize tar en ContainerSize enum som indikerer størelsen på kontaineren metoden skal hente ut fra skipets lasterom.
Metoden bruker denne verdien til å lete igjennom lasterommet etter en kontainer som matcher den angitte størelsen og returnerer denne kontaineren hvis den finnes. Ellers returnerer den null. Hvis man sender inn en kontainerstørelse som ikke er gyldig kaster metoden en ArgumentExeption.

internal int GetNumberOfContainersOnBoard(ContainerSize containerSize)
ContainerSize containerSize tar en ContainerSize enum som indikerer størelsen på kontainerne metoden skal telle.
Metoden bruker denne verdien til å lete igjennom skipets lasterom og telle antallet av den gitte kontainerstørelsen som finnes i lasterommet. Metoden returnerer deretter dette antallet som en int verdi.

internal bool RemoveContainer(Guid containerID)
Guid containerID tar ett Guid objekt som holder på IDen til kontaineren som skal fjernes fra skipets lasterom.
Metoden bruker denne verdien til å lete igjennom skipets lasterom of fjerner kontaineren med denne IDen fra lasterommet om den finnes. Hvis metoden finner og fjerner kontaineren returnerer den en bool verdi av true, ellers returnerer den false.

internal void AddContainer(Container container)
Container container tar ett container objekt.

Metoden legger deretter dette kontainerobjektet til skipets lasterom. Denne metoden brukes av simmuleringen til å legge til kontainere fra for eksempel kraner til skipets lasterom.

internal void SetHasBeenAlteredThisHourToFalse()

internal Status GetCurrentStatus()
Henter ut skipets nåværende status ved å sjekke hvilken status som sist ble logget i skipets History liste. Denne statusen returneres deretter av metoden. Hvis ingen simmulering har vært kjørt med skipet og skipets historie derfor er tom returnerer metoden en Status.None enum.

internal Status GetStatusAtPointInTime(DateTime time)
DateTime time tar ett DateTime objekt som holder på tidspunktet man ønsker å sjekke skipets status på.
Skipet bruker deretter denne verdien til å lete igjennom statusloggene i skipets historie liste for å finne den statusloggen som er nermest tidspunktet som ble sent inn til metoden. Skipet returnerer deretter denne statusen. Hvis skipet ikke finner noen statuser som er logget etter tidspunktet som ble sendt inn returnerer metoden en Status.None enum.

internal Container? UnloadContainer()
Tar den første kontaineren i skipets lager liste og fjerner denne fra lagerommet. Den trekker kontainerens vekt fra skipets nåværende vekt og returnerer kontaineren. Hvis skipet ikke har noen kontainere i lagerommet sitt returnerer metoden Null. Denne metoden brukes av simuleringer for å hente ut kontainere fra skipets lager. For eksempel når skipet laster av lasten sin til havnen.

ContainerSpace

En klasse som brukes for å definere en lagringsplass for en kontainer av full størelse, eller to kontainere av halv størelse. Disse lagringsplassene befinner seg i havnens lagringsområde for kontainere. ContainerSpace brukes ikke alene men heller som en del av klassen ContainerStorageRow for å indikere en lagerplass i rekken av lagerplasser som ContainerStorageRow klassen definerer.

internal Guid ID { get; } = Guid.NewGuid();
Unik Guid som representerer en container storage space.

internal ContainerSize SizeOfContainerStored { get; set; }
Gir eller setter størrelsen av containerene som kan bli lagret i denne container spacen. Hvis størrelsen er satt, er det kun containere av den størrelsen som kan lagres i container spacen.

internal bool FreeOne { get; set; }
Sjekker om den første halvdelen av containerspacen er tilgjengelig til å lagre en container. Denne halvdelen kan brukes til å lagre en half size container. Hvis en full size container skal lagres må begge halvdelene av containerspacen være ledige til å lagre en container.

interal bool FreeTwo { get; set; }
Sjekker om den andre halvdelen av containerspacen er tilgjenelig til å lagre en container. Denne halvdelen kan brukes til å lagre en half size container. Hvis en full size container skal lagres må begge halvdelene av containerspacen være ledige til å lagre en container.

internal Guid StoredContainerOne { get; set; }
Gir den unike ID'en til den full size containeren eller den første half size containeren som er lagret i containerspacen.

internal StoredContainerTwo { get; set; }
Gir den unike ID'en av den andre half size containerne som er lagret i containerspacen.

ContainerStorageRow

ContainerStorageRow brukes som en lagringsplass for kontainere i havnen for simuleringen.

internal IList<ContainerSpace> RowOfContainerSpace { get; set; } = new List<ContainerSpace>();

Gir en IList av ContainerSpace objekter som kan ble brukt til å lagre containere i ContainerStorageRow. Hvert ContainerSpace objekt har plass til å lagre en full size container eller to half size containere.

internal ContainerSpace GetContainerSpaceContainingContainer(Container container)

Henter ContainerSpace objektet brukt til å lagre den spesifiserte containeren. Den tar inn parameteret container som er objektet som skal brukes til å finne hvilke ContainerSpace den er lagret i.

internal ContainerSpace GetContainerSpaceContainingContainer(Container containerID)

Henter ContainerSpace objektet brukt til å lagre den spesifiserte containeren. Den tar inn parameteret containerID som er ID'en som skal brukes til å finne hvilke ContainerSpace den er lagret i.

internal ContainerSpace AddContainerToFreeSpace(Container container)

Legger til container objektet i tilgjengelig container storage space i container storage row. Den tar inn parameteret container som er objektet som skal legges til i tilgjengelig container storage row.

internal bool CheckIfFreeContainerSpaceExists(ContainerSize size)

Sjekker om tilgjengelig storage space eksisterer for den gitte størrelsen. Den tar inn parameteret ContainerSize enum som representerer størrelsen av containeren vi vil

finne ledig plass til.

internal ContainerSpace AddContainerToFreeSpace(Guid containerID, ContainerSize sizeOfContainer)

Legger til en container til ledig container storage space hvis det eksisterer ledig plass av samme størrelse som containeren. Den tar inn parameterne containerID som er ID'en til containeren som skal lagres i storage row, og sizeOfContainer som er størrelsen på containeren som skal legges til.

internal ContainerSpace RemoveContainerFromContainerRow(Container containerToBeRemoved)

Fjerner den spesifiserte containeren fra ContainerStorageRows storage. Den tar inn parameteret containerToBeRemoved som er container objektet som skal fjernes fra ContainerStorageRow.

internal ContainerSpace RemoveContainerFromContainerRow(Guid containerToBeRemoved)

Fjerner den spesifiserte containeren ut ifra ID'en fra ContainerStorageRow storage. Den tar inn parameterret containerToBeRemoved som er ID'en til container objektet som skal fjernes fra ContainerStorageRow.

Crane

Kraner blir brukt til å laste av og laste på kontainere fra skip, AGV'er og trucker i simuleringen.

internal Guid ID { get; set; }

Henter den unike ID'en for en kran, har også mulighet til å bruke set funksjonen.

internal Container? Container { get; set; }

Henter et container object som representerer containeren som er lagret i kranens

lagringsplass. den returnerer denne containeren, hvis det ikke er en container så returnerer den null.

internal int ContainersLoadedPerHour { get; set; }
Henter en int verdi som representerer antall containere kranen kan laste av og på iløpet av en time. Måten dette kan defineres er at en load skjer nå en kran laster en kontainer fra sin last eller den laster av en fra sin last.

internal Guid Location { get; set; }
Henter en GUID som representerer lokasjonen på hvor kranen befinner seg, Det kan være GUID til loading docks eller harborStorageArea. Den returnerer da et GUID objekt som er kranens nåværende posisjon.

internal Crane(int containersLoadedPerHour, Guid location)
Konstruktøren til en kran som brukes når man skal lage kran objekter. En kran kan brukes til å laste av og på kontainere fra skip, trucker og AGV'er. Den tar in parameterne containersLoadedPerHour som er antall loads en kran kan gjøre i timen, og location som representerer lokasjonen til kranen.

internal Guid LoadContainer(Container containerToBeLoaded)
denne funksjonen laster en kontainer til kranens last. Kontaineren kan komme fra et skip, agv eller truck.

internal Container? UnloadContainer()
Denne funskjonen laster av kontaineren som ligger i kranens last. Kontaineren kan lastes til et skip, en AGV eller en truck.

DailyLog

DailyLog klassen blir brukt til å lagre historiske data i simuleringen. Hvert objekt av denne klassen inneholder informasjon om tilstanden av havnen fra en spesifikk dag til en annen.

internal DailyLog(DateTime time, IList<Ship> shipsInAnchorage, IList<Ship> shipsInTransit, IList<Container> containersInHarbour, IList<Container> containersAtDestination, IList<Ship> shipsDockedInLoadingDocks, IList<Ship> shipsDockedInShipDocks)

Konstruktøren til DailyLog. Denne lager et DailyLog objekt som holder på historisk data om tilstanden til en simulering på en gitt dag.

Harbor

Harbor klassen blir brukt i simuleringen. En harbor representerer plassen hvor all aktiviteten i simuleringen tar plass.

internal IList<LoadingDock> allLoadingDocks = new List<LoadingDock>();
En liste som inneholder alle loading docks i harbor. En Loading dock er en plass hvor skipet kan legge til og laste av eller på kontainere. Returnerer en IList med loadingdocks.

internal IList<LoadingDock> freeLoadingDocks = new List<LoadingDock>();
En liste med alle ledige Loadingdocks, altså loading docks som ikke har skip som har lagt seg til. Returnerer en IList med ledige loading docks.

internal IDictionary<Ship, LoadingDock> shipsInLoadingDock = new Dictionary<Ship, LoadingDock>();
Alle skipene som nå har lagt til kai ved en loading dock lagres i en IDictionary med skipsobjektet og Loadingdock. Skipsobjektet er nøklene og LoadingDock er verdier.

internal IList<ShipDock> allShipDocks = new List<ShipDock>();
en liste som inneholder alle ShipDocks i havnen. Shipdocks er der skipene legger til kai når de er ferdig med å levere lasten sin og ikke har flere seillas. Returnerer en IList med shipdock objekter som representerer alle shipdocks i havnen.

internal IList<ShipDock> freeShipDocks = new List<ShipDock>();

En liste med alle ledige ShipDocks, altså Shipdocks som ikke har et skip som har lagt seg til. Returnerer en IList med Shipdock objekter som representerer alle ledige shipdocks.

internal IDictionary<Ship, ShipDock> shipsInShipDock = new Dictionary<Ship, ShipDock>();

En dictionary som inneholder Ship objekter og de ShipDock'ene de har lagt seg til. I Dictionarien er Ship objektene nøkler og ShipDock er verdier.

internal IList<Ship> Anchorage { get; } = new List<Ship>();

Henter en liste med skip som ligger ankret til anchorage akkurat nå. Anchorage er en plass hvor skipene ligger til anker imens den venter på ledige docks, eller hvis de er ferdige med seillas og ikke har flere seillas å gjennomføre. Returnerer en Ilist med Ship objekter som representerer skip ankret til anchorage.

internal IDictionary<Ship, int> ShipsInTransit { get; } = new Dictionary<Ship, int>();

En IDictionary som inneholder alle skipene som foreløpig er i transit og hvor mange dager de har vært i transit. I denne IDictionarien er Ship objekter nøklene og en int verdiene.

internal IList<Ship> AllShips { get; } = new List<Ship>();

En liste med alle skipene som er i simuleringen.

internal IList<ContainerStorageRow> AllContainerRows { get; set; }

En liste som inneholder alle ContainerStorageRows i simuleringen.

internal IDictionary<Container, ContainerStorageRow> storedContainers = new Dictionary<Container, ContainerStorageRow>();

En iDictionary som inneholder alle kontainere som ligger i Havnen sin StorageArea og hvilken ContainerStorageRow de ligger på. I denne Dictionarien er Kontainere nøkler og ContainerStorageRow verdien.

internal IList<Crane> HarborStorageAreaCranes { get; set; } = new List<Crane>();

En liste med kran objekter som befinner seg på havnens StorageArea.

internal IList<Crane> DockCranes { get; set; } = new List<Crane>();

En liste med kran objekter som befinner seg på havnens LoadingDocks.

internal IList<Agv> AgvWorking { get; set; } = new List<Agv>();

En liste med AGV objekter som for øyeblikket arbeider, altså flytter på kontainere.

internal IList<Agv> AgvFree { get; set; } = new List<Agv>();

En liste med AGV objekter som inneholder alle AGV'er som er ledige.

internal IList<Truck> TrucksInTransit { get; set; } = new List<Truck>();

En liste med alle Truck's som for øyeblikket har statusen "Transit" og er på vei til sin leveringsdestinasjon.

internal IList<Truck> TrucksInQueue { get; set; } = new List<Truck>();

En liste med Truck objekter som har lagt seg i kø for å kunne få lastet på eller av kontainere.

internal double PercentOfContainersDirectlyLoadedFromShips { get; set; }

Henter eller setter ett tall som representerer prosentandelen av kontainere som skal lastes direkte fra skip til trucks. De kontainere som ikke lastes på en truck blir kjørt til Havnens StorageArea.

internal double PercentOfContainersDirectlyLoadedFromStorageArea { get; set; }
Henter eller setter ett tall som representerer prosentandelen av kontainere som skal lastest direkte fra havnens StorageArea til en Truck. De kontainerne som ikke lastes på en truck blir lastet på ett skip.

internal int TrucksArrivePerHour { get; set; }
Henter eller setter en int verdi som representerer antall trucks som ankommer til havnen i timen.

internal int LoadsPerAgvPerHour { get; set; }
Henter eller setter antalled "loads" en AGV kan gjøre per time. En "load" kan bli sett på som når en AGV loader en container på lasteplanet eller avlaster en kontainer fra lasteplanet.

private void Initialize(IList<Ship> listOfShips, int numberOfSmallLoadingDocks, int numberOfMediumLoadingDocks, int numberOfLargeLoadingDocks, int numberOfCranesNextToLoadingDocks, int LoadsPerCranePerHour, int numberOfCranesOnHarborStorageArea, int numberOfSmallShipDocks, int numberOfMediumShipDocks, int numberOfLargeShipDocks, int percentageOfContainersDirectlyLoadedFromShipToTrucks, int percentageOfContainersDirectlyLoadedFromHarborStorageToTrucks, int numberOfAgv)
Setter informasjonen til Harbor basert på verdiene gitt i konstruktøren.

internal void CreateContainerSpaces(int numberOfContainerSpaces, int numberOfContainerRows)
Metoden lager ContainerStorageRows som skal legges til en liste med containerRows.

internal Container? ShipToCrane(Ship ship, Crane crane, DateTime currentTime)
Metoden laster en kontainer fra en skips lasteplass til en kran. Deretter returnerer den kontaineren som har blitt lastet

internal Container CraneToShip(Crane crane, Ship ship, DateTime currentTime)
Denne metoden laster en kontainer fra en gitt kran til ett skip. Når den har gjort dette returnerer den kontaineren som ble flyttet.

internal Container CraneToTruck(Crane crane, Truck truck, DateTime currentTime)
Denne metoden laster en kontainer fra en gitt kran til en gitt truck. Når den har lastet kontaineren vil den returnere den.

internal Crane? GetFreeLoadingDockCrane()
Metoden finner en ledig Kran som er på havnens Loading dock. Når denne er funnet blir den returnert, hvis det ikke er en ledig kran returneres Null.

internal void RemoveTruckFromQueue(Truck truck)
Metoden fjerner en truck fra TrucksInQueue Listen som inneholder trucks som står i kø for å få lastet på en kontainer.

internal void SendTruckOnTransit(LoadingDock loadingDock, Container container)
Metoden sender en gitt truck på transit til en leveringsdestinasjon så lenge den har en kontainer på lasteplanet.

internal Truck? SendTruckOnTransit(Container container)
metoden sender en truck til transit fra køen.

internal int NumberOfContainersInStorageToShips()
Metoden returnerer en int som indikerer antallet kontainere i simuleringen som har blitt lagret på skip.

internal int NumberOfContainersInStorageToTrucks()

Metoden returnerer en int som indikerer antallet kontainere i simuleringen som har blitt lagret på trucks.

internal Container? CraneToAgv(Crane crane, Agv agv, DateTime currentTime)
Metoden laster en kontainer fra en gitt kran til en gitt AGV. Den returnerer deretter container objektet som var lastet fra kranen til AGV'en.

internal Container AgvToCrane(Crane crane, Agv agv, DateTime currentTime)
metoden laster en kontainer fra en gitt AGV til en gitt kran. Når Lastingen er gjort så returnerer den kontaineren som ble lastet.

internal bool CraneToContainerRow(Crane crane,DateTime currentTime)
metoden laster av containeren som ligger i den gitte kranens last til en leding ContainerStorageRow i harbor StorageArea så lenge det er ledig plass. Den returnerer True hvis den har blitt lastet til ContainerStorageRow

internal Container? ContainerRowToCrane(ContainerSize size, Crane crane,DateTime currentTime)
metoden laster en kontainer etter en gitt størrelse fra ContainerStorageRow i havnens StorageArea til en gitt kran.

internal Agv? GetFreeAgv()
Metoden henter en AGV som er ledig, altså uten en kontainer på lasteplanet og returnerer den. Hvis det ikke eksisterer returnerer den Null.

internal Truck? GetFreeTruck()
Metoden henter en truck som ligger i T**rucksInQueue listen**.

internal void GenerateTrucks()
Metoden lager nye Truck objekter og legger den til TrucksInQueue listen.

internal Crane? GetFreeStorageAreaCrane()

Metoden finner en ledig kran fra HarborStorageArea altså en kran som ikke har en kontainer i lasten sin.

internal Agv? GetAgvContainingContainer(Container container)

Metoden finner den AGV'en hvor den gitte kontaineren ligger. Hvis den gitte kontaineren ikke er på en av AGV'ene returnerer den null.

internal Guid DockShipToLoadingDock(Guid shipID)

Metoden legger det gitte skipet med lik ShipID til LoadingDocks. Og returnerer GUID objekt som representerer dock'en skipet har lagt seg til, hvis dette ikke eksisterer så returnerer den en tom GUID.

internal Guid DockShipFromShipDockToLoadingDock(Guid shipID)

Metoden undocker et skip fra Shipdock og legger det til en Loading dock så lenge det finnes en ledig LoadingDock.

internal Guid DockShipToShipDock(Guid shipID)

Metoden legget et skip med den gitte GUID'en til en ledig ShipDock, Hvis dette ikke eksisterer så returnerer den et tomt GUID objekt.

internal Guid MoveShipFromAnchorageToLoadingDock(Guid shipID)

Metoden Flytter et skip med den Gitte GUID'en fra Anchorage listen til Loadingdock listen, gitt at LoadingDock har plass til skipet. Hvis ikke returnerer den en tom GUID.

internal Guid UnDockShipFromLoadingDockToTransit(Guid shipID)

metoden undocker et skip med den gitte GUID'en fra LoadingDock og sender det ut til transit.

internal Guid UnDockShipFromAnchorageToTransit(Guid shipID)

metoden "undocker" et skip med den gitte GUID fra Anchorage listen og sender det til

Transit Listen. Hvis GUID'en ikke kan matches med et skip så vil den returnere en tom GUID.

internal Ship GetShipFromAnchorage(Guid shipID)
metoden henter skipet fra anchorage med det gitte GUID, hvis den ikke finner dette skipet så kaster den en exception.

internal Ship GetShipFromLoadingDock(Guid shipID)
Metoden henter det gitte skipet med GUID fra havnens Loading dock. Hvis den ikke finner dette skipet vil den kaste en exception.

internal Ship GetShipFromShipDock(Guid shipID)
Metoden henter det gitte skipet med GUID fra havnens ShipDock. Hvis den ikke finner skipet kaster den en exception.

internal LoadingDock GetLoadingDockContainingShip(Guid shipID)
Metoden finner Loading docken hvor det gitte skipet har lagt seg til. Hvis den ikke finner denne loadingDocken blir det kastet en Exception.

internal List<Ship> DockedShipsInLoadingDock()
Metoden henter en liste som inneholder alle skip som foreløpig har lagt seg til en Loading dock i havnen.

internal void RestockContainers(Ship ship, DateTime time)
Denne metoden laster av alle kontainere fra et skips lasteplan og genererer nye kontainere og legger de til skipets lasteplan.

internal bool FreeLoadingDockExists(ShipSize shipSize)

Metoden ser om det finnes en loadingdock som er ledig og har plass til den gitte skipsstørrelsen, den returnerer enten True eller False.

internal bool FreeShipDockExists(ShipSize shipSize)
metoden ser om det finnes en ledig ShipDock hvor det er plass til den gitte skipsstørrelsen, den returnerer enten true eller false.

internal LoadingDock? GetFreeLoadingDock(ShipSize shipSize)
metoden henter en tilgjengelig LoadingDock som passer til den gitte Skipsstørrelsen og returnerer den.

internal ShipDock? GetFreeShipDock(ShipSize shipSize)
metoden henter en tilgjengelig ShipDock som passer til den gitte Skipsstørrelsen og returnerer den.

internal bool RemoveShipFromAnchorage(Guid shipID)
Metoden fjerner et skip fra AnchorageListen hvor det gitte GUID matcher Guid i anchorage listen. Hvis skipet ikke finnes returneres false.

internal Guid UnDockShipFromLoadingDockToShipDock(Guid shipID)
Metoden undocker et skip med det gitte GUID fra Loading dock til shipdock Så lenge det er ledig plass i ShipDock Listen. Hvis det ikke finnes ledig plass returneres en tom GUID.

internal Guid UnDockShipFromShipDockToLoadingDock(Guid shipID)
Metoden undocker et skip med det gitte GUID fra shipdock til Loading dock så lenge det er ledig plass i loadingdock listen. Hvis det ikke finnes ledig plass returneres en tom GUID.

internal bool RemoveLoadingDockFromFreeLoadingDocks(Guid dockID)
metoden fjerner en loading dock med den gitte GUID'en fra freeLoadingDock listen. Hvis den ikke finnes i listen returnerer den false.

internal int NumberOfFreeLoadingDocks(ShipSize shipSize)

Metoden returnerer ett tall som indikerer antall ledige loadingdocks det er for den gitte skipsstørrelsen.

internal int NumberOfFreeContainerSpaces(ContainerSize containerSize)

Metoden returnerer ett tall som indikerer antall ledige containerspaces som eksisterer i havnens StorageArea for å lagre containere til den gitte størrelsen.

internal int GetNumberOfOccupiedContainerSpaces(ContainerSize containerSize)

metoden returnerer ett tall som indikerer antallet okkuperte container spaces som eksisterer i havnens Storage Area til den gitte kontainerStørrelsen.

internal ContainerStorageRow? GetContainerRowWithFreeSpace(ContainerSize containerSize)

metoden finner en containerStorageRow som har ledig plass til den gitte kontainer størrelsen. Hvis det ikke eksisterer en containerStorageRow som har plass returnerer den null.

internal Container? GetStoredContainer(ContainerSize containerSize)

Metoden finner en kontainer som er lagret i havnens StorageArea som passer med den gitte størrelsen. Hvis det ikke eksisterer en kontainer med denne størrelsen returnerer den null.

internal void AddNewShipToAnchorage(Ship ship)

metoden legger det gitte skipet til anchorage listen og fjerner det fra transit hvis det var i transit listen.

internal bool LoadingDockIsFree(Guid dockID)

Metoden ser om det finnes en ledig loadingdock hvor det er mulig å legge til et skip.

internal IList<Container> GetContainersStoredInHarbour()

metoden returnerer en liste som inneholder alle kontainere som er lagret i havnens StorageArea.

internal IList<Ship> GetShipsInLoadingDock()

metoden returnerer en liste som inneholder alle skipene som har lagt til en loading dock.

internal IList<Ship> GetShipsInShipDock()

Metoden returnerer en liste som inneholder alle skipene som har lagt seg til en shipDock.

internal IList<Ship> GetShipsInTransit()

metoden returnerer en liste som inneholder alle skipene som ligger i transit listen.

StatusLog

StatusLog klassen blir brukt til å holde på et skips eller en kontainers historikk. Hvert objekt inneholder informasjon om subjektet på det tidspunktet subjektet undergikk en statusendring

internal StatusLog (Guid subject, Guid subjectLocation, DateTime pointInTime, Status status)

Metoden lager ett nytt StatusLog Objekt. Hvert objekt inneholder informasjon om subjektet sitt og tiden det gikk igjennom en status endring.

Truck

En klasse som representerer et truck kjøretøy. Truck frakter, loader og unloader

containere fra og til skip og AGV'er.

internal Truck (Guid location)

Konstruktøren til Truck klassen, den lager ett nytt truck objekt.

internal Guid LoadContainer(Container containerToBeLoaded)

metoden laster den gitte kontaineren på truck'ens lasteplan.

internal Container? UnloadContainer()

metoden laster av containeren som ligger på truck'ens lasteplan.

SimpleSimulation

En klasse som brukes til å definere kjøringen av simuleringen til harbor.

private DateTime startTime;

Variabelen holder på datoen og tiden simuleringen startet.

private DateTime currentTime;

variabelen holder på hvilken dato og klokkeslett det er nå i simuleringen

private DateTime endTime;

variabelen holder på dato og klokkeslett når simuleringen avsluttet.

private Harbor harbor;

variabelen holder på HarborObjektet som blir brukt i simuleringen.

private int numberOfStorageContainersToShipThisRound;

Variabelen holder på en int verdi som representerer antall containerer som ble tranportert fra storage til et skip.

private int numberOfStorageContainersToTrucksThisRound;

variabelen holder på en int verdi som representerer antall kontainere som ble transportert fra storage til en truck.

internal IList<DailyLog> HistoryIList { get; } = new List<DailyLog>();

Henter historikken for alle skip og kontainere i simuleringen i form av et DailyLog Objekt. Hvert DailyLog objekt inneholder informasjon fra en dag.

private void SetStartLocationForAllShips()

Metoden setter start lokasjonen og initierer første StatusLog for alle skip basert på deres StartTime.

private void EndOf24HourPeriod()

Metoden avslutter dagen med en log og status oppdatering, deretter “raise” den en event.

private void AnchoringShips()

Metoden ankrer skip til anchorage og legger det i anchorage listen.Statusen til skipet blir også endret til anchoring.

private bool ShipIsAnchoring(Ship ship, StatusLog lastStatusLog)

denne metoden utfører en enkel sjekk for å se om skipets status er Anchoring.

private StatusLog GetStatusLog(Ship ship)

denne metoden henter det siste StatusLog objektet fra det gitte skipets historikk.

private StatusLog? GetSecondLastStatusLog(Ship ship)

Denne metoden returnerer det nest siste StatusLog Objektet fra skipets Historikk.

private void DockingShips()

Denne metoden Docker ett skip, Den setter status til Docking så lenge den ikke allerede har dette.

private static bool EmptySingleTripShip(Ship ship)

Metoden sjekker om et skip er tomt for last og er ett SingleTripShip.

private bool SingleTripShipAnchoring(Ship ship)

Metoden sjekker om det gitte skipet er et singleTripShip som har statusen Anchoring.

private bool NonEmptyShipReadyForLoadingDock(Ship ship)

Metoden sjekker om det gitte skipet er tomt og at det eksisterer en ledig LoadingDock for det.

private static bool ShipCanDock(Ship ship, StatusLog lastStatusLog)

Metoden sjekker om statusen til ett skip er akseptabelt for docking.

private void ShipAnchoringToAnchoring(Ship ship)

metoden utfører Anchoring for det gitte skipet ved å sette transitStatus til Anchoring.

private void ShipNowDockingToShipDock(Ship ship, Guid shipID)

metoden utfører Docking til Ship, dette gjør den ved å endre statusen på skipet og invoker ShipDockingtoShipDock event.

private static bool ShipHasNoContainers(Ship ship)

metoden sjekker om det gitte skipet er tomt for kontainere.

private bool FreeShipDockExists(Ship ship)

Metoden sjekker om det finnes en ledig dock som passer størrelsen til det gitte skipet.

private void StartLoadingProcess(Ship ship, Guid dockID)
denne metoden starter Loading prosessen for det gitte skipet ved å endre statusen til skipet og invoke ShipStartingLoading eventet.

private static bool SingleTripShipInTransit(Ship ship)
metoden sjekker om det gitte skipet er et singleTripShip og har statusen “Transit”

private void StartUnloadProcess(Ship ship, Guid dockID)
Denne metoden starter avlastnings prosessen for det gitte skipet. Den setter statusen til skipet til “unloading” og invoker eventet ShipStartingUnloading.

private bool ShipDockingForMoreThanOneHour(StatusLog lastStatusLog)
Denne metoden sjekker om et skip har prøvd å docke i over en time.

private bool ShipFromTransitWithContainersToUnload(Ship ship)
Denne metoden sjekker om skipet kommer fra transit (status “transit”) og at det ikke er tomt på lasteplanet.

private void ShipIsDockingToLoadingDock(Ship ship, Guid dockID)
Denne metoden Setter skipsStatusen til DockingToLoadingDock og invoker eventet ShipDockingToLoadingDock.

private void ShipIsDockedToLoadingDock(Ship ship, Guid dockID)
Denne metoden Setter skipsStatusen til DockedToLoadingDock og invoker eventet ShipDockedToLoadingDock.

private static bool ShipCanBeAltered(Ship ship, StatusLog lastStatusLog)
denne metoden sjekker om skipet har blitt endret på eller blitt gjort noe med den siste timen.

private void UnloadingShips()

Denne metoden starter unloading Prossesen fra skipet til havnen.

private bool IsShipReadyForUnloading(Ship ship, StatusLog lastStatusLog, StatusLog secondLastStatusLog)

Denne metoden sjekker om skipet oppnår kriteriene som trengs for å kunne starte å unloade lasten den har på lasteplanet.

private bool ShipNowUnloading(Ship ship, StatusLog lastStatusLog)

denne metoden sjekker om det gitte skipet allerede unloader kontainere.

private void ShipFinishedUnloading(Ship ship, Guid currentLocation)

metoden endrer Statusen på det gitte skipet til “UnloadingDone” og invoker ShipDoneUnloading eventet.

private bool ShipIsDockedToLoadingDock(StatusLog lastStatusLog)

Metoden sjekker om et skip har docket til LoadingDock.

private void UpdateShipStatus(Ship ship, StatusLog lastStatusLog, StatusLog? secondLastStatusLog)

Denne metoden oppdaterer det gitte skipets status og legger statusendringen inn i skipets historikk.

private void UnloadShipForOneHour(Ship ship)

Denne metoden unloader containere fra spesifisert skip sitt cargo til harbor i en time.

private bool ShipIsDockingToShipDock(StatusLog lastStatusLog)

Denne metoden sjekker om et skip er i prosessen av å docke til en shipDock

private bool ShipIsFinishedLoadingContainers(StatusLog lastStatusLog)

Denne metoden sjekker om et skip er ferdig med å unloade kontainere fra lasteplanet sitt.

private bool ShipIsUndocking(StatusLog lastStatusLog)

denne metoden sjekker om et skip er i prossessen av å undocke.

private static bool ContainsTransitStatus(Ship ship)

Denne metoden sjekker om det gitte skipet har statusen “Transit” og returnerer deretter dette.

private void LoadingShips()

denne metoden utfører Loading til skip, den sjekker om den oppfyller kravene for å kunne loades og utfører deretter loading prosessen.

private void ShipNowUndocking(Ship ship, Guid currentLocation)

Denne metoden starter prosessen av å undocke for det gitte skipet, Historikken til det gitte skipet blir deretter endret og det invoker ShipUndocking.

private void ShipFinishedLoading(Ship ship, Guid currentLocation)

Denne metoden sjekker om spesifisert skip i en gitt lokasjon er ferdig med å loade containere. Hvis så, så legges det til en status change i skipets history.

private void LoadShipForOneHour(Ship ship, Guid currentLocation)

Denne metoden loader et spesifisert skip i en gitt lokasjon med containere for en time.

internal Container? LoadContainerOnShip(Ship ship)

Denne metoden loader en container fra Crane til et spesifisert skip.

internal Container? LoadContainerOnStorageAgv(Ship ship)

Denne metoden loader en container fra Agv til et spesifisert skip.

private Container? MoveOneContainerFromContainerRowToAgv(ContainerSize containerSize)

Denne metoden flytter en container av spesifisert størrelse fra ContainerRow til en Agv

private Container? MoveOneContainerFromAgvToShip(Container? container, Ship ship)

Denne metoden flytter den spesifiserte containeren fra Agv til det spesifiserte skipet.

private void InTransitShips()

Denne metoden sjekker om det er noen skip som for øyeblikket er i transit og er klare for å ankomme harbor, hvis så, så blir skipet lagt til i anchorage.

private bool IsShipInTransit(Ship ship, StatusLog lastStatusLog)

Denne metoden sjekker det spesifiserte skipets status for å se om det er i transit.

private bool ShipHasReturnedToHarbor(Ship ship, StatusLog LastHistoryStatusLog)

Denne metoden sjekker det spesifiserte skipet status for å se om det har returnert til harbor.

private void LoadingTrucksFromStorage()

Denne metoden loader truck med containere from harbor storage area.

private void ContainersOnTrucksArrivingToDestination()

Denne metoden sjekker om containere har ankommet med truck til deres endelige destinasjon. Hvis så, så oppdateres containerens History med ny status.

private int CalculateNumberOfStorageContainersToTrucks()

Denne metoden gir et nummer som representerer mengden containere i harbor storage area som skal loades direkte til trucks.

private Container? MoveOneContainerFromContainerRowToTruck(Container

container)

Denne metoden flytter den spesifiserte containeren fra harbor storage til ledig truck.

5. Ekstraoppgaver

5.1 Evaluering for forbedring av tildelt gruppes API - Første runde

Fra delinnlevering 2.

(Se eksternt dokument)

5.2 Evaluering for forbedring av tildelt gruppes API - Andre runde

Fra delinnlevering 4.

(Se eksternt dokument)

5.2 Refleksjon og diskusjon rundt Microsofts dokumentasjon

I løpet av prosjektet har Microsoft sin dokumentasjon vært essensiell i utviklingen av GUI, Ettersom at vi aldri har jobbet med dette før. Problemet som er med å lage dokumentasjon er at det alltid er rom for forbedringer og i denne refleksjonen skal vi ta utgangspunkt i siden for data binding (https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/data-binding).

det første man møter når man skal se på databinding dokumentasjonen er en oversikt over hva Data binding er og hva det kan gjøre. Dette ga oss ett godt innblikk i hvordan det fungerte rent teknisk. Dette er en Side vi mener burde bli slik som den er mtp. At

man lett kan forstå hensikten med denne delen av dokumentasjonen. I basic binding delen av dokumentasjonen er det mye informasjon på en gang med nye "kodespråk" og syntax. For å forstå grunnlegende databinding var man også nødt til å forstå Data Binding. Det var ikke et stort problem, men en måte å gjøre det lettere for en bruker på er å tilby linker til XAML siden sin fundementals, isteden for at man må ha flere vinduer oppe samtid og evt måtte hoppe frem og tilbake på sidene. De har dette for Label og Slider men ikke det som ligger inne i disse taggene.

Selve dokumentasjonen gir god informasjon og eksempler, men de kan fort bli komplekse for en nybegynner å sette seg inn i, derfor kan det også være smart å ha trinnvise eksempler som viser brukeren en ting om gangen. Som for eksempel hvordan de forskjellige "way" fungerer. Hva er forskjellen på "oneway" og "twoway" data binding. Det står skrevet hva som er forskjellen, men det kan være en god idé å vise brukeren bruksområder på de forskjellige veiene og hvordan hver enkelt fungerer. Dokumentasjonen gir også beskrivelser på trinnene om hvordan man implementerer dataen, men i trinnene viser den ikke fra start til slutt av implementasjonen. Dette kan forbedres ved bruk av flere bilder, men forklaringer på trinnvis binding av enkelte verdier til for eksempel flere verdier.

Når vi skulle jobbe med databinding fant vi også ut at vi var nødt til å implementere MVVM, Dette var den letteste måten å få ett skille mellom koden og dataen som bruker skulle sende inn. I denne delen av dokumentasjonen var det klart hva som måtte gjøres trinnvis for å kunne lage en GUI-applikasjon, hvor de også hadde lagt med bilder for å gjøre det lettere for brukere å forstå.

I all hovedsak må vi si at dokumentasjonen Microsoft har for MAUI gui'et er veldig godt skrevet. Men som sagt er det noen få endringer som vi selv skulle ønske blir lagt til.

1. Enkle trinnvise eksempler
2. Legge til flere linker til de forskjellige delene av XAML når man er inne på databinding siden.
3. Vise brukeren forskjellen på f.eks "oneway" og "twoway" data binding, samt vise bruksområder på de forskjellige veiene.

4. Bruk av flere bilder og kortere eksempler som tar seg av en ting om gangen for mindre kode og xaml tekst å lese over for å forstå hva som foregår i koden.

Med Disse endringene tror vi selv at det hadde vært lettere for oss å forstå bruk av data binding og MVVM.

6. Refleksjon

Ny for mappeinnleveringen.

I starten av semesteret når vi fikk oppgaven var vi usikre på hvordan det ville gå med å jobbe som en gruppe på et slikt prosjekt. Vi hadde også tenkt at denne oppgaven ville ha størrelsen til tidligere oppgaver vi har hatt i studieløpet vårt, noe som viste seg å være feil. Oppgaven vi fikk å jobbe med startet som ett ganske enkelt kodeprosjekt som senere ble bygget ut til et prosjekt med relativt stort omfang. Vi har fått prøvd og feilet på hvordan vi skal få rammeverket til å fungere slik som vi ønsket. Vi har i løpet av semesteret fått det vi selv ser på som godt læringsutbytte og kunnskapen vi har fått her blir høyt sannsynlig relevant senere i karrieren vår.

Rammeverket vårt har fra tidlige stadier vært ett ganske generelt rammeverk. Dette har hjulpet oss med å takle senere oppgaver uten store endringer i rammeverket. En erfaring vi gjorde tidlig var at hvis vi holdt rammeværket så generelt som mulig i hver gang vi implementerte nye features jo mindre tilpassing trengte vi å gjøre hver gang det kom nye krav i nye delinleveringer. Eksempler på dette var at vi i så stor grad som mulig ungikk å hardcode inn verdier i rammeværket vårt. Hvis det var snakk om at vi måtte få inn ett antall av noe, en prosentverdi av noe eller en størelse av noe så prøvde vi altid å gjøre det slik at brukeren av rammeværket sendte disse verdiene inn selv og at rammeværket bare brukte de verdiene som blir sendt inn. På denne måten spillte det ikke så mye rolle de fleste tilfellene når det kom nye krav fra delinleveringene om hvordan havnen skulle se ut ettersom det ofte bare var å endre på værdiene brukeren sendte inn til rammeværket og ikke den underliggende logikken til hvordan rammeværket var konstruert. Ett godt eksempel på dette var i den obligatorisk oppgave

3 hvor vi fikk utdelt en oppgave hvor en kunde ville ta i bruk vårt rammeverk. Her fikk vi utbetaling for hvor generelt vi hadde skrevet rammeværket vårt og mange av kravene som oppgaven spurte etter handlet bare for vår del om verdiene som sendtes inn gjennom de ulike konstruktørene. Utover det handlet det mest om at vi kun trengte å legge til de nye elementene. Dette var en nyttig erfaring som vi vil ta med videre i fremtidige kodeprosjekter.

Når vi jobbet med simuleringen la vi merke til at det ble ganske mange kompliserte IF tester og liknende, vi diskuterte innad i gruppen om en løsning på hvordan vi kunne få koden mer oversiktlig og lesbar. Vi endte opp med å gjøre om store deler av metodene som var gjort inne i klassen. Endringen vi gjorde var at vi flyttet ut if tester og noen deler av koden ut i nye metoder slik at en metode kun har ett ansvarsområde. Dette gjorde det lettere for oss som har jobbet med det å lese det, samt vil det forhåpentligvis også gjøre det lettere for brukeren å forstå hva som skjer innad i metodene. Dette var noe vi tenkte på litt sent i utviklingen og det tok derfor en god del tid når det skulle skilles ut. Vi diskuterte også å gjøre det samme i de andre klassene, men kom frem til at det var noe vi kunne se på senere i utviklingen. Etterhvert som vi jobbet videre la vi merke til at dette var noe vi ikke fikk tid til.

Hvordan vi ville jobbet videre med rammeverket

Hvis vi skulle jobbe videre med dette rammeverket er det flere veier å gå. En av veiene er for eksempel at en bruker skal kunne hente ut en spesifikk kontainer med ID, hvis denne kontaineren ligger på lagringsplassen, men har en kontainer som er lagret over denne så er man nødt til å flytte på kontainere for å få ut den riktige kontaineren. Vi har en versjon av ContainerStorageRow som vi begynte å kode hvor vi bruker en løsning med stacks for å simulere det at kontainere i StorageRowen legges oppå værandre og bare den øverste raden av kontainere til en hver tid kan hentes ut. Vi rakk aldri å bli helt ferdig å bugfixe denne implementasjonen som er grunden til at vi aldri endte opp med å bruke den i den endelige versjonen av rammeverket. Men hvis vi hadde hatt yterligere tid er

det noe vi ville jobbet for å implementere. Det er alltid måter å få ett rammeverk til å bli mer likt hvordan det fungerer i virkeligheten og det er nok dette vi ville jobbet oss mot.

En annen feature vi ikke hadde tid til å gjøre, men som man også kan se på som en vei videre i rammeverket er å implementere ett fullt grafisk grensesnitt. Vi implementerte ett grafisk grensesnitt i delinnlevering 4 som benyttet en tidligere versjon av rammeverket, men hvis vi hadde hatt mer tid vi hadde hadde kunnet jobbe videre med dette grafiske grensensnittet hadde vi jobbet for å bedre visualisere havnen. Dette ville forhåpentligvis gjøre det enklere for brukere å visualisere havnen og se hvordan de forskjellige elementene beveger seg rundt på havnområdet.

Når det kommer til exceptions i rammeverket vårt er der forbedringer vi ville gjort hvis vi hadde hatt mer tid til å jobbe med det. Slik rammeverket står i dag er det litt for inkonsekvent om når og hvor den kaster exceptions. Ett eksempel er at i enkelte metoder gjøres det sjekker for å se om trucks, containere eller ships man sender inn finnes i simuleringen, hvis disse ikke finnes kastes det en exception for å fortelle brukeren at det de leter etter ikke finnes i simuleringen. Det er ikke alle steder hvor det er relevant at en slik exception blir kastet hvor den faktisk blir det i vårt rammeverk. Hvis vi kunne jobbet videre med rammeverket ville vi forbedret metodene slik at de kastet flere slike relevante exceptions.

Vi ville også videreutvikle skips klassen til ikke lenger bare å ta en bool verdi for hvor vidt skipet skal være ett singletrip skip eller ikke, men heller implementert en løsning der brukeren selv kunne velge ett antall seilaser skipet skulle utføre. Dette ville gitt brukeren mer kontroll over skipene i simuleringen.

Til slutt ville vi jobbet ytterligere med å forbedre brukervennligheten til APIet. Gjort flere brukertester hvor brukere brukte rammeverket uten stegvise oppgaver for å ytterligere teste brukervennligheten til rammeverket. Ytterligere forbedringer og testing av XML kode hadde også vært bra.

Det er alltid flere forbedringer man kan gjøre på ett rammeverk som dette. Bare det å videreutvikle realismen og featurene til rammeverket er en jobb som man aldri blir helt ferdig med. Det er mange forbedringer som kan gjøres i rammeverket for å videreutvikle det i nyere versjoner, men endringene nevnt ovenfor er de vi ville gjort i de neste iterasjonene.

Hvordan har gruppearbeidet gått:

(Michael): Dette er en gruppe jeg har jobbet tidligere med i andre skoleprosjekter. Vi har iløpet av dette prosjektet og de tidligere blitt bedre på å se hvilke styrker og svakheter vi har og har klart på en god måte å jobbe ut fra kunnskapen vi har om hverandre. Jeg synes at gruppearbeidet har vært strålende uten problemer. Vi har vært flinke til å holde ukentlige møter og ha gode diskusjoner rundt problemene som har oppstått og funnet gode måter å løse de på. Ikke bare med implementering av kode men også i planleggingsfasen, Alle har hatt mulighet til å komme til med sine meninger og tanker rundt måten vi skal takle en oppgave på.

(Tuva): Det har gått fint å jobbe sammen som gruppe. Vi har jevnlig holdt ukentlige gruppemøter, både fysisk og digitalt, som har gjort det enkelt å holde seg oppdatert og legge videre plan for arbeid sammen.

(Andreas): Jeg synes gruppearbeidet har fungert fint. Alle har bidrat til prosjektet og vi har holdt god fremgang gjennom prosjektarbeidet. Vi har holdt ukentrlige fremgangsmøter om prosjektet og det har ikke vært noe problem med at gruppemedlemmer ikke møter opp eller ikke gjør de arbeidsoppgavene de har fått tildelt.

(Mathilde): Jeg syns gruppearbeidet har gått fint. Vi har jobbet som gruppe sammen før, så vi har litt erfaring med hvordan vi fungerer best som gruppe. Vi har likevel lært mer om hvor vi kan jobbe hverandre opp og fram. Alle har vært flinke til å møte på møter, kommunisere når noe ikke passer eller om man står noe fast.

Læringsutbytte

Igjennom dette prosjektet har vi fått mye ny læring. Spesielt har vi fått god læring innenfor samarbeid, tekniske ferdigheter og styring av et prosjekt. Erfaringen vi har fått av å utvikle et rammeverk som er fleksibelt og generelt er noe vi alle kommer til å ta med oss videre. Prosjektet har også gitt oss muligheten til å se hvordan man kan takle reelle utfordringer og deretter implementere løsninger som kan takle utfordringene på en realistisk måte.

I starten av semesteret var den ingen av oss som hadde erfaring med C#, men gjennom prosjektarbeidet har vi fått god erfaring i programmeringspråket, .NET rammeverket og hvordan det brukes. Enkelte på gruppen har også startet å programmere i C# på egene prosjekter etter at vi begynte med dette kurset.

Det har vært lærerikt å utvikle sitt eget rammeverk og vi føler vi har fått god innsikt i hvordan vi planlegger og gjennomfører større kodeprosjekter på egenhånd. Det har også gitt oss god innsikt i hvordan brukertesting fungerer og hva vi må tenke på før vi utfører tester. Det å se hvordan brukere reagerer på oppgavene vi gir dem, og hvordan de bruker ||rammeværket vårt og hvis de står fast, hvordan de går frem for å prøve å finne ut av problemet har vært veldig interesant. Måter som vi har satt ting opp i rammeværket vårt som vi trodde skulle være opplagt har til tider vært vanskelig for brukere å forstå, mens andre ting vi trodde skulle være vanskelig har de skjønt umiddelbart. For eksempel distinksjonen mellom en shipdock og en loadingdock som vi selv tenkte kunne skape forvirring har alle brukerne vi testet mot ikke hatt noe problem med å forstå, mens hvilken klasse de skulle gå inn i for å hente ut spesefikk informasjon om en simulering var det flere som brukte tid på. Dette var grunden til at vi endte opp med å legge inn måter å hente ut informasjon fra simuleringer i andre klasser en SimpleSimulation klassen i tilleg til den sistnevnte klassen.

Vi synes også det har vært en god erfaring å kunne jobbe på en lignende måte som det oppdragstagere i arbeidslivet gjør med oppdrag fra en kunde som må vi må løse. Det har vært en fin utfordring i hvordan man tar en prosjektbeskrivelse fra kunden og utvikler det til endelig kode som kan leveres på andre siden av arbeidsprosessen.

Vi er stort sett fornøyd med arbeidsprosessen og synes vi har lært mye gjennom å jobbe med dette prosjektet og føler oss bedre rustet nå til å takle utfordringer som vi kommer til å møte på ute i arbeidslivet.

7 Litteratur

Vedlegg 1: V1 API-dokumentasjon - av deres bibliotek

(kan være egen pdf fil). Skal være oppdatert for endelig versjon.

(Se separat dokument)

Vedlegg 2: Diskusjon om regler fra boken

(kan være egen pdf fil).

Som beskrevet i “Delinnlevering 4 - 3 Formelle krav til arbeidet”. Skal være oppdatert for den endelige versjonen.

Kapitel 2: Framework Design Fundamentals

Helt fra første delinnlevering har det vært viktig for oss at APIet vi designet for vårt rammeverk er så intuitivt og selvdokumenterende som mulig.

- ‘’DO design frameworks that are both powerful and easy to use.’’
  (Cwalina; Jeremy; Brad. et al., 2020 s. 9)

- ''DO understand and explicitly design for a broad range of developers with different programming styles, requirements, and skill levels.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 10)
- ''DO make sure that the API design specification is the central part of the design of any feature that includes a publicly accessible API.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 17)
- ''DO define top usage scenarios for each major feature area. The API specification should include a section that describes the main scenarios and shows code samples implementing these scenarios.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 17)
- ''DO ensure that each main feature area namespace contains only types that are used in the most common scenarios.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 25)
- "DO provide simple overloads of constructors and methods. A simple overload has a very small number of parameters, and all parameters are primitives"
  (Cwalina; Jeremy; Brad. et al., 2020 s. 26).
- ''DO communicate incorrect usage of APIs using exceptions.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 28)
- ''DO ensure that APIs are intuitive and can be successfully used in basic scenarios without referring to the reference documentation.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 30)
- ''DO provide great documentation with all APIs.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 30)
- ''DO provide code samples illustrating how to use your most important APIs in common scenarios.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 30)

Det har vært ett tydelig mål fra vår side å designe APIet på en slik måte at brukeren ikke skal trenge å lese ekstern dokumentasjon for å kunne forstå hvordan APIet skal brukes.

Vi har derfor lagt stor vekt på navngivning i APIet vårt.

- ''DO make the discussion about identifier naming choices a significant part of specification reviews. What are the types most scenarios start with? What are the names most people will think of first when trying to implement this scenario? Are the names of the common types what users will think of first? For example, because "File" is the name most people think of when dealing with file I/O scenarios, the main type for accessing files should be named File.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 31)
- ''DO provide strongly typed APIs if at all possible.''
  (Cwalina; Jeremy; Brad. et al., 2020 s. 33)

- ‘’DO use exception messages to communicate framework usage mistakes to the developer.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 33)
- ‘’DO ensure consistency with .NET and other frameworks your users are likely to interact with.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 34)

- ‘’CONSIDER reserving the best type names for the most commonly used types.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 32)

- ‘’AVOID many abstractions in mainline-scenario APIs.’’
  (Cwalina; Jeremy; Brad. et al., 2020 s. 35)

Det er viktig at navnene som blir brukt i klasser og parametere er intuitive og beskrivende for hva de symboliserer. I brukertestene brukte la vi vekt på å spørre brukerne om hva de trodde var ment med de forskjellige klassenavnene og parameternavnene i APIet for å prøve å finne ut om navnene på best mulig måte kommuniserte det vi ønsket at de skulle.

- ‘’DO NOT be afraid to use verbose identifier names when doing so makes the API self-documenting.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 31)

Kapitel 3: Navngivningskonventioner

Mer forståelige navn er bedre en kortere og mer obskure navn. Og det er en tommelfingerregel som vi har hatt med oss hele veien gjennom utviklingen av APIet.

I tillegg har vi passer på å følge god praksis når det kommer til standard navngivningskonvensjoner i programmering. Dette gjelder alle reglene nevnt nedunder fra side 43-52.

- “DO use PascalCasing for the names of namespaces, types, members, and generic type parameters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 43)
- “DO use camelCasing for parameter names.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 43).
- “DO capitalize both characters of two-character acronyms, except the first word of a camel-cased identifier.” (Cwalina; Jeremy; Brad. et al., 2020 s. 45)

- “DO capitalize only the first character of acronyms with three or more characters, except the first word of a camel-cased identifier.” (Cwalina; Jeremy; Brad. et al., 2020 s. 46).
- “DO choose easily readable identifier names.
  For example, a property named HorizontalAlignment is more English readable than AlignmentHorisontal.” (Cwalina; Jeremy; Brad. et al., 2020 s. 52)

Enkelte paremeternavn i APIet vårt kan til tider være veldig verbose, men gjennom brukertesting fant vi ut at disse navnene ofte var bedre for at APIet skulle bli enkelt å forstå for brukeren. Læreboken understreker også dette synspunktet flere ganger.

- "DO favor readability over brevity." (Cwalina; Jeremy; Brad. et al., 2020 s. 52).

Dette er vanlige konvensjoner i programmering og derfor viktig å følge ettersom koden da blir mer forståelig for brukeren. Det å ikke følge disse konvensjonene kan også skape problemer hvis brukeren prøver å benytte rammeverket på tvers av programmeringsspråk. For eksempel er ikke alle programmeringsspråk case sensitive og navngivning som skiller seg kun på store og små bokstaver kan fort skape problemer i disse senarioene.

Rammeverket vårt har bare ett namespace, Gruppe8.HarbNet.
Navngivningen her følger konvensjonene gitt i læreboka.

- “DO prefix namespace names with a company name to prevent namespaces from different companies from having the same name. For example, the Microsoft Office automation APIs provided by Microsoft should be in the namespace Microsoft.Office.” (Cwalina; Jeremy; Brad. et al., 2020 s. 63).
- ‘’DO use a stable, version-independent product name at the second level of a namespace name.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 63)
- ‘’DO use PascalCasing, and separate namespace components with periods (e.g., Microsoft.Office.PowerPoint).’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 64)

Overlappende namespace navngivning gjør at det fort kan bli krasj når forskjellige rammeverk skal integreres med hverandre. For eksempel hvis brukeren ønsker å bruke features fra .NET rammeverket sammen med HarbNet rammeverket.

Hvis namespacene blir for like kan det føre til at brukeren kaller på feil klasser, metoder eller variabler som gir uønskede resultater.

Grunnen til at vi har valgt å bare ha ett namespace så langt i utviklingen av vårt API er fordi rammeverket vårt fremdeles har ett veldig begrenset antall klasser. Det oppstår derfor ikke noen konflikt i vårt rammeverk der forskjellige deler av rammeverket har lignende navn eller lignende funksjoner som kan være forvirrende enten programmatisk eller for brukeren som skal benytte rammeverket. Derfor for fremdeles å holde på prinsippet om at APIet og rammeverket vårt skal være så enkelt og selvdokumenterende som mulig anser vi det som best og holde antall namespaces så lavt som mulig. Det kan hende at det må opprettes flere namespaces hvis rammeverket må utvides til å inneholde flere klasser, men så langt har ikke dette vært nødvendig.

Når det kommer til navngivning av klasser ber boken om at det brukes substantiv i navngivningen.

- ‘’DO name classes and structs with nouns or noun phrases, using PascalCasing.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 68)

APIet vårt er definert i en rekke interfacer som lager kontraktene for de public delene av rammeverket vårt.

Det har vært mye diskusjon frem og tilbake i gruppen om hvor hvidt disse kontraktene burde være definert som interfacer eller abstrakte klasser. Vi er stort sett enige om at abstrakte klasser er det mest riktige for rammeverket vårt. Boken sier:

- “DO name interfaces with adjective phrases, or occasionally with nouns or noun phrases.” (Cwalina; Jeremy; Brad. et al., 2020 s. 68)

Alle navn på klassene våre er substantiv. Dette gir mest mening ettersom klasser som oftest representerer oppskrifter på objekter og ting i det virkelige live. For eksempel klassen Ship som beskriver oppskriften på ett skip. Vi følger reglene nevnt på side 69 og 70 nevnt nedenunder når det kommer til navngivning av interfaces.

- ‘’DO prefix interface names with the letter I, to indicate that the type is an interface.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 69)

- ‘’DO ensure that the names differ only by the “I” prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 70)
- ‘’DO use a singular type name for an enumeration unless its values are bit fields.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 72)

Våre interfacer deler samme navnet som klassene de definerer kontrakten for og har strukturen “IKlasseNavn”. Dette gjør det enklere for brukeren å vite hvilke klasser interfacet definerer kontrakten for som igjen hjelper til å gjøre APIet selvdokumenterende.

- ‘’DO name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek).’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 75)
- “DO name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has,” but only where it adds value.” (Cwalina; Jeremy; Brad. et al., 2020 s. 76).

Når vi designet Eventene, var det viktig å navngive de basert på handlingen/aktiviteten som ble gjort, i tillegg til om det er i nåtid eller fortid. Dette klargjør for brukeren hva eventet faktisk er «raised» av, og gir en indikasjon på hva som er neste skritt for objektet det er snakk om.

```
ShipAnchoring;
ShipAnchored;
```

Dette følger og referer de første følgende to nevnte reglene nedenfor til i boka på side 77;

- “DO name events with a verb or a verb phrase. Examples include Clicked, Painting, and DroppedDown.” (Cwalina; Jeremy; Brad. et al., 2020 s. 77)
- “DO give events names with a concept of before and after, using the present and past tenses, such as Closing and Closed. For example, a close event that is raised before a window is closed would be called Closing, and one that is raised after the window is closed would be called Closed.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 77)

- "DO name events with a verb or a verb phrase. Examples include Clicked, Painting, and DroppedDown." (Cwalina; Jeremy; Brad. et al., 2020 s. 77.)

Boka refererer også til regler relatert til opprettelse av egne EventHandlers, som to parametere (sender, e) og riktig suffix. Siden vi ikke har hatt behovet for å lage våre egne EventHandlers, og heller følger «DO use System.EventHandler<T> (...)» helt ut, faller ikke disse reglene eller DO's inn i vårt API. Vi har likevel **vurdert** behovet for både egne EventHandlers (delegates) og callbacks. Dette gjelder begge reglene nedenfor fra side 77.

- "DO name event handlers (delegates used as types of events) with the "EventHandler" suffix, such as ClickedEventHandler."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 77.).
- "DO use two parameters named sender and e in event handlers."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 77)

Når det gjelder navngiving av subklassene, bruker vi «EventArgs» suffix, for å gjøre det klart og tydelig at dette er en Args-klasse, som tilhører et Event. Vi har også konsekvent navngitt disse klassene etter Eventene som tar de i bruk, slik at det er lett for bruker å finne tilhørende Args.

```
public event EventHandler<ShipLoadedContainerEventArgs>? ShipLoadedContainer;
```

```
public class ShipLoadedContainerEventArgs : BaseShipEventArgs
```

- "DO name event argument classes with the "EventArgs" suffix"
  (Cwalina; Jeremy; Brad. et al., 2020 s. 78).
- "DO use camelCasing in parameter names."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 79)
- "DO use descriptive parameter names. Parameter names should be descriptive enough to use with their types to determine their meaning in most scenarios."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 79)

Vi bruker ikke mange booleans i vårt API, men det var relevant for om skip skulle være single trip eller ikke.

```
internal bool IsForASingleTrip { get; set; } = false;
```

I tillegg til atributten som setter verdien i konstruktøren

```
bool isForASingleTrip,
```

Vi passet her på at disse hadde navn som reflekterte True verdien av booleanen slik at det følger standarden satt i boka.

Dette er relevant for de propertisene som bruker enum verdier i rammeverket vårt. Mest relevant for Ship og Container klassen som har properties som bruker enumene ShipSize og ContainerSize.

```
public ContainerSize ContainerSize { get; internal set; }
```

```
public ShipSize ShipSize { get; internal set; }
```

Her har vi hatt litt diskusjon om denne regelen er verdt å følge. På den ene siden kan man argumentere for at det blir dobbelt opp med informasjon. Hvis jeg kaller Container objektet Container.Size så er dette navnet antageligvis like tydelig som å kalle på container.ContainerSize, og bruker vil mest sannsynlig skjønne ut ifra navngivningen alene at det er kontainerens størrelse de henter ut i begge tilfellene.

Vi valgte likevel å bruke det mer verbose navnet ContainerSize ettersom ved å følge standard konvensjonen nevnt i boka vil navnet kanskje være litt mer intuitivt å søke opp i dokumentasjon eller i listen over propertisene til klassen i IDE hvis det følger de konvensjonene som er vanlig i bransjen.

- “CONSIDER giving a property the same name as its type.” (Cwalina; Jeremy; Brad. et al., 2020 s. 76).
- “CONSIDER using named based on parameter meaning rather than parameter type.” (Cwalina; Jeremy; Brad. et al., 2020 s. 80).
- “AVOID using identifiers that conflict with keywords of widely used programming languages.” (Cwalina; Jeremy; Brad. et al., 2020 s. 53)

- “DO NOT capitalize any of the characters of any acronyms, whatever their length, at the beginning of a camel-cased identifier.” (Cwalina; Jeremy; Brad. et al., 2020 s. 46)
- “DO NOT assume that all programming languages are case sensitive. They are not. Names cannot differ by case alone.” (Cwalina; Jeremy; Brad. et al., 2020 s. 52)
- “DO NOT use underscores, hyphens, or any other non-alphanumeric characters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 52)
- “DO NOT use abbreviations or contractions as part of identifier names.” (Cwalina; Jeremy; Brad. et al., 2020 s. 55)
- “DO NOT use any acronyms that are not widely accepted, and even if they are, only when necessary.” (Cwalina; Jeremy; Brad. et al., 2020 s. 55).

Navngivningen følger konvensjonen CompanyName.SolutionName, der Gruppe8 er vårt firmanavn og HarbNet er navnet på selve rammeverket.
Namespacet bruker PascalCasing (Cwalina; Jeremy; Brad. et al., 2020 s. 64) og vi passer på at namespacet ikke ligner på noen av namespacene i det offisielle .NET rammeverket.

- “DO NOT give types names that would conflict with any type in the core namespaces. For example, never use Stream as a type name. It would conflict with System.IO.Stream, a very commonly used type.” (Cwalina; Jeremy; Brad. et al., 2020 s. 66)
- “DO NOT use an “Enum” suffix in enum type names.” (Cwalina; Jeremy; Brad. et al., 2020 s. 72)
- “DO NOT use “Before” or “After” prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.” (Cwalina; Jeremy; Brad. et al., 2020 s. 77).

Kapitel 4: Type design guidelines

I denne endelige versjonen av rammeverket bestemte vi oss for å gjøre om interfacene som definerte kontraktene for APIet om til abstrakte klasser. Det har vert mye diskusjon i gruppen gjennom hele utviklingsprosessen om vi skulle gjøre denne endringen. Vi har alle vært enige om at abstrakte klasser ville være det mest riktige i forhold til det læreboken sier, men vi har i de tidligere iterasjonene av rammeverket fremdeles valgt å bruke Interface ettersom de tidligere del innleveringene spesifikt sa at vi skulle definere APIet gjennom bruk av Interface. Grunnen til at vi bestemte oss for å endre Interface implementasjonen til istedenfor å bruke abstrakte klasser i denne endelige innleveringen er fordi oppgaveteksten for denne mappeleveringen ikke spesifiserer at Interfaces må brukes og vi derfor fremdeles anser rammeverket til å være i tråd med det oppgaven ber om selv om det ikke brukes Interface.

Grunnen til å ha abstraksjoner til å definere APIet er todelt. For det første tvinger det klasser som skal være en del av APIet til å inkludere public metodene som APIet krever at de skal ha. Sånn sett definerer abstraksjonene en kontrakt for hva brukeren skal kunne forvente av klasser som er en del av APIet. Den andre årsaken er at hvis det skal lages tester for rammeverket vil abstraksjonene være ett godt utgangspunkt for å utvikle Fakes for bruk i testingen, siden vi altid kan regne med at APIet bruker metodene definert i abstraksjonene.

Interface setter begrensninger på videre utvikling av API ettersom klasser som arver fra dem tvinges til å bruke alle eksisterende members i interfacet de arver fra.
Dette gjør interface basert design lite fleksibelt ettersom enhver endring i Interface risikerer å ødelegge eksisterende implementasjoner av rammeverket. Abstrakte klasser derimot tilbyr de samme mulighetene til å definere kontrakter for APIet, men gir yterligere fleksibilitet ved at disse kontraktene kan endres i ettertid. Om metodeimplementasjon må endres ved ett senere tidspunkt eller ikke lenger trenger å være en del av kontrakten kan innholdet i den abstrakte klassen endres for å etterkomme de nye behovene til APIet uten at dette ødelegger eksisterende implementasjoner av klassen.
Nye metoder kan legges til, eller eksisterende metoder kan endres fra abstrakt til ikke-abstrakt for å ikke lenger gjøre at klasser som arver fra den abstrakte blir tvunget til å

implementere dem. Metodene i den abstrakte klassen kan også ha sine egne konkrete implementasjoner slik at klassene som arver fra den abstrakte ikke lenger trenger å implementere sin egen versjon av metoden. Dette gjør abstrakte klasser mye mer fleksibelt en interface og gjør rammeverket mye mer robust og åpen for videre utvikling i fremtiden.

Når det kommer til selve årsakene til hvorfor vi anser det som mer riktig å bruke abstrakte klasser istedenfor Interfaces er det flere grunder til dette. DO reglene nedenunder fra side 97 og 98 sier dette:

- "DO favor defining classes over interfaces. Class-based APIs can be evolved with much greater ease than interface-based APIs because it is possible to add members to a class without breaking existing code."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 97).
- DO use abstract classes instead of interfaces to decouple the contract from implementations." (Cwalina; Jeremy; Brad. et al., 2020 s. 98).

Når det kommer til konstruktørene for disse abstrakte referer vi til DO regelen nedenunder fra side 100.

Vi har valgt å sette konstruktøren for disse klassene til internal. Det er liten grunn til at brukere av APIet skal trenge å lage objekter av disse abstraksjonene ettersom de ikke inneholder nok informasjon til å kunne brukes i en simulering av en havn. Å la brukere kunne opprette objekter av disse klassene kan fort føre til forvirring og gjøre det utydelig hvordan rammeverket er ment å brukes. Det er fint at klassene er tilgjengelig for brukeren å kunne studere for å se hva slags funksjonalitet APIet tilbyr, men det å lage objekter av disse abstraksjonene anser vi som unødvendig. Ett eksempel på hvordan konstruktøren til disse klassene er definert kan vi se i klassen CargoVessel som har følgende konstruktør:

```
internal CargoVessel()
{
}
```

- "DO define a protected or an internal constructor in abstract classes. A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 100).

Alle de abstrakte klassene som er definert har en konkret klasse som arver fra dem. Dette hjelper til å forsvare hvorfor abstraksjonen eksisterer. Dette følger også lærebokens råd om design av abstrakte klasser.

Under er en liste av de konkrete klassene og de abstrakte klassene de arver fra.

```
public class Container : StorageUnit
public class ContainerStorageRow : StorageArea
public class DailyLog : HistoryRecord
public class Harbor : Port
public class Ship : CargoVessel
public class SimpleSimulation : Simulation
internal class ShipDock : Dock
internal class LoadingDock : Dock
public class StatusLog : StatusRecord
```

Vi har brukt enumer til å definere størrelsene på containere og skip. Her synes vi enumer gir mest mening for det første fordi størrelsene her har konsekvenser for andre deler av objektene de hører til. For eksempel vil antallet containere et skip har plass til i lasterommet være avhengig av størrelsen til skipet. For det andre hjelper enumene til å gi brukeren ett sett antall størrelser å velge mellom. Brukeren har derfor aldri mulighet til å velge størrelser det ikke har vært tatt høyde for i resten av rammeverket.

- “DO use an enum to strongly type parameters, properties, and return values that represent sets of values.” (Cwalina; Jeremy; Brad. et al., 2020 s. 112)

None er en litt spesiell verdi ettersom den symboliserer ett skip eller kontainer uten størelse, noe som i virkeligheten ikke vil være mulig. Men boken ønsker at enumer skal inneholde en 0 verdi så vi har valgt å følge denne konvensjonen.

- “DO provide a value of zero on simple enums. Consider calling the value something like “None.” (Cwalina; Jeremy; Brad. et al., 2020 s. 116)

Denne regelen passer bra i vårt rammeverk ettersom størrelsen på skip og containere også bestemmer vekten til skipet eller kontaineren. Det passer derfor fint å kunne gi

enumverdiene i disse enumene tallverdier som representerer vekten på objektet av den gitte størrelsen. Vi kan se ett eksempel på dette i ContainerSize klassen:

```
public enum ContainerSize
{
    /// <summary>
    /// No size.
    /// </summary>
    None = 0,
    /// <summary>
    /// Half container. Weight: 10 tonn.
    /// </summary>
    Half = 10,
    /// <summary>
    /// Full container. Weight: 20 tonn.
    /// </summary>
    Full = 20,
```

Der half størrelse kontainere har en vekt på 10 tonn og full size containere har en vekt på 20 tonn. Dette gjør det også enklere for brukeren og hente ut disse tallverdiene eller gjøre sammenligninger på vekt hvis brukeren ønsker det.

- “CONSIDER adding values to enums, despite a small compatibility risk.” (Cwalina; Jeremy; Brad. et al., 2020 s. 123).
- “DO NOT define public or protected internal constructors in abstract types. Constructors should be public only if users will need to create instances of the type.” (Cwalina; Jeremy; Brad. et al., 2020 s. 100)

Det er slik vi også har tenkt når det kommer til enumene i rammeverket vårt. Det at størrelsene på kontainere og skip blir strongly typed gjør det enklere for brukeren å vite hva det er de skal forholde seg til av verdier og gjør det forutsigbart hva det er de får tilbake i metodekall. Med untak av verdien None har vi heller ikke enumverdier i enumene som ikke brukes av resten av rammeverket, noe boken også understreker.

- “DO NOT provide reserved enum values that are intended for future use.” (Cwalina; Jeremy; Brad. et al., 2020 s. 113)

Kapitel 5: Member design

- “DO try to use descriptive parameter names to indicate the default used by shorter overloads.” (Cwalina; Jeremy; Brad. et al., 2020 s. 137).
- “DO make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 138)

Når det kommer til gettere settere har nesten alle propertiene våre som tilhører det public APIet tilhørende gettere. Vi tenker gettere er en standard og naturlig måte for brukere å hente ut informasjon fra objektene som de oppretter.

- “DO create get-only properties if the caller should not be able to change the value og the property. If the type off the property is mutable refrence type, thhe property value can be changed even if the property is get-only”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 158).
- “DO explicitly declare the public default constructor in classes, if such a constructor is required.” (Cwalina; Jeremy; Brad. et al., 2020 s. 169)

I dokumentasjonen og XML refererer vi til når eventet skjer og trigges med «raise», for å følge riktig konvensjon.

- “DO use the term “raise” for events rather than “fire” or “trigger.” When referring to events in documentation, use the phrase “an event was raised” instead of “an event was fired” or “an event was triggered.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 176.).

Vi bruker også konsekvent System.EventHandler<T> (hvor T er Args som sendes med eventet), i stedet for å opprette våre egne delegates.

- “DO use System.EventHandler<T> instead of manually creating new delegates to be used as event handlers.” (Cwalina; Jeremy; Brad. et al., 2020 s. 177)
- “DO use enums if a member would otherwise have two or more Boolean parameters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 205)

Vi synes dette forslaget passer godt for vårt rammeverk i dette tilfelle der en mindre kompleks konstruktør kan gjøre det enklere for brukere å raskere opprette havner i de tilfellene hvor høy grad av nøyaktighet i hvordan havnen ser ut ikke er så viktig. Vi definerte derfor denne overloaden til konstruktøren i Harbor:

```
public Harbor(int numberOfShips, int containerStorageCapacityInEachStorageRow, int numberOfHarborContainerStorageRows, int numberOfLoadingDocks,
    int numberOfCranesNextToLoadingDocks, int numberOfCranesOnHarborStorageArea, int numberOfAgvs,
    int loadsPerCranePerHour = 35, int trucksArrivePerHour = 10, int loadsPerAgvPerHour = 25,
    int percentageOfContainersDirectlyLoadedFromShipToTrucks = 10, int percentageOfContainersDirectlyLoadedFromHarborStorageToTrucks = 15)
```

Denne konstruktøren bruker bare primitive verdier og i halvparten av parameterne har vi gitt default verdier som brukeren kan bruke hvis de ikke ønsker å definere disse verdiene selv. Verdiene gitt som default verdier er verdier vi ser går igjen i virkelige havner.

Vi håper dette bidrar til å gjøre Havn klassen vår enda mer brukervennlig, spesielt for brukere som ikke har nok domenekunskap til å kunne sette alle verdiene selv. Den forenkler også prosessen med å opprette en simulering ettersom brukeren her ikke trenger å opprette egne skip eller storage rows som skal brukes i simuleringen selv. De kan bare gi antallet så gjør konstruktøren jobben for dem. Hvis brukeren selv ønsker å opprette disse kan de fremdeles bruke den mer komplekse konstruktøren.

- “CONSIDER providing simple, ideally default, constructors. A simple constructor has a very small number of parameters, and all parameters are primitives or enums.” (Cwalina; Jeremy; Brad. et al., 2020 s. 165)
- “CONSIDER using a subclass of EventArgs as the event argument, unless you are absolutely sure the event will never need to carry any data to the event handling method, in which case you can use the EventArgs type directly.” (Cwalina; Jeremy; Brad. et al., 2020 s. 178.)

For event-håndtering vår bruker vi konsekvent bare egenlagde subklasser av EventArgs klassen. Dette for å forsikre fremtidig kompatibilitet, og fordi vi ville sende data(args) med eventene, noe EventArgs alene ikke gjør. Vi så også at det var større mengde felles argumenter som ville gå igjen i noen av EventArgs klassene, så vi lagde også en base-klasse for disse, nemlig BaseShipEventArgs. Dette unngår dobbelt opp med kode og gjør det lettere å vedlikeholde.

- “CONSIDER using a subclass of EventArgs as the event argument” (Cwalina; Jeremy; Brad. et al., 2020 s. 178)

Her er to overload konstruktører som har noen ulike parametere.

```
public Ship(string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip,
    int roundTripInDays,IList<Container> containersToBeStoredInCargo, int directDeliveryPercentage)
```

```
public Ship(string shipName, ShipSize shipSize, DateTime startDate, bool isForASingleTrip, int roundTripInDays,
    int numberOfHalfContainersOnBoard,
    int numberOfFullContainersOnBoard)
```

De har for det første beholdt navnet på parameteren som er like, men de har også etter beste evne blitt lagt på samme plass. Hva vi mener med dette er at shipName ligger først og shipSize ligger som nr.2 etc. På denne måten er mulig for brukeren å kunne forstå hvordan man kan lage objekter ut av de forskjellige konstruktørene, og brukeren vil ha mulighet til å vite eksakt hva som er forskjellen på de ulike overloadene ved å se på hva som er likt og ikke. Ifølge boken så er det på denne måten det skal løses på:

- “AVOID being inconsistent in the ordering of parameters in overloaded members. Parameters with the same name should appear in the same position in all overloads.” (Cwalina; Jeremy; Brad. et al., 2020 s. 138)
- “AVOID having two overloads of the same method that both use default parameters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 146)
- “AVOID throwing exceptions from property getters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 159)

Under utviklingen av rammeverket vårt var det viktig for oss at rammeverket skulle være lett å forstå for utenomstående personer som skal se og bruke rammeverket.

Vi har igjennom brukertesting prøvd å finne gode parameternavn som er lette å forstå for brukerene, og vi har kommet frem til at mange av parameternavnene våre er lette å forstå for en bruker som ikke har sett rammeverket før.

Hvis man ser på konstruktørene våre har vi holdt navngivningen på de parameterne like, altså hvis en parameter i en overload skal gjøre det samme som en annen parameter i en annen overload har de samme navn.

- “DO NOT arbitrarily vary parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 137)
- “DO NOT have overloads with parameters at the same position and similar types, yet with different semantics.” (Cwalina; Jeremy; Brad. et al., 2020 s. 139)

Vi har også passet på at vi ikke har public settere som har høyere grad av tilgjengelighet en det getteren til en property har. Det gir ikke mening å gi brukeren tilgang til å endre properties de hverken kan hente ut eller bruke og det å gi dem tilgang til å gjøre dette kan fort føre til forvirring og feil fra brukerens side.
I de aller fleste tilfeller er det bare getteren som er tilgjengelig for brukeren gjennom APIet ettersom de fleste av endringene som blir gjort på propertisene gjøres av selve simuleringen og det å gi tilgang til å endre disse propertisene ofte bare vil ende opp med å ødelegge informasjonen som genereres av simuleringen.

For eksempel en containers CurrentPosition property vil endre seg basert på hvordan simuleringen utspiller seg.

```
public Guid CurrentPosition { get; internal set; }
```

Her gir det mening for brukeren å kunne hente ut informasjonen av hvor en container befinner seg, men å sette denne verdien selv vil ødelegge informasjonen om kontainerens bevegelse som simuleringen genererer.

- ‘’DO NOT provide set-only properties or properties with the setter having broader accessibility than the getter.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 158)
- ‘’DO NOT use Boolean parameters unless you are absolutely sure there will never be a need for more than two values.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 206)

Kapitel 6: Designing for Extensibility

- ‘’DO choose carefully between an abstract class and an interface when designing an abstraction.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 240)

Våre event er designet til å bli «raised» når det skjer større aktivitet i simuleringen, og med det gi beskjed til bruker at “nå har noe spesifikt skjedd”. Det er for eksempel aktivitetene et skip gjør, som å ankre, laste av eller på og reise ut av havnen. På bakgrunn av og med dette i tanken, har vi ikke implementert «callbacks», som etter vår forståelse er en mer avansert teknikk, som tillater bruker å sende noe tilbake til rammeverket, som for eksempel en funksjon, som en respons til beskjeden de mottar. Det er det altså ikke bruk for i vårt design, da APIet kun har behov for “enveiskommunikasjon" fra rammeverk til bruker.

Dette følger og forsterkes av følgende CONSIDER regler i boka på side 231, om event design og valget mellom callbacks og events.

Boka refererer også til regler relatert til opprettelse av egne EventHandlers, som to parametere (sender, e) og riktig suffix. Siden vi ikke har hatt behovet for å lage våre egne EventHandlers, og heller følger «DO use System.EventHandler<T> (…)» helt ut, faller ikke disse reglene eller DO’s inn i vårt API. Vi har likevel **vurdert** behovet for både egne EventHandlers (delegates) og callbacks.

- ‘’CONSIDER using callbacks to allow users to provide custom code to be executed by the framework.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 231)
- ‘’CONSIDER using events instead of plain callbacks, because events are more familiar to a broader range of developers and are integrated with Visual Studio statement completion.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 231)
- ‘’CONSIDER making base classes abstract even if they don’t contain any abstract members. This clearly communicates to the users that the class is designed solely to be inherited from.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 242)
- “CONSIDER making base classes abstract even if they don’t contain any abstract members. This clearly communicates to the user that the class is designed solely to be inherited from.” (Cwalina; Jeremy; Brad. et al., 2020 s. 243).

I tillegg til navnendringene valgte vi også å legge disse klassene til et eget namespace kalt Advanced.

Selv om disse abstrakte klassene er public og tilgjengelig gjennom API er det ikke klasser som de aller fleste brukere av rammeverket vil ha behov for å direkte bruke selv i sine egene implementasjoner. Vi anså det derfor for mer brukervenlig å separere ut disse klassene i ett eget namespace for å skille dem fra resten av APIet. På den måten blir disse baseklassene mer tydelig separert fra de mer vanlig brukte klassene i APIet og skille mellom dem mer tydelig for brukeren. Vi håper dette gjør rammeverket mer forståelig for brukeren og gir en bedre indikasjon på hvilke klasser brukeren burde benytte i sin egen kode.

- "CONSIDER placing base classes in a separate namespace from the mainline scenario types. By definition, base classes are intended for advanced extensibility scenarios and are not interesting to the majority of users." (Cwalina; Jeremy; Brad. et al., 2020 s. 243)

- "AVOID naming base classes with a "Base" suffix if the class is intended for use in public APIs." (Cwalina; Jeremy; Brad. et al., 2020 s. 243)

Kapitel 7: Exceptions

- ''DO report execution failures by throwing exceptions.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 256.).
- "DO provide a rich and meaningful message text targeted at the developer when throwing an exception. The message should explain the cause of the exception and clearly describe what needs to be done to avoid the exception." (Cwalina; Jeremy; Brad. et al., 2020 S.262).
- ''Do provide (at least) these common constructors on all exceptions.'' (Cwalina; Jeremy; Brad. et al., 2020 S.262).
- ''DO create a new exception type to communicate a unique program error that cannot be communicated using an existing framework exception.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 263.)

- ''DO throw the most specific (the most derived) exception that makes sense.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 263.)

Det er viktig at exceptionene gir gode beskrivelser om hva det er brukeren gjør feil og hvordan de skal rette opp i feilen.
Målet vårt har vært å ha exception messages som er beskrivende nok til at brukeren ikke skal trenge å slå opp i ekstern dokumentasjon for å vite hvordan de skal håndtere eller rette opp feilen. Ett eksempel på en exeption message:

```
ArgumentOutOfRangeException("percentageOfContainersDirectlyLoadedFromShipToTrucks must be a number between 0 and 100.");
```

Her beskriver vi hvilken parameter brukeren har skrevet feil i, hva feilen er og hva brukeren må skrive for at det skal bli riktig.

- ''DO provide a rich and meaningful message text targeted at the developer when throwing an exception.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 264)
- ''DO ensure that exception messages are grammatically correct.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 264.)
- ''DO ensure that each sentence of the message text ends with a period.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 264.)

Når det kommer til exceptions i rammeverket vårt har vi valgt å holde oss til de standard exceptionene som defineres i .NET rammeverket så mye som mulig. Det er ikke mange steder brukeren har mulighet til å gjøre feil som kan kaste exeptions i rammeverket vårt.

Ett eksempel på et tilfelle der det er mulighet for brukeren å gjøre feil er i en av konstruktørene til Harbor klassen. Den har to parametere hvor brukeren skal skrive inn en prosentverdi:

```
int percentageOfContainersDirectlyLoadedFromShipToTrucks,
int percentageOfContainersDirectlyLoadedFromHarborStorageToTrucks,
```

I disse paramterene kan brukeren gjøre feilen å enten skrive inn en prosentverdi som er mindre en 0 eller større en 100. Disse verdiene vil ikke gi mening ettersom det ikke kan være ett negativt antall prosent av containere som lastes eller ett prosentantall som er høyere en 100. Hvis brukeren gjør dette vil konstruktøren kaste en exception.

Her har vi valgt å bruke den innebygde .NET exeptionen ArgumentOutOfRangeExeption istedefor å lage en egen exeptionklasse som er unik for vårt rammeverk. Vi følger da reglen nedenunder, i tillegg til DO NOT regelen nevnt på s. 262 fra boka, som er nevnt i DO NOT seksjonen nedenunder.

- ''DO throw ArgumentException or one of its subtypes if bad arguments are passed to a member. Prefer the most derived exception type, if applicable.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 275)
- ''DO derive exceptions from System.Exception or one of the other common base exceptions.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 279)

Exception klassene i seg selv følger veiledningen gitt i boka og alle exceptionene inneholder de 3 standard konstruktørene som boken etterspør.

```
public class CraneCantBeLoadedExeption : Exception
```

```
public class AgvCantBeLoadedExeption : Exception
```

```
public class TruckCantBeLoadedExeption : Exception
```

```
public class TruckCantBeLoadedExeption : Exception
{
    /// <summary> Creates new TruckCantBeLoadedException object.
    0 references
    public TruckCantBeLoadedExeption() { }


    /// <summary> Creates new TruckCantBeLoadedException object.
    3 references
    public TruckCantBeLoadedExeption(string message) : base(message) {

    }

    /// <summary> Creates new TruckCantBeLoadedException object.
    0 references
    public TruckCantBeLoadedExeption(String message, Exception innerException) : base(message, innerException)
    {
    }
}
```

- ''DO end exception class names with the "Exception" suffix.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 280)

Å bruke eksisterende og kjente exception klasser gjør det lettere for brukeren å forutsi og håndtere mulige exceptions som kan dukke opp når de bruker APIet. Det å lage nye og unike exceptions for alle tilfeller der brukeren kan gjøre feil vil fort gjøre rammeverket mye større og mer uoversiktlig en det trenger å være.

Det er steder i rammeverket hvor feil kan oppstå i interne metoder.
Ett slikt eksempel er CraneToTruck metoden i Harbor klassen som håndterer å laste av en kontainer fra en krans last, til en lastebils lasterom. Her kan det i teorien oppstå feil hvis lastebilen enten har forlatt havnen (altså ikke tilgjengelig for kranen å laste til) eller hvis truck objektet ikke eksisterer i havnen som brukes i simuleringen. For eksempel hvis en feil har skjedd der ett truck objekt har blitt opprettet, men ikke blitt lagt til havnen som brukes i simuleringen.

Disse tilfellene ser vi på som så spesefikke til vårt rammeverk at vi har valgt å opprette egene exception klasser for disse tilfellene.

Ett eksempel på løsning av en slik feil kan være å enten legge til truck objektet som blir brukt til havnen eller opprette ett nytt truck objekt som brukes istede. Disse løsningene er igjen spesifikke løsninger for vårt rammeverk så vi synes å ha egne exceptions gir mening i disse tilfellene. Dette er imidlertidig feil som går på implementasjonsiden og ikke API siden av rammeverket og er derfor ikke feil som brukeren selv kan gjøre i rammeverket.

Vi har allikevel valgt å gjøre disse exception-klassene public slik at hvis en slik feil skulle forekomme har brukeren tilgang til informasjonen om hvordan disse klassene ser ut.

- ‘’CONSIDER creating and throwing custom exceptions if you have a program error condition that can be programmatically handled in a different way than any other existing exception.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 261)

- ‘’DO NOT return error codes.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 255.)
- ‘’DO NOT have public members that can either throw or not throw based on some option.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 258)
- ‘’DO NOT create new exception types to communicate usage errors. Throw one of the existing base library exceptions instead.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 261)
- ‘’DO NOT create a new exception type if the exception would not be handled differently than an existing framework exception.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 262.).

- ''DO NOT create and throw new exceptions just to have your feature name in the exception type or namespace name.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 263.)
- ''DO NOT disclose security-sensitive information in exception messages.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 265.)
- ''DO NOT throw System.Exception or System.SystemException.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 274.)

Kapitel 8.2: Usage Guidelines – Arrays

- ''DO prefer using collections over arrays in public APIs.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 287)

Kapitel 8.3: Usage Guidelines – Collections

Det er deler av rammeverket vårt hvor det er behov for å returnere read only collections til brukeren. Dette gjøres i de tilfellene der endringer brukeren kan gjøre i innholdet til listene som returneres vil ødelegge informasjonen som lagres i listen.

Ett eksempel på dette er listene som er tilgjengelige DailyLog klassen. DailyLog klassen brukes for å lage DailyLog objekter som inneholder informasjon om statusen til havnen på ett gitt tidspunkt i simuleringen.

For å få tilgang til denne informasjonen er det viktig at bruker kan hente ut verdiene i kollectionene som disse objektene inneholder. Dessverre for de fleste liste/collection former hvis brukeren får tilgang til pekeren til listen gjennom metodekall vil de også få tilgang til selve listen pekeren peker mot. Hvis brukeren endrer på verdiene i disse listene vil den historiske dataen som disse listene inneholder bli ødelagt.
En problemstilling brukeren kanskje ikke er klar over.

ReadOnlyCollection unngår at slike brukerfeil oppstår ettersom brukeren får tilgang til listen, men ikke kan gjøre endringer på informasjonen listen inneholder. Boken anbefaler også denne liste/collection formen for slike tilfeller. Vi benytter derfor ReadOnlyCollection i alle tilfeller der endring av dataene i listen vil ødelegge for informasjonen som generes av simuleringen.

Dette er en begrensning som settes for bruker som kan minimere fleksibiliteten til rammeverket, men i disse tilfellene tenker vi at verdien av denne begrensningen overveier nytten brukeren vil ha av å kunne endre disse dataene.

Det er ikke alle lister som trenger å være read only. Ett eksempel er ContainersOnBoard listen i ship klassen som er direkte tilgjengelig for brukeren å endre på.

```
public IList<Container> ContainersOnBoard {  get; } = new List<Container>();
```

Her kan brukeren selv hente ut og legge til container objekter slik de ønsker, som gir mer fleksibilitet for brukeren når de oppretter skip som skal brukes i simuleringen. Siden denne listen ikke inneholder historiske data gjør det ikke så mye om brukeren selv endrer informasjonen som ligger i listen.

Her gjelder DO reglene fra side 298 og 302 nevnt nedenfor.

- ''DO use ReadOnlyCollection<T>, a subclass of ReadOnlyCollection<T>, or in rare cases IEnumerable<T> for properties or return values representing read-only collections.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 298)
- ''DO prefer collections over arrays.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 302)

Vårt rammeverk bruker en rekke lister for å spesifisere lokasjonen på objekter i simuleringen. For eksempel er lokasjonen til skip definert i en rekke lister som listene Anchorage, Transit, LoadingDocks og ShipDocks. Mange av disse listene er ikke direkte tilgjengelig for brukeren gjennom API, men brukeren kan få tilgang til dem indirekte gjennom Log klassene og gjennom metodekall. Brukeren blir også spurt om å sende inn sine egene lister gjennom konstruktøren til Harbor som derretter brukes av simuleringen. Det har derfor vært viktig for oss at alle listene som sendes inn, kopieres eller returneres direkte gjennom APIet er fornuftig satt opp slik at det blir intuetivt for brukeren å bruke listene.

ArrayList er fristende å bruke for disse listene ettersom det er en kjent listeform, men den har en rekke utfordringer fra ett brukerperspektiv. Arraylister strongly typed. Det er derfor vanskelig for både bruker og utvikler å vite hva det er de får ut av listene som sendes inn eller ut av APIet, noe som kan føre til forvirring for brukeren som får ut lister

av APIet eller problemer som kan lede til exeptions for rammeverket som får inn lister fra brukeren hvis disse listene ikke inneholder de forventede dataene.

Vi har derfor valgt å gå for listeformen IList<T> for alle lister som brukeren sender inn gjennom APIet. Dette er den enkleste listeformen og burde derfor være kjent for brukeren. I tilleg tvinger den brukeren til å sette innholdsverdien til listen noe som minimerer feil brukeren kan gjøre når de skal sende denne listen inn gjennom APIet.

Her gjelder DO NOT reglene fra side 294 og 295 fra boka nevnt nedenfor.

- ''DO NOT use weakly typed collections in public APIs.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 294)

Når det kommer til hashtable og dictionary så er det enkelte steder hvor det er behov for å returnere key-value pair lister. Tilfeller av dette i vårt rammeverk er steder der der det er nødvendig å se relasjonen mellom to forskjellige verdier.

Ett eksempel på dette er metoden GetStatusAllShips som skal gi brukeren den nåværende statusen for alle skipene i simuleringen. Her vil det ikke være nok å returnere enten statusene til alle skipene eller bare skipene, her må begge returneres slik at brukeren kan se relasjonen mellom disse to bitene av informasjon.

Vi gikk for listetypen IDictionary<TKey, TValue>. Denne listeformen har fordelen med at verdiene i dictionarien må settes eksplisitt slik at det ikke blir noen forvirring fra brukerens side hva det er de får tilbake av denne listeformen. Her passer vi også på at hvis det ikke finnes noe informasjon å hente ut vil brukeren få en tom IDictionary returnert istedefor en Null verdi.

```
public IDictionary<Ship, Status> GetStatusAllShips()
{
    IDictionary<Ship, Status> statusOfAllShips = new Dictionary<Ship, Status>();

    foreach (Ship ship in AllShips)
    {
        StatusLog test = ship.HistoryIList.Last();
        if (test != null)
        {
            statusOfAllShips[ship] = test.Status;
        }
    }
    return statusOfAllShips;
}
```

Boken sier dette om lister for Key-value pairs:

- "DO NOT use ArrayList or List<T> in public APIs (...) Instead of using these concrete types, consider Collection<T>, IEnumerable<T>, IList<T>, IReadOnlyList<T>, or other collection abstractions. " (Cwalina; Jeremy; Brad. et al., 2020 s. 295).
- “DO NOT use Hashtable or Dictionary<TKey,TValue> in public APIs. (...) Public APIs should use IDictionary, IDictionary <TKey, TValue>, or a custom type implementing one or both of the interfaces.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 296).

Når det kommer til brukervennligheten til listene som returneres av APIet gir boken råd gjennom DO NOT regelen fra side 301.

Det å returnere Null gjennom metodekall hvor det forventes å få tilbake en liste kan fort skape problemer for bruker og lede til situasjoner der det oppstår exceptions. Vi passer derfor på at alle steder der brukeren forventer å få en liste tilbake at de faktisk får det, selv om denne listen ikke inneholder noe data. Ett eksempel på dette er igjen ContainersOnBoard listen i ship som returnerer listen selv om den er tom.

```
public IList<Container> ContainersOnBoard {  get; } = new List<Container>();
```

- ‘’DO NOT return null values from collection properties or from methods returning collections. Return an empty collection or an empty array instead.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 301)

Kapitel 8.4: DateTime and DateTimeOffset

- “DO use DateTime with a 00:00:00 time component rather than DateTimeOffset to represent whole dates, such as a date of birth” (Cwalina; Jeremy; Brad. et al., 2020 s. 307).

Når vi skulle begynne med simuleringen måtte vi ha en måte å loggføre forskjellige hendelser til ulike tider. Vi kom frem til at vi skulle bruke klokkeslett og dato som finnes i DateTime.

Simuleringen tar utgangspunkt i klokkeslettet og datoen som er i øyeblikket simuleringen kjøres, altså datetime.now() som blir satt i en variabel som heter currentTime. Det er også en variabel som holder på hvor lenge en simulering skal kjøre, denne variabelen legger bare til antall dager på currentTime. vi vurderte også å bruke DateTimeOffsett, boken sier dette:

- ‘’DO use DateTimeOffset whenever you are referring to an exact point in time. For example, use it to calculate “now,” transaction times, file change times, logging event times, and so on.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 307)
- ‘’DO use DateTime for any cases where the absolute point in time does not apply, such as store opening times that apply across time zones.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 307)

Etter vi hadde lest litt mer om dette virket det ut som for oss at dette gjaldt for forskjellige tidssoner, vi valgte derfor å gå for DateTime vi fant ut av at dette var mest relevat siden vår simulering ikke trenger å ta hensyn til tidssoner.

- ‘’DO use DateTime when the time zone either is not known or is sometimes not known. This may happen in cases where it comes from a legacy data source.’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 307)

Kapitel 8.8: Nullable<T>

Nullable<T> har vært en sentral og bevisst del av utviklingen av rammeverket.
Store deler av simuleringen tar i bruk Nullable<T>, for å sikre at ønskede resultater faktisk skjer, og forbedrer feilhåndtering.

Et eksempel er gjennom metodene som tar seg av flyttet av container fra skip til lagringsplass.

Nederste metode, ShipToCrane, vil returnere Container objektet som den laster av fra skipet til kran – hvis det *ikke* er noen Container å laste av, returnerer metoden i stedet *null*.

Dette er mulig med bruk av Nullable<T>, hvor vi setter returtypen Container til 'Container?', hvor spørsmålstegnet er et C# alias for Nullable<Container>.

Når skipet er tomt, og med det ikke returnerer noen container, vil de resterende ytre metodene respondere basert på dette.
I den yterste metoden, UnloadShipForOneHour, sjekker den om den tenkt flyttede containeren eksisterer (har variabelen mottatt et Container objekt?) - hvis den har det, rapporterer den videre at en container har blitt lastet av.

- "CONSIDER using Nullable<T> to represent values that might not be present (i.e., optional values). For example, use it when returning a strongly typed record from a database with a property representing an optional table column." (Cwalina; Jeremy; Brad. et al., 2020 s. 311)

Vi har videre også passet på å ikke bruke Nullable<T> ved valgfrie parametere, som ved overload konstruktøren for Harbor-klassen.

- "DO NOT use Nullable<T> unless you would use a reference type in a similar manner, taking advantage of the fact that reference type values can be null." (Cwalina; Jeremy; Brad. et al., 2020 s. 312)

Kapitel 8.9: Object

I rammeverket vårt har vi overskrevet ToString i alle klasser som er public.
Standard ToString for objekter er ikke så veldig nyttig og vi mener det er mye bedre for brukeren å få ut en string som faktisk er lesbar og inneholder informasjom om objektet.

- "DO override ToString whenever an interesting human-readable string can be returned.'' (Cwalina; Jeremy; Brad. et al., 2020 s. 316)

Når det kommer til lengden på stringen som returneres har vi prøvd å holde den så kort som vi mener er hensiktsmessig for de forskjellige klassene.

- “DO try to keep the string returned from ToString short.” (Cwalina; Jeremy; Brad. et al., 2020 s. 317).

Vi har også i stor grad prøvd å legge til informasjon i ToStringen som gjenspeiler det spesifikke objektet som metoden kalles fra.

Man kan se ett eksempel på dette i ToString metoden til Container klassen:

```
public override String ToString()
{
    return ($"ID: {ID}, Size: {ContainerSize}, Weight: {WeightInTonn} tonnes");
}
```

Her får man ut data som størrelse, vekt og ID til containeren som gjenspeiler det spesifikke objektet som metoden kalles på.

Vi mener dette gjør stringen som returneres mer informativ og gjør at man kan bruke ToString metoden til å sammenligne informasjon fra objekter av samme klasse.

- ‘’CONSIDER returning a unique string associated with the instance (of the object).’’ (Cwalina; Jeremy; Brad. et al., 2020 s. 317)

Det er enkelte klasser, for eksempel Harbor klassen som har en ToString som går over 2 linjer. Men vi ønsker at ToStringen faktisk skal gi nyttig informasjon om tilstanden til objektet samtidig som informasjonen som gis må være beskrivende nok til at det ikke skal bli forvirrende for brukeren hva informasjonen i stringen faktisk betyr.

Vi anser derfor disse stringene til å være så korte som mulig uten at stringen mister informasjonsverdi for brukeren. Vi har også passet på at ToString altid returnerer en string verdi og aldri returnerer en null verdi. Noe som kunne ha skapt problemer for brukeren og lede til mulige exeptions.

- “DO NOT return an empty string or null from ToString. " (Cwalina; Jeremy; Brad. et al., 2020 s. 318.)

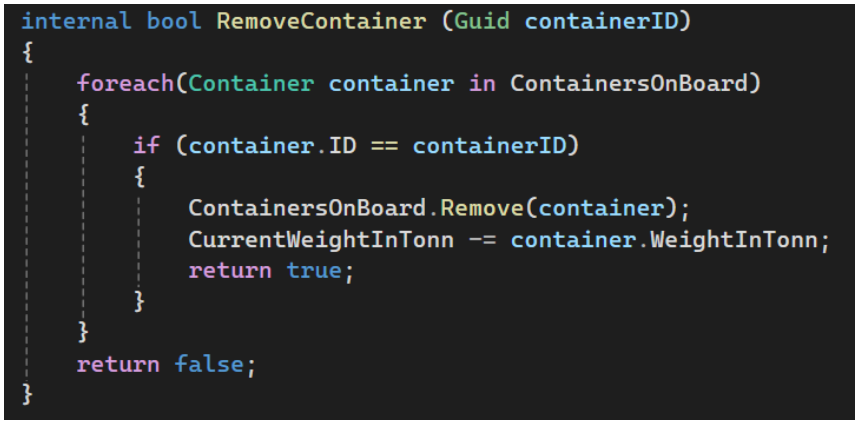
Appendix A: C# Coding Style Conventions

Gjennom prosjektet har vi passet på å bruke C# stil-konvensjoner.

Krøllparentes (braces)

Vi følger generelle regler når det kommer til bruk av krøllparenteser, også kalt Allman-style bracing:

- "DO place the opening brace on the next line, indented to the same level as the block statement." (Cwalina; Jeremy; Brad. et al., 2020 s. 466.)
- "DO align the closing brace with the opening brace." (Cwalina; Jeremy; Brad. et al., 2020 s. 466.)
- "DO place the closing brace on its own line, except the end of a do..while statement." (Cwalina; Jeremy; Brad. et al., 2020 s. 467.)



```
internal bool RemoveContainer (Guid containerID)
{
    foreach(Container container in ContainersOnBoard)
    {
        if (container.ID == containerID)
        {
            ContainersOnBoard.Remove(container);
            CurrentWeightInTonn -= container.WeightInTonn;
            return true;
        }
    }
    return false;
}
```

Vi passer også på og alltid bruke braces, selv om det er en-linje kode:

- "AVOID omitting braces, even if the language allows it." (Cwalina; Jeremy; Brad. et al., 2020 s. 467.)
- DO NOT use the braceless variant of the using (dispose) statement.

Mellomrom (space usage)

Vi følger riktig bruk av mellomrom mellom alle deler av koden, som metoder og parenteser, parametere, argumenter og operatorer:

- “DO use one space before and after the opening and closing braces when they share a line with other code. Do not add a trailing space before a line break.” (Cwalina; Jeremy; Brad. et al., 2020 s. 469.)
- “DO use a single space after a comma between parameters.” (Cwalina; Jeremy; Brad. et al., 2020 s. 469.)
- “DO use a single space between arguments.” (Cwalina; Jeremy; Brad. et al., 2020 s. 469.)
- “DO NOT use spaces after the opening parenthesis or before the closing parenthesis.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)
- “DO NOT use spaces between a member name and an opening parenthesis.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)
- “DO NOT use spaces after or before square braces.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)
- “DO use spaces between flow control keywords and the open parenthesis.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)

```
while (currentTime < endTime)
{
    SetStartLocationForAllShips();
```

- “DO use spaces before and after binary operators.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)

```
else if (truck.Container != null)
{
```

- “DO NOT use spaces before or after unary operators.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)

```
if (!loadingDock.TruckExistsInTruckLoadingSpots(truck))
{
```

Vi bruker riktig form av innrykk (indent), med fire mellomrom:

- “DO use four consecutive space characters for indents.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)
- “DO NOT use the tab character for indents.” (Cwalina; Jeremy; Brad. et al., 2020 s. 470.)

- “DO indent contents of code blocks.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 471.)

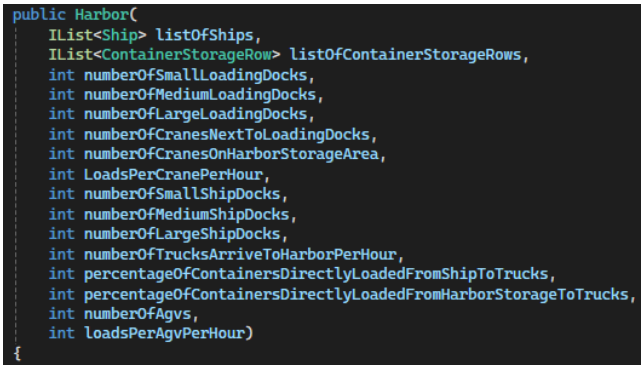
Innrykk (Indents)

Vi følger også riktige konvensjoner når det kommer til bruk av innrykk for lesbarhet av parametere og lange kodelinjer.

- “DO indent all continued lines of a single statement one indentation level.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 471.)

Vi bruker skjønn og ser på lesbarheten når vi bestterm om parameterlisten eller argumentlisten er “exceedingly long”. I tillegg har vi tatt med oss tankene til Jeremy Barton, hvor han har sin IDE-guidelines satt til 120 karakter, som han anser som en “soft limit” (Cwalina; Jeremy; Brad. et al., 2020 s. 472.)

- “DO chop before the first argument or parameter, indent one indentation level, and include one argument or parameter per line when a method declaration or method invocation is exceedingly long.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 472.)



Whitespace

Vi har riktig bruk av whitespace mellom statements, metoder og avsluttende krøllparenteser, bortsett fra når neste linje også er avsluttende krøllparentes. I tillegg bruker vi whitespace for å forbedre lesbarheten:

- “DO add a blank line before control flow statements.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 472.)

- “DO add a blank line after a closing brace, unless the next line is also a closing brace.” (Cwalina; Jeremy; Brad. et al., 2020 s. 472.)
- “DO add a blank line after “paragraphs” of code where it enhances readability.” (Cwalina; Jeremy; Brad. et al., 2020 s. 473.)

```
if (ShipDockingForMoreThanOneHour(lastStatusLog))
{
    Guid dockID = lastStatusLog.SubjectLocation;

    ShipIsDockedToLoadingDock(ship, dockID);

    if (ShipFromTransitWithContainersToUnload(ship))
    {
        StartUnloadProcess(ship, dockID);
    }

    else if (ShipHasNoContainers(ship) && !ship.IsForASingleTrip)
    {
        StartLoadingProcess(ship, dockID);
    }
}
```

Member modifisers

Ved bruken av modifiers, passer vi på plasseringen av disse i riktig rekkefølge som rådet av boka:

- “DO always explicitly specify visibility modifiers.” (Cwalina; Jeremy; Brad. et al., 2020 s. 474.)
- “DO specify the visibility modifier as the first modifier.” (Cwalina; Jeremy; Brad. et al., 2020 s. 474.)

```
public abstract class HistoryRecord
{
```

- “DO specify the static modifier immediately after visibility, for static members or static classes.” (Cwalina; Jeremy; Brad. et al., 2020 s. 474.)

```
private static bool ShipHasNoContainers(Ship ship)
{
```

- “DO specify the readonly modifier for fields or methods immediately after the member slot modifier (if specified).” (Cwalina; Jeremy; Brad. et al., 2020 s. 474.)

```
private readonly DateTime startTime;
```

Annet

Vi har med den "alternative" etterfølgende kommaen på slutten av alle de siste enum-medlemmene:

- "DO add the optional trailing comma after the last enum member." (Cwalina; Jeremy; Brad. et al., 2020 s. 475.)

```
public enum ShipSize
{
    /// <summary>
    /// No size
    /// </summary>
    None = 0,
    /// <summary>
    /// Small ship. base weight: 5000 tonns, container capacity: 20.
    /// </summary>
    Small = 5000,
    /// <summary>
    /// Medium ship, base weight: 50000 tonns, container capacity: 50
    /// </summary>
    Medium = 50000,
    /// <summary>
    /// Large ship, base weight: 100000 tonns, container capacity: 100
    /// </summary>
    Large = 100000,
}
```

Vi bruker ikke "this." - for eksempel i konstruktører eller annen interne tildelinger, om ikke nødvendig (som ved kall på eventer (this,

- "DO NOT use this. unless absolutely necessary." (Cwalina; Jeremy; Brad. et al., 2020 s. 476.)

  Nødvendig:

```
SimulationStarting?.Invoke(this, simulationStartingEventArgs);
```

  Ingen bruk av 'this.':

```
internal LoadingDock(ShipSize shipSize) : base(shipSize)
{
    Size = shipSize;
}
```

Vi bruker "language keywords" i stedet for BCL type navn:

- "DO use language keywords (string, int, double, ...) instead of BCL type names (String, Int32, Double, ...), for both variable declarations and method invocation." (Cwalina; Jeremy; Brad. et al., 2020 s. 476.)

Vi bruker "var" på det vi mener er steder hvor typen er åpenbar (for eksempel for-løkker) og på akseptabelt vis etter regel i boken:

- "DO NOT use the var keyword except to save the result of a new, as-cast or "hard" cast." (Cwalina; Jeremy; Brad. et al., 2020 s. 476.)

```
foreach (var pair in loadingDock.TruckLoadingSpots)
{
    if (pair.Value?.Container == container)
    {
        truck = pair.Value;
        break;
    }
}
```

Vi tar også i bruk "expression-bodied members" når propertien ikke forandrer seg, som ved readonly lister:

- "CONSIDER using expression-bodied members when the implementation of a property or method is unlikely to change."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 478.)

```
public override ReadOnlyCollection<DailyLog> History => HistoryIList.AsReadOnly();

internal IList<DailyLog> HistoryIList { get; } = new List<DailyLog>();
```

Vi bruker nameof() i stedet for string, for eksempel ved exceptionshåndteringer:

- "DO use the nameof(...) syntax, instead of a literal string, when referring to the name of a type, member, or parameter." (Cwalina; Jeremy; Brad. et al., 2020 s. 479.)

Vi bruker readonly modifier bevisst som for eksempel på start og sluttidene i simuleringen:

- "DO apply the readonly modifier to fields when possible."
  (Cwalina; Jeremy; Brad. et al., 2020 s. 479.)

```
private readonly DateTime _startTime;

private DateTime _currentTime;

private readonly DateTime _endTime;
```

Navngiving

Vi følger først og fremst retningslinjene fra Kapittel 3: Naming conventions, som det også er en regel om i Appendix A:

- “DO follow the Framework Design Guidelines for naming identifiers, except for naming private and internal fields.” (Cwalina; Jeremy; Brad. et al., 2020 s. 480.)

I tillegg er vi bevisste på riktig PascalCasing og camelCasing i intern kode.

- “DO use camelCasing for private and internal fields.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 481.)
- “DO use camelCasing for local variables.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 482.)
- “DO use camelCasing for parameters.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 482.)

I tillegg bruker vi underscore prefix (“_”) for private og interne instans-fields, som starttid og sluttid i simuleringen. Underscore prefix brukes altså for private fields, og ikke properties.

- “DO use a prefix “_” (underscore) for private and internal instance fields, “s_” for private and internal static fields, and “t_” for private and internal thread-static fields.” (Cwalina; Jeremy; Brad. et al., 2020 s. 481.)

```
private readonly DateTime _startTime;

private DateTime _currentTime;

private readonly DateTime _endTime;
```

Klassene er også strukturert etter bokens to regler om fil organisering:

- “DO group members into the following sections in the specified order:
  - All const fields
  - All static fields
  - All instance fields
  - All auto-implemented static properties
  - All auto-implemented instance properties
  - All constructors
  - Remaining members
  - All nested types”

  (Cwalina; Jeremy; Brad. et al., 2020 s. 484.)

- “CONSIDER grouping the remaining members into the following sections in the specified order:
    - Public and protected properties
    - Methods
    - Events
    - All explicit interface implementations
    - Internal members
    - Private members”

  (Cwalina; Jeremy; Brad. et al., 2020 s. 484.)

Relatert til vårt API og rammeverk, med de “members” vi har, har vi tolket retningslinjene slik …

- Alle fields skal først (f.eks _startTime, _endTime etc. i SimpleSimulation)
  (DO – All const, static, instace fields)
- Så alle properties (med {get; set;})
  (DO – All auto-implemented instance properties)
- Så alle konstruktører
  (DO – All constructors)
- Så alle metoder
  (CONSIDER – Methods)
- Events (relevant i klassen SimpleSimulation)
  (CONSIDER – Events)
- Så alle internal og private members som er igjen
  (CONSIDER – Internal members + Private members)

Vi har også passet på å plassere alle “using” utenfor namespace og riktig rekkefølge på “using” setninger:

- “DO place the using directives outside the namespace declaration.”
  (Cwalina; Jeremy; Brad. et al., 2020 s. 485.)
- “DO sort the using directives alphabetically, but place all System namespaces first.” (Cwalina; Jeremy; Brad. et al., 2020 s. 485.)

Vedlegg 3: Utlevert materiale for første runde med brukertester.

(Se separat vedlegg)

Vedlegg 4: Utlevert materiale for andre runde med brukertester.

(Se separat vedlegg)

Vedlegg 5+++: Andre vedlegg

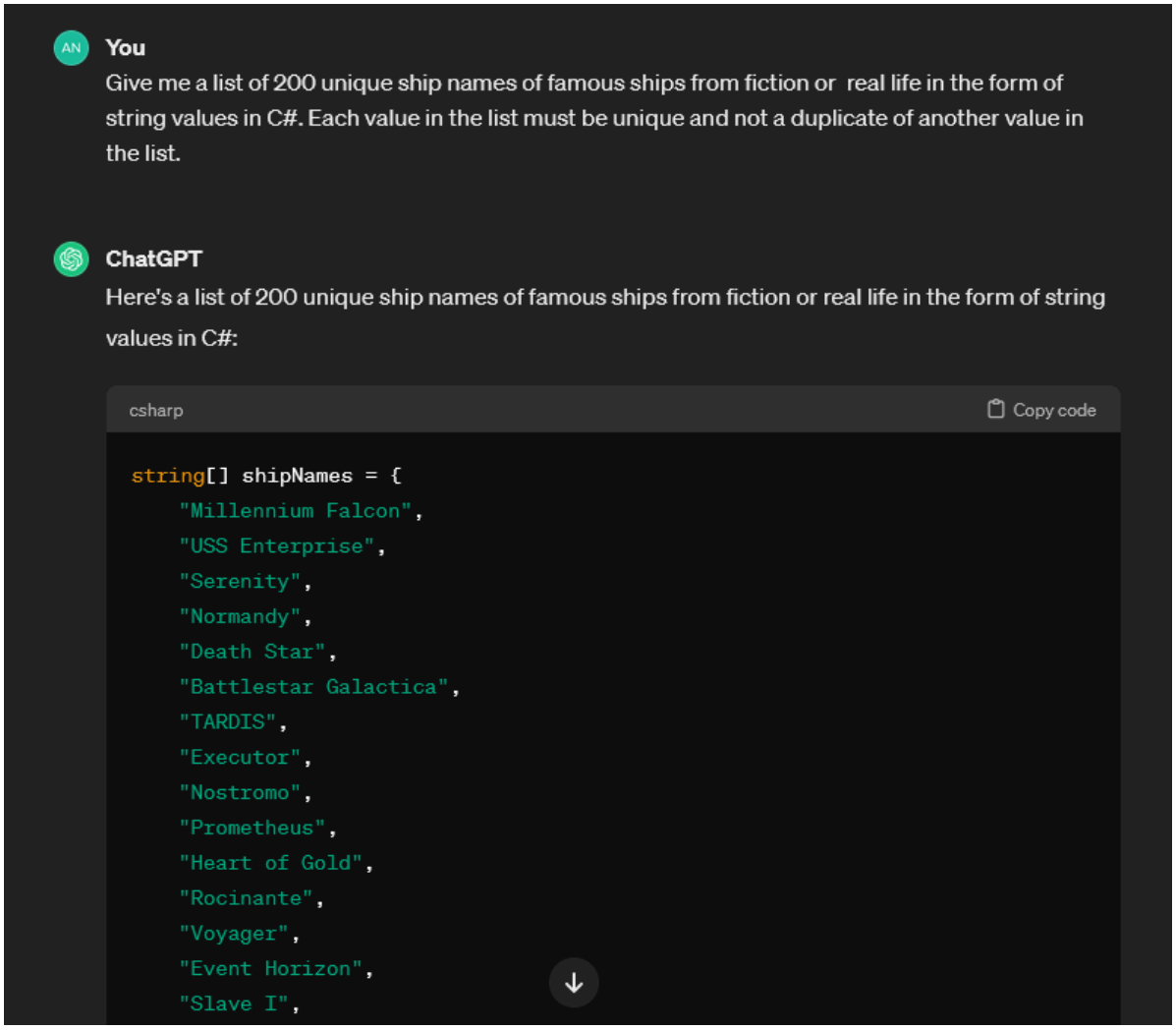
(Se vedlagt kode)

Kilder:

Cwalina, Krzysztof; Barton, Jeremy; Abrams, Brad. (2020) Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries Pearson Education. (Kindle Edition.) Addison-Wesley Microsoft Technology

OpenAI. (2023). *ChatGPT* (25. mars versjon) [Stor språkmodell]. https://chat.openai.com/

Prompt:


You
Give me a list of 200 unique ship names of famous ships from fiction or real life in the form of string values in C#. Each value in the list must be unique and not a duplicate of another value in the list.
ChatGPT
Here's a list of 200 unique ship names of famous ships from fiction or real life in the form of string values in C#:
csharp
Copy code
string[] shipNames = {
"Millennium Falcon",
"USS Enterprise",
"Serenity",
"Normandy",
"Death Star",
"Battlestar Galactica",
"TARDIS",
"Executor",
"Nostromo",
"Prometheus",
"Heart of Gold",
"Rocinante",
"Voyager",
"Event Horizon",
"Slave I",