Design Like a Robot!
Rulebook

[v0_03, door Marius Versteegen, 2021]

Over dit document

Dit “Rulebook” bevat een extractie van de theoretische delen van het document “Design Like a Robot!”. Dus zonder een in muizestapjes uitgewerkte casus erbij als toelichting. Het is vooral handig als je “Design Like a Robot!” al doorgenomen en begrepen hebt.

95% van het software design proces om te komen van requirements tot een oplossing bestaat uit het eenvoudig volgen van vuistregels. Dus zodra je de vuistregels kent, gaat het design proces al bijna automatisch.

De requirements

Het uitgangspunt voor het maken van een software design bestaat uit requirements. Dat zijn:

- Een use case diagram
- Use case beschrijvingen
- Eventueel een activity diagram ter verduidelijking van de use case beschrijvingen. (niet in dit document)

De design fasen

Vanuit de requirements zijn de vaste stappen om te komen tot een oplossing:

1. Object model met object beschrijvingen
2. Taakstructurering
3. Klassediagram
4. Stds
5. Code
6. Concurrency Diagram

Design Fase1:
Object model met object beschrijvingen

De eerste fase bestaat uit het maken van een object model en parallel daaraan een lijst met object beschrijvingen.

Wat is een Object model?

Voorbeeld van een Object Model

Een object model is een weergave van objecten en de berichten die ze met elkaar uitwisselen. De objecten worden gerepresenteerd door vierkanten met namen er in (BlaKnop en BlaControl). Boven de naam kan een “categorie”/stereotype (bijvoorbeeld <<boundary>>) staan aangegeven. De vierkanten worden met pijlen met elkaar verbonden. De pijlen geven de richting van de berichten aan. Naast de pijlen staan de namen van de berichten.

Wat is een object?

Een object is een stuk software met een bepaalde verantwoordelijkheid. Een softwarematig “ding”.

Welke stereotypen / categorieen objecten zijn er?

Binnen CSM hoef je alleen te verifieren of een object van een van onderstaande stereotypen is:

- <<boundary>> Een boundary object is de grens van de software, en praat met de “buitenwereld”, dwz met de hardware. Voorbeelden zijn sensoren (knoppen, meetinstrumenten) ,actuatoren (motors, displays, speakers etc) en proxy objecten (remote database proxy: een stuk software dat communiceert met een remote database).
- <<control>> Een control object coordineert een groep andere objecten. Het is een soort “dirigent”.
- <<entity>> Een entity object encapsuleert data (in het algemeen via setters en getters of read en write functies). Het is in het algemeen mooi om bij mekaar horende data te groeperen in een entity object. Vooral als die data als pakket zijn weg door het systeem vindt (bijvoorbeeld samen wordt opgeslagen) of op een centrale plek moet staan die voor meerdere objecten toegankelijk is.
- Iets anders Als een object niet in een van bovengenoemde categorien valt, hoef je er geen stereotype voor in te vullen.

Wat is een bericht?

Een bericht is iets wat het ene object tegen het andere object “zegt”. In het uiteindelijke klassediagram wordt het bericht een functie. Een bericht kan een “payload” hebben. Die staat tussen haakjes. In bovenstaand bijvoorbeeld, zegt het het object BlaKnop “buttonpressed(BlaKnop)“ tegen BlaControl. Dus “er is een knop ingedrukt” en het ging om “BlaKnop”. Meerdere berichten kun je langs dezelfde pijl weergeven door ze met comma te scheiden.

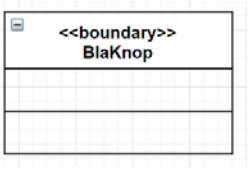
Zo maak je een object model

Het maken van een object model bestaat uit de volgende stappen:

1. Maak een controller object aan voor elke use case.
    a. Het aanmaken van een object gaat in 3 stappen:
        i. Voeg het toe aan het object model.
        ii. Geef het een eventueel geschikt stereotype (zie even verderop).
        iii. Voeg een beschrijving ervan toe aan de lijst met objecten.
2. Verwerk de use case beschrijvingen zin voor zin:
    a. Lees een zin.
    b. Bedenk welke “dingen” je systeem nodig heeft om te kunnen doen wat er staat. (vaak staan ze al in de zin als zelfstandige naamwoorden)
    c. Maak voor elk van die “dingen” een object aan (weer in 3 stappen):
        i. Voeg het toe aan het object model.
        ii. Geef het een eventueel geschikt stereotype (zie even verderop).
        iii. Voeg een beschrijving ervan toe aan de lijst met objecten.
    d. Ga na welk “ding” iets tegen welk “ander ding” “zegt” om de zin waar te maken.
    e. Zet wat er “wordt gezegd” als bericht langs een pijl vanuit het sprekende object naar het luisterende object.

Praktische Tips

- Als je opdrachtgever het toelaat, gebruik dan tijdens het tekenen van je object model geen vierkanten, maar gebruik klasse placeholders (dat is eigenlijk niet officieel, maar bespaart dubbel werk wanneer je in een latere design fase je object model omzet in een klassediagram):



  (in drawIO kun je in het member-veld en het functie-veld een spatie invullen, dan kun je daar later altijd nog - na het omzetten van object model naar klassediagram – de members en functies invullen).
- Gebruik geen spaties of speciale karakters in de namen voor je objecten. Later worden die namen immers de uitgangspunten voor de klasse namen.
- Gebruik geen spaties of speciale karakters in de namen voor je berichten of hun payload. Later worden die namen immers de uitgangspunten van publieke operaties van je klassen.
- Voor use cases waarvan een activity diagram beschikbaar is, kun je de bovenstaande stappen (nog makkelijker-) uitvoeren door stap voor stap door dat activity diagram te lopen (in plaats van zin voor zin door de use case beschrijving. In dit document gaan we even uit van het worst case geval – dus dat er geen activity diagram beschikbaar is.

Timers

Wel of geen object aanmaken voor een timer?

Timers zijn waitables zoals flags en channels. In het algemeen voegt het niet veel toe om waitable objecten expliciet in het object model op te nemen. Tijdens het omzetten van het STD naar Code volgt al op logische wijze welke waitables je moet aanmaken.

Okee, tot zover de theorie voor het maken van een object model. Now, let's Due it!

Databeheer

Publieke data

NB: we hebben het hier over "**publieke data**" die wordt **uitgewisseld** tussen objecten. Als het slechts binnen een object gebruikt wordt, kan het verborgen kunnen blijven, en zien we het niet in het objectmodel terug.

Waar "zit" die publieke data nu?

In het algemeen probeer je voor elke publieke data in je systeem te laten beheren door een enkele "**(hoofd-) eigenaar**". Als je de data wilt opvragen, kun je die eigenaar raadplegen. Wil je de data veranderen, dan kun je dat bij die eigenaar aanvragen. Je kunt ervoor kiezen om een entity object (hoofd-) eigenaar te maken, of een ander object.

Wat is netjes? Data beheer door entity object of ander object?

Het is in het algemeen netjes dat data wordt beheerd in het object **waar het het meest mee verbonden is / wordt "gebruikt"**. In dat geval hoef je er geen apart entity object voor te maken, en encapsuleer je de data in het betreffende object.

De trilfrequentie wordt weliswaar ingesteld in InstelControl, maar wordt echt gebruikt in TrilControl, je zou het dus prima kunnen encapsuleren in TrilControl.

Maar wanneer gebruik je dan een entity object?

Dat kun je doen in de volgende gevallen:

- **Datapakket**
- **Geen logische eigenaar**

Datapakket

Het had ook kunnen zijn dat trilfrequentie data is die in **samenhang met andere data als een geheel** kan worden **gezien**, **behandeld**, **weggeschreven of verzonden**, als **datapakket**. In dat geval is het netter om die bij mekaar horende data te bundelen in een apart "**entity object**".

Bijvoorbeeld als de data zich bevindt in traag geheugen, en/of om bij mekaar horende data logisch te groeperen en/of als de data om de een of andere reden geencapsuleerd moet worden. Bijvoorbeeld bij een database of bij data die via encryptie wordt opgeslagen en opgevraagd.

Bij ons trilapparaat is er alleen een Trilfrequentie die ingesteld moet worden. In een wat ingewikkelder, realistischer voorbeeld wil je misschien meerdere data instellen, die tesamen een entiteit "programma" vormen. Dat bevat dan bijvoorbeeld Trilfrequentie, Trilintensiteit en Loopduur.

Je kunt zo'n entiteit dan bijvoorbeeld in zijn geheel aanpassen met InstelControl en eventueel lezen of schrijven naar disk met een DiskControl.

Stel nu dat je in zo'n geval DiskControl weglaat, en de entiteit "programma" als geheel **alleen** wordt gebruikt door InstelControl. Het gaat dan alleen nog om een conceptuele groepering van data. In dat geval zou je er ook voor kunnen kiezen om die data toch te **encapsuleren** in InstelControl (in plaats van in een apart entity object. Dat is een kwestie van smaak.

Geen logische eigenaar

In het geval van de Trilfrequentie is het wel duidelijk dat die het meest verbonden is met TrilControl. Het kan ook voorkomen dat je data tegenkomt die door meerdere objecten wordt gebruikt, maar waarvan het niet duidelijk het meest bij een enkel object hoort.

Ook in zo'n geval is het netter om een apart entity object eigenaar voor die data te laten zijn.

Taakstructurering

Zodra de lijst van objecten en hun beschrijvingen compleet is, moet aan elk van de objecten een van de volgende "taaksoorten" worden toegekend:

1. IO (input output) : dat zijn taken van boundary objecten.
   Daarbij moet ook het subtype worden aangegeven:
   a. Periodical
      -> wanneer iets met vast interval moet worden gedaan, zoals pollen.
   b. Event driven
      -> wanneer er iets wordt gedaan na optreden van een hardware interrupt (komt verder niet voor in deze cursus)
   c. Demand driven
      -> overig: gaat pas iets doen als er een bericht binnenkomt.
2. Intern : dat zijn taken van overige objecten.
   a. Periodical
      -> wanneer iets met vast interval moet worden gedaan, zoals pollen.
   b. Demand driven
      -> overig: gaat pas iets doen als er een bericht binnenkomt.
3. Geen taak
   Soms kun je direct al besluiten dat een object geen eigen taak nodig heeft, bijvoorbeeld als hetgeen wat hij doet vrijwel instantaan is. Voorbeelden: entitiy objecten, leds.

Taakstructurerings-tabel

De objecten en hun klassificering zet je op een rijtje in een tabel:

Objecten	Taaksoort	Periode	Deadline	Prioriteit

Behalve de objectnaam en de taaksoort vul je per object ook een deadline en eventueel een periode in.

Periode

Objecten met als taaksoort “Periodical IO of Periodical intern” komen met de regelmaat van een klok in actie om iets te doen.

Deadline

Een deadline is de tijd die het object er over mag doen om “zijn ding te doen” nadat hij de beurt heeft gekregen.

NB:

- Als iets niet heel tijd-kritisch is (bijvoorbeeld het doorgeven van de hoeveelheid zon voor een zonnescherm), dan mag daar best lang (een minuut?) over gedaan worden. Dat het in praktijk binnen een microseconde afgehandeld wordt, speelt daarbij geen rol.
- Andersom, als je gebonden bent aan bepaalde hardware die nu eenmaal niet snel is (bijvoorbeeld als een refresh van je display 200ms kost), en je zou een deadline van kleiner dan 200ms kiezen, dan zou je meteen kunnen ophouden: je weet dan bij voorbaat dat die deadline nooit gehaald zal worden met dat display.
- Een deadline zegt niet dat de betreffende taak al die tijd de thread voor zichzelf mag houden. Het object met de kleinste deadline moet immers ook zijn deadline kunnen halen. Bijvoorbeeld, je hebt:
    - Een BitSend object met een deadline van 50uS
    - Een display met een deadline van 200ms.

      De display mag dan weliswaar in totaal 200ms over zijn update doen, maar hij moet binnen die tijd vaak genoeg zijn thread afstaan (middels een hwlib_waitus(1) bijvoorbeeld)) om ervoor te zorgen dat het BitSend object vaak genoeg de kans krijgt om zijn deadline te halen. “vaak genoeg” betekent in dit geval dat elk tijdsinterval dat display de thread voor zich alleen claimt “substantieel lager” moet zijn dan 50us. Laten we zeggen: 10us.

Prioriteit

Nadat alle deadlines zijn ingevuld, geef je de objecten prioriteiten. Hogere prioriteiten hebben lagere nummers. Dus prioriteit 1 is een hogere prioriteit dan prioriteit 2. Vuistregel: des te korter de deadline, des te hoger de prioriteit.

Taken samenvoegen

Het laatste deel van de taakstructurering bestaat uit het eventueel samenvoegen van taken. Dat samenvoegen kan op grond van verschillende soorten “cohesie” (dat is een ander woord voor “samenhang”).

Je kunt die samenhang op vele fancy wijzen ontdekken en toepassen maar wij adviseren om het alleen te doen volgens deze vuistregels
(reden: het moet zoden zetten en dingen niet veel complexer maken):

- ### Temporele Cohesie:
  Periodieke taken met dezelfde periode, deadline en prioriteit kun je samennemen.
- ### Control cohesie:
  Demand Driven taken die alleen worden aangestuurd door een enkel ander object kun je soms samenvoegen met dat andere object. Dat kan alleen als wat ze moeten doen “acceptabel kort is”.
  Voorbeeld: in het object model zien we dat Display alleen door InstelControl wordt aangestuurd. In de taakstructureringstabel zien we dat de Display een deadline heeft van 200ms. Als het acceptabel zou zijn dat het InstelControl 200ms moet wachten tot de display update klaar is voordat het kan reageren op nieuwe frequentie-aanpassingen, dan zouden we de taak van de display dus kunnen samenvoegen met die van InstelControl.
  Het lijkt mijn echter fijner dat de trilfrequentieaanpassing zo snel mogelijk wordt verwerkt na het indrukken van een knop. Vandaar dat ik ervoor heb gekozen om display zijn eigen taak te laten houden.
- ### Sequentiele cohesie:
  Als er een aantal objecten altijd vlak na elkaar hun ding moeten doen kun je hun taken samenvoegen. In deze cursus ben ik dat tot nog toe niet tegengekomen.

Wat betekent “Samenvoegen”

Dat kan twee dingen betekenen:

1. Een van de objecten verliest zijn taak. De dingen van het andere object worden (ook) in de thread van de taak van de aanroeper uitgevoerd.
2. In het geval van temporele cohesie:
   Alle betrokken objecten verliezen hun eigen taak. Er komt een extra “Handler” object met een taak. Vanuit dat object worden periodiek “update” functies van de betrokken objecten aangeroepen. Zodoende voeren alle betrokken objecten periodiek hun “ding” uit zonder dat elk van hen een eigen taak nodig heeft.

Waarom Samenvoegen

Elke taak heeft een zekere overhead in cpu time en geheugen. Het doel van taken was om dingen overzichtelijker te maken. Door het samenvoegen van taken kan een balans worden gevonden tussen die twee wensen. Neem bijvoorbeeld de snoepautomaat. Die had 40 buttons en 40 actuatoren om de candy bars uit te werpen. Zonder handler objecten zou je daar 80 afzonderlijke taken voor nodig hebben gehad. Met handler objecten kun je dat op nog steeds overzichtelijke manier met 2 taken af.

Klassediagram maken

Het opstellen van het klassediagram kan in de volgende stappen worden gedaan:

1. **Initiele Klassen**
   Maak een kopie van het object model. Vervang alle overeenkomstige objecten door klassen.
2. **Publieke operaties**
   Schuif de berichten langs de pijlen in de richting van de pijl door naar de klasse die zich daar bevindt. Het bericht wordt daar geimplementeerd als publieke functie. Met andere woorden: elk bericht wordt een publieke operatie bij zijn ontvanger.
   Uitzondering: de timers zijn hier weer de vreemde eend in de bijt. Om op een bericht van een timer te kunnen wachten hoef je geen publieke functie aan je klasse toe te voegen. Je kunt hem gewoon opnemen in een rtos wait instructie.
3. **Verwijder dubbele pijlen**
   Dubbele pijlen zijn dubbelop in een klassediagram.
4. **Representeer gelijksoortige objecten door een enkele klasse**
5. **Composition / Member variabelen**
   Het is handig dat wanneer je een object aanmaakt, dat dat object zelf automatisch zoveel mogelijk de objecten aanmaakt die hij nodig heeft. Dat doe je door het toevoegen van de benodigde objecten als private member variabelen in de member sectie van de klasse. Probeer dus als het even kan composition toe te passen.
   Vuistregel: als een object A alleen communiceert met object B, kun je overwegen om B eigenaar te maken van object A. Test dan nog even of “B heeft een A” natuurlijk klinkt.
   Regel: Zodra B eigenaar is van object A, kan een composition-relatie van B naar A worden getekend (het zwarte wiebertje aan de kant van de eigenaar).
   Als er al een pijl van B naar A liep, mag deze dan worden verwijderd.
   Immers: eigenaarschap impliceert kennen: als je een member variabele hebt, kun je zijn member functies aanroepen. Je hebt daar dan geen reference meer voor nodig.
6. **Member references**
   De pijlen in het klassediagram die niet zijn omgezet in composition relaties betekenen alleen nog “kennen”, geen eigenaarschap (want de pijlen die eigenaarschap vertegenwoordigden zijn in de vorige stap al omgezet naar compositie-relaties). Voeg voor elk object dat een klasse “kent” een private member reference variabele toe.
7. **Listener Patterns**
   Klassen die berichten versturen naar andere klassen moeten die andere klassen kennen. Daardoor werken ze alleen met die andere klassen. Als je klassen algemener toepasbaar/herbruikbaar wilt maken, moet je ze hun berichten laten sturen naar een zogenaamde “Listener”. Dat is een interface klasse die de juiste functies voor de berichten bevat. Zodoende kunnen alle klassen die afleiden van zo’n interface klasse (ook toekomstige klassen) probleemloos naar de verstuurde berichten “luisteren”. Buttons zijn bijvoorbeeld typisch klassen die je in meerdere designs wilt kunnen hergebruiken. *(TODO: insert doc over listener patterns).*
8. **Markeer isActive**
   Markeer klassen die een taak hebben met het (extra) stereotype <<isActive>>.

9. **Voeg Handler klassen toe**
   Voeg de extra Handler klassen toe die zijn ontstaan als gevolg van de taaksamenvoeging van periodieke taken.
10. **Weak dependencies**
    Het kan zijn dat een klasse in een van de functie-parameters van een van zijn functie een klasse gebruikt waarvan hij verder geen member of reference member heeft. In dat geval laat je de afhankelijkheid zien mbv een “weak dependency” relatie (een gestippelde pijl).
    Voorbeeld: Stel je voor dat je Display klasse een DrawRect(rect:Rect) functie heeft.
    De Rect wordt alleen als tijdelijke variabele meegegeven aan de DrawRect functie. De Display klasse heeft verder geen members of member references van type Rect. In dat geval is de Dislay “weak dependent” van Rect, en loopt er een gestippelde pijl van Display naar Rect om dat aan te geven. Hetzelfde geldt als een tijdelijk object van een klasse in een functiebody wordt aangemaakt. Bijvoorbeeld als je display klasse de functie update() heeft, en die functie in zijn body een tijdelijk Rect object aanmaakt.
    Het nut van het ook het weergeven van de weak dependencies is dat als je bijvoorbeeld een library van je klassen wilt maken, dat je ziet van welke andere libraries die allemaal afhankelijk moet zijn en waarom.
11. **Multiplicity**
    Zorg dat aan het uiteinde van de de associatiepijlen en de compositie-relaties de juiste multipliciteit in het klassediagram is aangegeven.
    Tel eenvoudigweg hoeveel members en reference members een klasse heeft.
    Voorbeelden:
    a. 1..3 betekent: 1 tot 3.
       Bijvoorbeeld in het geval van een array met 3 entries waar pointers in opgeslagen kunnen worden.
    b. * betekent: 0 of meer.
       Bijvoorbeeld in het geval van een vector.
    c. 2..* betekent: 2 of meer.
       Bijvoorbeeld: .. ik kan even niets bedenken. Dit komt niet vaak voor.
    d. Niets aangeven betekent: 1.

Synchronisatiemechanismen

Het is goed gebruik om taken alleen te laten communiceren en data te laten uitwisselen via synchronisatiemechanismen.

De voordelen van het gebruik van synchronisatiemechanismen zijn:

1. Taken kunnen dingen aan elkaar doorgeven en vervolgens zelf doorgaan met hun eigen taak (= asynchroniteit).
2. Data wordt veilig doorgegeven tussen de taken. Het RTOS voorkomt bijvoorbeeld door middel van het gebruik van mutexen waar nodig dat data nooit gelijktijdig door meerdere taken wordt bepoteld (anders zou het bijvoorbeeld kunnen voorkomen dat terwijl een struct nog wordt gelezen door de ene taak, dat de andere taak al een deel ervan aan het overschrijven is, waardoor de data niet meer consistent is.

De synchronisatiemechanismen die we binnen rtos gebruiken zijn:

- **Waitables**
  Waitables zijn events waar je middels de rtos functie wait() op kunt wachten. Een waitable hoort altijd bij een enkele taak. Alleen die ene taak kan op de waitable wachten. Zodra een waitable/event waarop gewacht werd “is opgetreden”, wordt hij automatisch weer gereset. Dit zijn de rtos waitables die tot je beschikking staan:
  - **Flag**
    De ene taak set een flag. De andere taak kan er op wachten. Het “afgaan” van een flag betekent “let op”. Niet meer, niet minder. Een flag is dus niet hetzelfde als een bool. Als iemand tegen je zegt “let op”, weet je niet of hij “true” of “false” bedoelt.
  - **Channel**
    Een channel is een FIFO (first in, first out) queue waar je op kunt wachten.
    Zodra er iets in de channel wordt geschreven is zal de er aan gelieerde waitable/event optreden.
    int newTemperature = tempChannel.read();
    Uiteraard moet er een andere taak zijn die iets in de channel schrijft:
    tempChannel.write(30);
  - **Timer**
    Je kunt een timer object gebruiken als een soort kookwekker. Na het aanroepen van zijn start() functie ga je normaal gesproken gelijktijdig ook op andere waitables wachten. Als je alleen op de timer zelf zou wachten, had je in het algemeen ook uitgekund met een eenvoudige hwlib::waitus() aanroep.
- **Pool**
  Een pool is een variabele die gewrapped is in het pool<> template.
  Neem bijvoorbeeld het object:
  pool<int> temperatuurPool
  Dat is een int, gewrapped in het pool<> template. Je kunt de int veilig lezen en schrijven door de read() en write() member functies van de pool<> template te gebruiken.
  Een pool is geen waitable. Elke taak kan er naar believen uit lezen of naartoe schrijven.
- **Mutexen**
  Als je voor RTOS designt, heb je veelal genoeg aan de bovengenoemde synchronisatiemechanismen. In een uitzonderlijk geval wil je misschien dat een reeks activiteiten altijd gegroepeerd moet worden uitgevoerd, dus dat het klaar moet zijn voordat een andere taak dezelfde reeks start. In dat geval kun je die reeks activiteiten beschermen met een mutex. Daar ga ik in dit document niet verder op in.

STDs ontwerpen

Objecten die code uitvoeren binnen een taak volgen meestal een toestandsdiagram (een STD).
In het algemeen zijn dat de <<control>> objecten. Maar ook veel boundary objecten die periodieke taken verrichten doorlopen ook een STD.

Hoe kom je vanuit een gegeven klassediagram tot de STDs?

Dat gaat in de volgende stappen:

- Maak de interface sectie van het STD
- Schrijf voor jezelf in woorden de reeks van acties en beslissingen op die horen bij de uitvoering van de taak van het betreffende object.
- Zet je beschrijving om in een STD:
    - Als je ergens op **wacht**, dan maak je daar een event van langs de uitgaande pijl van een toestand. Bedenk ook een passende naam voor de toestand. Op de timer events na zijn events altijd afkomstig uit de STD interface (zie verderop).
    - Bijvoorbeeld als je wacht tot de "aan-knop" is ingedrukt, maak dan een toestand met de naam "uit", en verlaat die via een uitgaande pijl met de event "flagAanButton".
    - Bedenk je of er actions zijn die altijd moeten gebeuren bij het binnengaan van de toestand (entry events), bij het verlaten van de toestand (exit events) of tijdens een event langs een uitgaande pijl.
      De acties kunnen voor een deel bestaan uit **aanroepen van publieke memberfuncties** van andere objecten. In het klassediagram kun je zien welke andere objecten je klasse kent en heeft, en dus welke publieke functies er aangeroepen kunnen worden.
    - Verwerk je **logische beslissingen** met behulp van guards en eventueel decision nodes.
    - **Voorkom duplicate actions en guards** eventueel met het gebruik van merge nodes.
    - Gebruik eventueel een **samengestelde toestand** (composite state) als het gedrag uit meerdere lagen bestaat.

Timers

Net als bij de overige typen waitables (flags, channels) – zie onderstaand – zouden er nu in het STD ook timer objecten kunnen worden gebruikt. Timers zijn een vreemde eend in de bijt. Wanneer gebruik je daar nu wel of niet een apart object voor **in je STD**? De vuistregel is: Benoem in je STD **geen** apart timer object als:

- Je timer **altijd wordt gestart in het entry-event** van een enkele, specifieke toestand, en de toestand **altijd verlaten wordt bij het vuren** van zijn timer-event

In zo'n geval kun je bijvoorbeeld after(1000ms) naast de uitgaande pijl zetten (dat impliceert dat de betreffende timer is gestart tijdens het entry-event).

In de overige gevallen geef je de timer een naam, en start je hem expliciet, met bijvoorbeeld myTimer.start(1000), en kan de timer-event eventueel afgehandeld worden door een event-handler in de body-sectie (het deel tussen entry en exit event) van de betreffende toestand. Voorbeeld: myTimer/doAction().

Ontwerp van de STD interface

- Het ontwerpen van STDs begint met het ontwerp van zijn interface sectie.
  Het STD communiceert met andere taken via de interface functies in zijn interface sectie.
- Die interface functies worden uitgevoerd in de thread van de aanroepende taak.

Je kunt de STD interface ontwerpen met de volgende vuistregels:

1. Voeg elke publieke functie van de klasse waarvoor je een STD maakt, en die wordt aangeroepen door een andere taak, toe aan de STD interface.
2. Je kunt de publieke functies in twee soorten verdelen:

- **Functies die “even snel” een waarde opvragen**:
  Bijvoorbeeld getTemperature() en isTemperatureTooHigh().
  Die functies implementeer je door een pool waarde te retourneren.
  Voorbeeld van zo’n functie:
  float getTemperature() {return temperaturePool.read();}
  De betreffende pool maak je een member van de klasse van het STD.
  De notatie van zo’n functie in de interface sectie is zoiets:
  getTemperature/return temperaturePool.read()
  De **types** van parameters en return values noteer je daar dus **niet**.
- **Overige functies** – in het algemeen functies die een bericht doorgeven of expliciet opdracht geven om iets te gaan doen. Die functies kun je ook weer in twee categorien verdelen:
  - **Functies zonder argument**
    Functies zonder argument, zoals bijvoorbeeld zetAan() implementeer je door een flag te zetten:
    zetAan/flagAan.set()
  - **Functies met argument**
    - **Als alleen de laatst binnengekomen waarde ertoe doet**
      Als alleen de laatst binnengekomen waarde ertoe doet, implementeer je zo’n functie met een **pool** en een **flag**. De waarde wordt in de pool geschreven en de flag wordt gezet om te signaleren dat er een nieuwe, nog niet verwerkte waarde is. Voorbeeld:
      buttonPressed(buttonID)/buttonIdPool.write(buttonID);buttonFlag.set()
    - **Als het belangrijk is dat je geen voorgaande waarde misloopt**
      Als het belangrijk is dat je geen voorgaande waarde misloopt, kun je in plaats van een flag+pool combinatie zoals hierboven beter kiezen voor een **channel**:
      buttonPressed(buttonID)/buttonChannel.write(buttonID)

Van STD naar Code

Het omzetten van een STD naar Code gaat vrij rechttoe rechtaan:

1. Creeer een klasse die is afgeleid van **rtos::task**
2. Creeer een **enum** waarmee de **toestanden** geduid kunnen worden.
3. Voeg de **member variabelen en references** toe zoals aangegeven in het klassediagram. (normaliter zijn die private).
4. Voeg een **constructor** toe die:
    a. De **reference members initialiseert**.
    b. Als de klasse een **listener** is, laat het **eigen object zich als listener toevoegen** aan de objecten waar het naar luistert.
    c. Voeg de members die een **Handler** nodig hebben toe aan de betreffende Handler.
5. Maak voor **elke STD interface** een gelijknamige **publieke functie** aan. De code kun je 1 op 1 overnemen.
6. Implementeer **de main() functie**:
    a. Creeer een **switch** statement dat springt naar de code voor de huidige toestand.
    b. Voer daar als eerste de **entry event** code uit.
    c. Vervolgens eventuele **"do"** code.
    d. **Wacht op** het optreden van **events**:
    auto evt = wait(een of meer waitables hier)
    e. Interpreteer de event, en laat de bijbehorende **guard, actie en/of transitie** uitvoeren:
    if(evt=flagButton){state = Idle;}

Timers

Net als bij de overige waitables (flags, channels) zouden er nu **in de code** timer objecten kunnen worden aangemaakt. Timers zijn een vreemde eend in de bijt. Wanneer maak je daar nu wel of niet een object voor aan in je code? De vuistregel is: als je **alleen** op de timer gaat wachten en ondertussen **niets** anders doet, hoef je geen apart timer object aan te maken. Je gebruikt dan in de code bijvoorbeeld een hwlib::waitus(1000) call. De code gaat pas verder na het verstrijken van die tijd.

Je maakt wel expliciet een timer object aan bij "time-out" situaties. Het aparte timer object is dan als een kookwekker die na enige tijd een event kan genereren. In de code kan dan worden gewacht op **zowel** die timer als op andere events. Daardoor kan het object tot de timer afgaat door blijven gaan met andere dingen.

Enum prefixes

Omwille van overzicht is het in het bovenstaande niet gedaan, maar omwille van het voorkomen van nameclashes is het een betere gewoonte om enum variabelenamen te prefixen met de enum type naam.

STD naar Code bij meer geavanceerde varianten

Je hebt nu voorbeelden gezien voor het omzetten van STD naar code bij het Trilapparaat.
Er zijn echter situaties die extra aandacht vergen:

- Een entry event in een state die nested substates heeft.
- Een action bij de uitgaande pijl van een state die nested substates heeft.
- Een exit event in een state die nested substates heeft.
- Een event bij de uitgaande pijl van een state die nested substates heeft.

Een entry event in een state die nested substates heeft

Kijk bijvoorbeeld nog eens naar de huidige code van TrilControl:

```
for(;;)
{
    switch(state)
    {
        case NORMAAL:
            switch(substateNormaal)
            {
                case NIET_TRILLEN:
                    auto evt = wait(bnReleaseChannel,bnPressChannel);
                    if(evt==bnPressChannel)
                    {
                        bnID = bnPressChannel.read();
                        if(nbID==NoodKnopID)
                        {
                            state = NOODTOESTAND;
                        }
                        else // bnID moet TrilKnopID zijn
                        {
                            substateNormaal = TRILLEN;
                        }
```

Stel nu dat de onderste else case wordt uitgevoerd: substateNormaal wordt gezet op TRILLEN. Dan is de case afgehandeld, en komen we weer terug bij het begin van de for(;;) loop, bij het bovenste switch statement. Aangezien de hoofdtoestand nog steeds NORMAAL is, gaat het programma vervolgens CASE NORMAAL in.

So far, geen probleem. Maar wat als NORMAAL entry event code zou hebben gehad, waar zetten we dat dan neer? Als we de entry event code simpelweg onder CASE NORMAAL: neerzetten, wordt die code steeds opnieuw uitgevoerd als er een substate verandert (zoals in bovenstaand voorbeeld). Maar entry event code van de state NORMAAL mag natuurlijk alleen uitgevoerd worden bij het binnengaan van NORMAAL vanuit een andere (main-) state.

Je zou in het algemeen dit op de volgende manier kunnen oplossen:

```
for(;;)
{
    switch(state)
    {
        case NORMAAL:
            if (prevState != NORMAAL)
            {
              // begin entry event stuff
              // …
              // end of entry event stuff
              prevState = NORMAAL;
            }
            switch(substateNormaal)
```

```
{
    case NIET_TRILLEN:
```

Een action bij de uitgaande pijl van een state die nested substates heeft, of een exit-event van een state die nested substates heeft

In het STD van TrilControl kun je zien dat er naast de **uitgaande pijl van de toestand NORMAAL** geen event staat, en ook geen actions. Stel nu dat er een **action** zou hebben gestaan, dan zou die action uitgevoerd moeten worden door **elk van de substates** op het event dat van hen naar de exit node leidt. Hetzelfde geldt voor het geval dat de toestand NORMAAL exit event code had gehad. Ook die zou moeten worden uitgevoerd door **elk van de substates** op het event dat van hen naar de exit node leidt.

Een event bij de uitgaande pijl van een state die nested substates heeft.

In het STD van TrilControl is ervoor gekozen om de toestand NORMAAL te verlaten via een exit-node, in combinatie met een uitgaande pijl zonder event.

Alternatief had ervoor gekozen kunnen worden om de toestand NORMAAL te verlaten via een pijl naar Noodtoestand met naast die pijl een event “noodButtonPressedFlag” waarop wordt gewacht. Je kunt niet wait(noodButtonPressedFlag) onder case NORMAAL: zetten, omdat dan alle substates eronder worden geblokkeerd. De oplossing is **elk van de substates (ook) op noodButtonPressedFlag te laten wachten.**

Concurrency Diagram (dit hoofstuk/onderwerp vervalt/mag je skippen vanaf 2021)

Het concurrency diagram geeft weer via welke synchronisatiemechanismen te taken met elkaar communiceren.

Ik heb er in dit document voor gekozen om de keuze van de synchronisatiemechanismen te bepalen op het moment dat de interface voor de std moet worden ontworpen. In dit geval kan het concurrencydiagram dus helemaal achteraf worden getekend en is het niet nodig geweest tijdens het design proces.

Taken en synchronisatiemechanismen

Zo maak je een concurrency diagram:

1. Noteer de namen van de taken die uit de taakstructurering+taaksamenvoeging kwamen in parallellogram-vormen.

2. Kijk in het klassediagram welke functies van een taak door een andere taak worden aangeroepen. Bedenk voor die functies welke syncronisatiemechanismen het best gekozen kunnen worden.

   In het voorgaande hadden we al gezien hoe je dat doet:
   - **Functies die “even snel” een waarde opvragen**:
     Bijvoorbeeld getTemperature() en isTemperatureTooHigh().
     Die functies implementeer je door een pool waarde te retourneren.

- **Overige functies** – in het algemeen functies die een bericht doorgeven of expliciet opdracht geven om iets te gaan doen. Die functies kun je ook weer in twee categorien verdelen:
  - **Functies zonder argument**
    Functies zonder argument, zoals bijvoorbeeld zetAan() implementeer je door een flag te zetten.
  - **Functies met argument**
    - **Als alleen de binnengekomen waarde ertoe doet**
      Als alleen de laatst binnengekomen waarde ertoe doet, implementeer je zo'n functie met een pool en een flag. De waarde wordt in de pool geschreven en de flag wordt gezet om te signaleren dat er een nieuwe, nog niet verwerkte waarde is.
    - **Als het belangrijk is dat je geen voorgaande waarde misloopt**
      Als het belangrijk is dat je geen voorgaande waarde misloopt, kun je in plaats van een flag+pool combinatie zoals hierboven beter kiezen voor een channel.

3. Ga na welke klassen op een timer wachten, en voeg dat toe aan het diagram.
4. Ga na of er entity klassen zijn die vanuit verschillende taken gelezen en geschreven kunnen worden.
   Ga na wie de setters en getters van de betreffende entity aanroepen. De betreffende data moet geencapsuleerd zijn in in pools.

That's all. Alleen even opletten op de volgende valkuil:

- Op een waitable (flag, timer, channel) mag alleen gewacht worden door die ene taak die hem bezit.
- Alle pijlen wijzen naar de synchronisatie-elementen toe (dus niet met de informatiestroom mee, wat je intuitief misschien zou doen).

Wrapup

Dit waren alle stappen van een ontwerp van kop tot staart:

1. Object model met object beschrijvingen
2. Taakstructurering
3. Klassediagram
4. Stds
5. Code
6. Concurrency Diagram

Een belangrijke volgende stap is oefenen, oefenen en nog eens oefenen. Het zijn steeds dezelfde, eigenlijk wel logische stappen.

Als je het een beetje in de vingers hebt, kun je een ontwerp als dit in een stief ochtendje van requirements naar code + goede documentatie brengen.