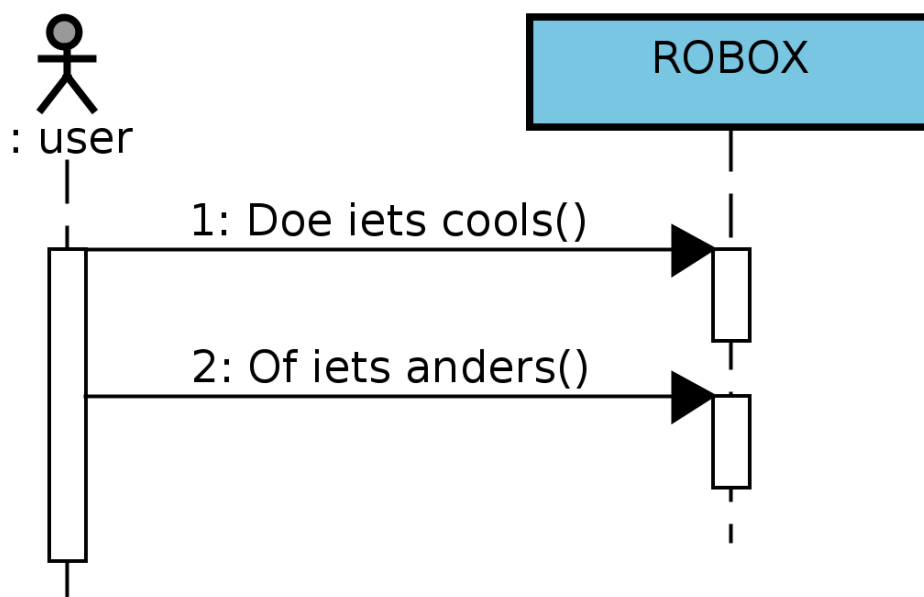


ROBOX - Firmware

Software design description



Auteur	Luke van Luijn	Minor	Digital Media Productions (DMP)
Student nummer	587478	Docentbegeleider	Mario de Vries
Opleiding	HBO-ICT	Plaats	Nijmegen
Profiel	Embedded Software Development (ESD)	Datum	27-05-2022
Studiejaar	Jaar 3	Versie	1.0

Inhoudsopgaven

- 1 [Termen](#)
- 2 [Introductie](#)
 - 2.1 [Doel en domein](#)
 - 2.2 [Doelgroep](#)
 - 2.3 [Doel van het document](#)
- 3 [Architectonisch overzicht](#)
- 4 [Detailed design description](#)
 - 4.1 [Package - Driver](#)
 - 4.1.1 [Ontwerp keuzes](#)
 - 4.2 [Package - Logger](#)
 - 4.2.1 [Ontwerp keuzes](#)
 - 4.3 [Package - Widget](#)
 - 4.3.1 [Ontwerp keuzes](#)
 - 4.4 [Package - Frame](#)
 - 4.4.1 [Ontwerp keuzes](#)
 - 4.5 [Package - Base](#)
 - 4.6 [Package - Utils](#)
- 5 [Literatuurlijst](#)

1. Termen

Index	Term	Beschrijving
00	ROBOX/robot	Hiermee wordt de firmware en het daadwerkelijke apparaat bedoelt.
01	UML	<i>Unified modeling language</i> de syntax gebruikt voor het opstellen van de verschillende diagrammen.
02	product	De uiteindelijke applicatie, het resultaat beschreven in dit document.
03	package/namespace	Een groepering van software componenten die een soortgelijk doel nastreven.
05	parsen	Het uitlezen en interpreteren van data, meestal een string zodat de correcte waardes eruit gehaald kunnen worden.
06	seriële bus	Een veelgebruikt, maar ouderwets, protocol waarmee verschillende apparaten, bijvoorbeeld via USB, data kunnen uitwisselen (Wikipedia-bijdragers, 2022).
07	thread-safe	Een onderdeel is thread-safe wanneer er geen onverwachte dingen gebeuren wanneer dit onderdeel vanaf een andere thread dan de main thread afgesproken wordt. Bijvoorbeeld bij het schrijven naar de console kan het zo zijn dat wanneer twee threads dit tegelijkertijd doen de tekst door elkaar heen geschreven wordt. Een thread-safe onderdeel vangt dit af.
08	severity	De graad van een bericht. ERROR > WARNING > INFO > DEBUG.
09	tooltip	Het kleine schermpje wat soms verschijnt als je met de muis over een schermonderdeel zweeft.
10	calls	Het aanroepen van een methode.
11	multithreaded	Een applicatie die gebruik maakt van twee of meer threads.
12	scherm onderdeel	Een onderdeel van de applicatie die zichtbaar is op het scherm, bijvoorbeeld een tekstveld of knop.

2. Introductie

In dit document zal het ontwerp van de software beschreven worden. Eerst zal er gekeken worden naar de gehele applicatie in het hoofdstuk 'Architectural overview'. Vervolgens in het hoofdstuk 'Detailed design description' zal er per onderdeel een diepere toelichting gegeven worden op de functie, werking en eventuele beslissingen die genomen zijn om tot het eind resultaat te komen. Het laatste hoofdstuk 'Interface' bevat een toelichting op de ontwikkelde Grafische user interface.

2.1. Doel en domein

Dit onderdeel van het *ROBOX* project is bedoeld als fundering voor verdere ontwikkeling. Het is een basis opzet die gebruik maakt van de verschillende functionaliteiten van de *ROBOX* firmware. De software is opgezet met het idee dat toekomstige ontwikkelaars er zonder te veel moeite nieuwe elementen en functionaliteiten toe kunnen voegen.

2.2. Doelgroep

Het software design description document is geschreven voor de projectbegeleiding en eventueel andere (externe) belangstellende. In dit document wordt aangenomen dat de lezer een basis kennis van *UML* en softwareontwikkeling heeft. Verder is het aangeraden om het Software requirements document van beide de firmware en de software door te nemen, ook het Software design description document van de firmware kan verheldering bieden tijdens het lezen van dit document.

2.3. Doel van het document

Het doel van dit document is om een duidelijk beeld te creëren voor de belangstellende over het ontwikkelde *product*. Met name de keuzes die gemaakt zijn tijdens de ontwikkeling en de opzet van het product zelf. Hoe het in de toekomst gebruikt en uitgebreid kan worden en wat er mee kan.

3. Architectonisch overzicht

De applicatie is onderverdeeld in verschillende *packages*, ook wel *namespaces*. De verschillende klassen zijn onderverdeeld in deze namespaces op basis van de functionaliteit die ze vertonen. De Driver namespace is bijvoorbeeld verantwoordelijk voor de communicatie met de *robot* zelf.

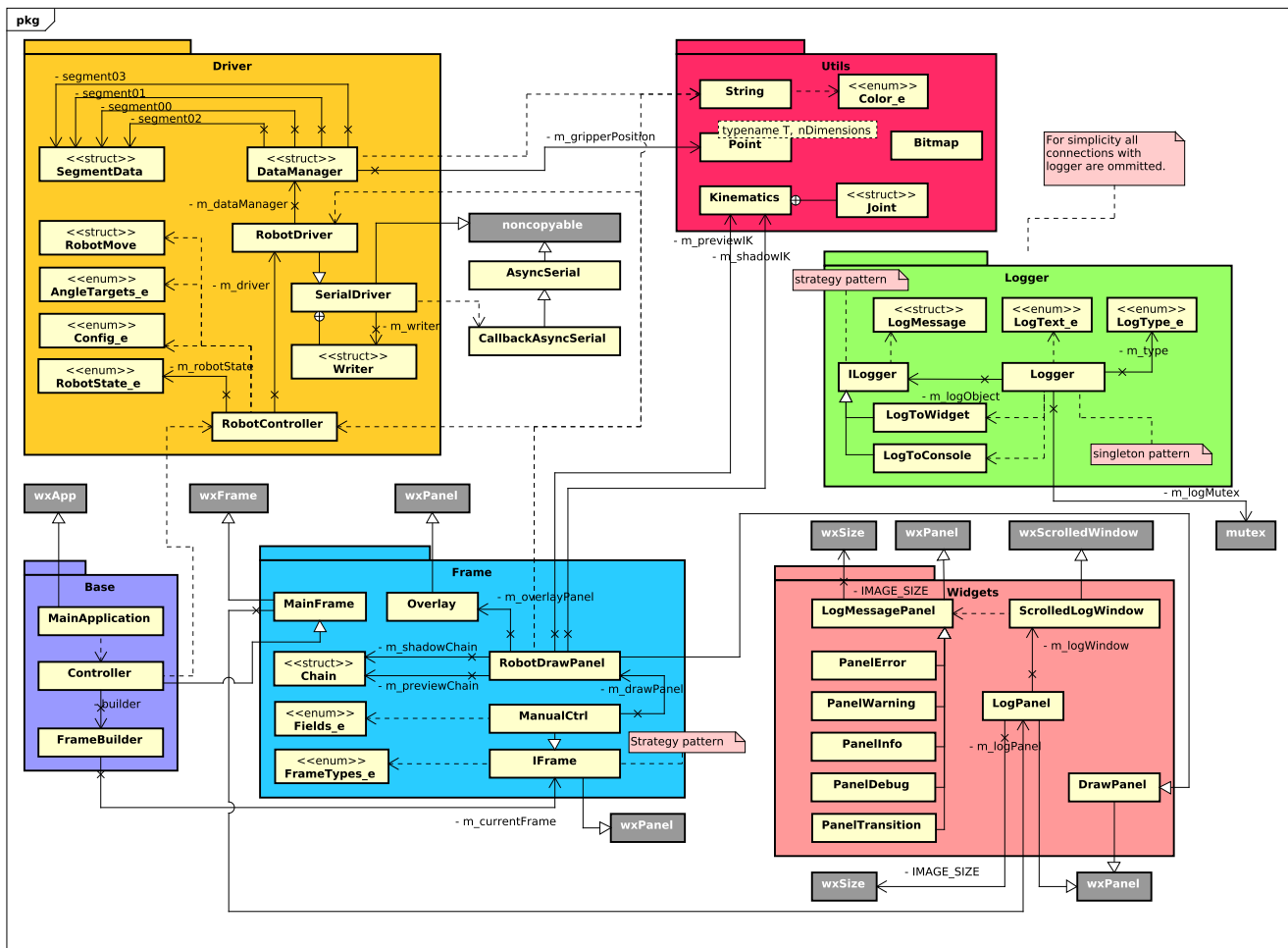


Diagram 1 - Design class diagram - Software applicatie

De softwareapplicatie is geschreven zodat het makkelijk uitgebreid kan worden met nieuwe functionaliteit. De GUI is opgezet zodat het uitgebreid kan worden met nieuwe *Control methods*. Momenteel bevat de applicatie een enkele control method; **ManualCtrl**, hierover meer in het hoofdstuk 'Package - Frame'. In toekomstige iteraties kan dit bijvoorbeeld uitgebreid worden met een **KeyboardCtrl** of een **BLTCtrl** klasse.

De applicatie is ook ingericht om te communiceren met meerdere microcontrollers tegelijkertijd. Het zou bijvoorbeeld goed kunnen dat de robot gebruikt zal worden in combinatie met andere apparaten. Dit zou dus allemaal in deze applicatie verwerkt kunnen worden.

4. Detailed design description

In dit hoofdstuk worden de verschillende packages en de beoogde functionaliteit dieper toegelicht. Voor elke package zal er een beschrijving zijn voor de verschillende klasse en de werking. Bovendien zal elke package voorzien zijn van een beschrijving welke keuzes er gemaakt zijn tijdens de implementatie van dat onderdeel.

4.1. Package - Driver

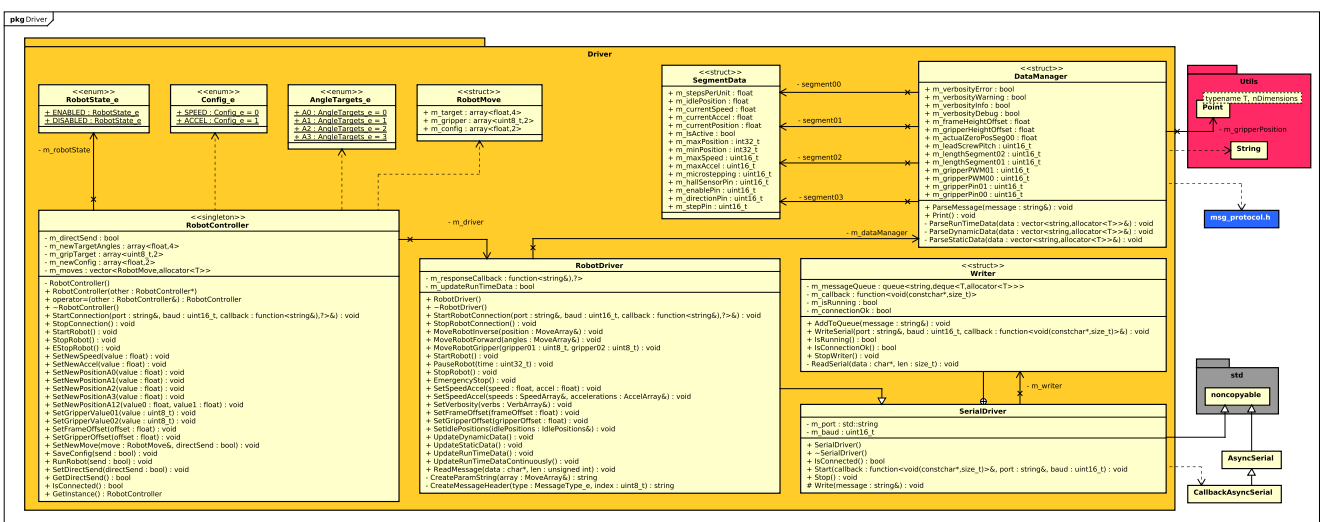


Diagram 2 - Class diagram - Driver package

De package **Driver** heeft als doel het realiseren van een stabiele connectie met de robot over de *seriële bus*. Deze connectie wordt gebruikt voor het versturen en ontvangen van berichten. Door middel van deze connectie is het mogelijk om commando's te versturen en het opvragen van data.

De klasse **SerialDriver** is verantwoordelijk voor het daadwerkelijk opstarten, uitlezen en afsluiten van de seriële connectie. De SerialDriver klasse maakt gebruik van de lokale library **AsyncSerial**. Deze library is geschreven door **Terraneo Federico**. De library heeft als doel het asynchroon schrijven en lezen naar en van de seriële bus. De SerialDriver klasse maakt gebruik van een thread, **Writer**, die continue een queue naar de seriële bus probeert te schrijven. Deze queue kan van buiten de thread gevuld worden zodat de communicatie, indien actief, altijd aanstaat. Het uitlezen van de seriële bus gebeurt door middel van een callback, zodra er nieuwe data beschikbaar is zal de callback aangeroepen worden en kan deze data ge-*parsed* en verwerkt worden in de applicatie.

Doordat de SerialDriver klasse een losse klasse is kan er een nieuwe driver voor bijvoorbeeld een extra microcontroller geschreven worden zodat de applicatie gebruik kan maken van meerdere externe microcontrollers.

TODO linkje

De klasse **RobotDriver** is verantwoordelijk voor het uitwerken van alle mogelijke functionaliteiten die de *firmware* te bieden heeft. Zoals te zien in de afbeelding zijn alle methodes beschreven in het *message protocol* ondersteund. De hoofdfunctionaliteit van de RobotDriver klasse is het uitlezen en versturen van commando's over de seriële bus, het is dan ook om die reden dan de RobotDriver erft van de SerialDriver klasse. De eerder genoemde callback voor het verwerken van binnenkomende berichten is ook verwerkt in de RobotDriver. De RobotDriver heeft een statische instantie van de klasse **DataManager**. De DataManager is verantwoordelijk voor het parsen van de binnenkomende berichten en het bijhouden van de verschillende datavelden (zie *message*

protocol). Op het moment van schrijven zijn er drie data request commando's GET_STATIC_DATA, GET_DYNAMIC_DATA & GET_RUNTIME_DATA. De verschillende velden verkregen uit deze drie commando's zijn verwerkt in de DataManager klasse.

De klasse **RobotController** is het aanspreekpunt voor de applicatie qua communicatie met de robot. Deze klasse is geïmplementeerd volgens het **singleton pattern**. Dit houdt in dat er altijd maar een enkele instantie van deze klasse in de applicatie aanwezig is. Door dit design pattern kun je zeker zijn dat de resources die deze klasse bevat altijd maar door een instantie aangesproken wordt. Gezien het niet handig is als de robot meerdere commando's binnenkrijgt als dat niet de bedoeling is is een singleton pattern voor deze klasse een geschikte keus.

De RobotController klasse fungeert ook als een **facade pattern**, het verbergt de complexe functionaliteit van de SerialDriver en RobotDriver.

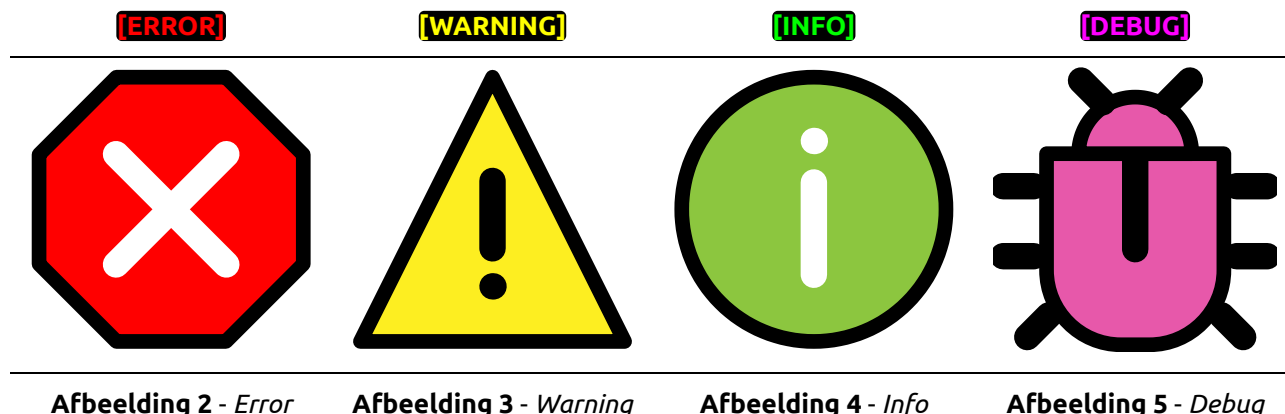
De RobotController implementeert de functionaliteiten van de firmware die nodig zijn voor de werking van de applicatie. Het levert de mogelijkheid om verschillende posities van de robot op te slaan en op een later tijdstip samen te voegen en te versturen naar de robot. Ook is het mogelijk om een nieuwe positie direct te versturen als hier vraag naar is.

4.1.1. Ontwerp keuzes

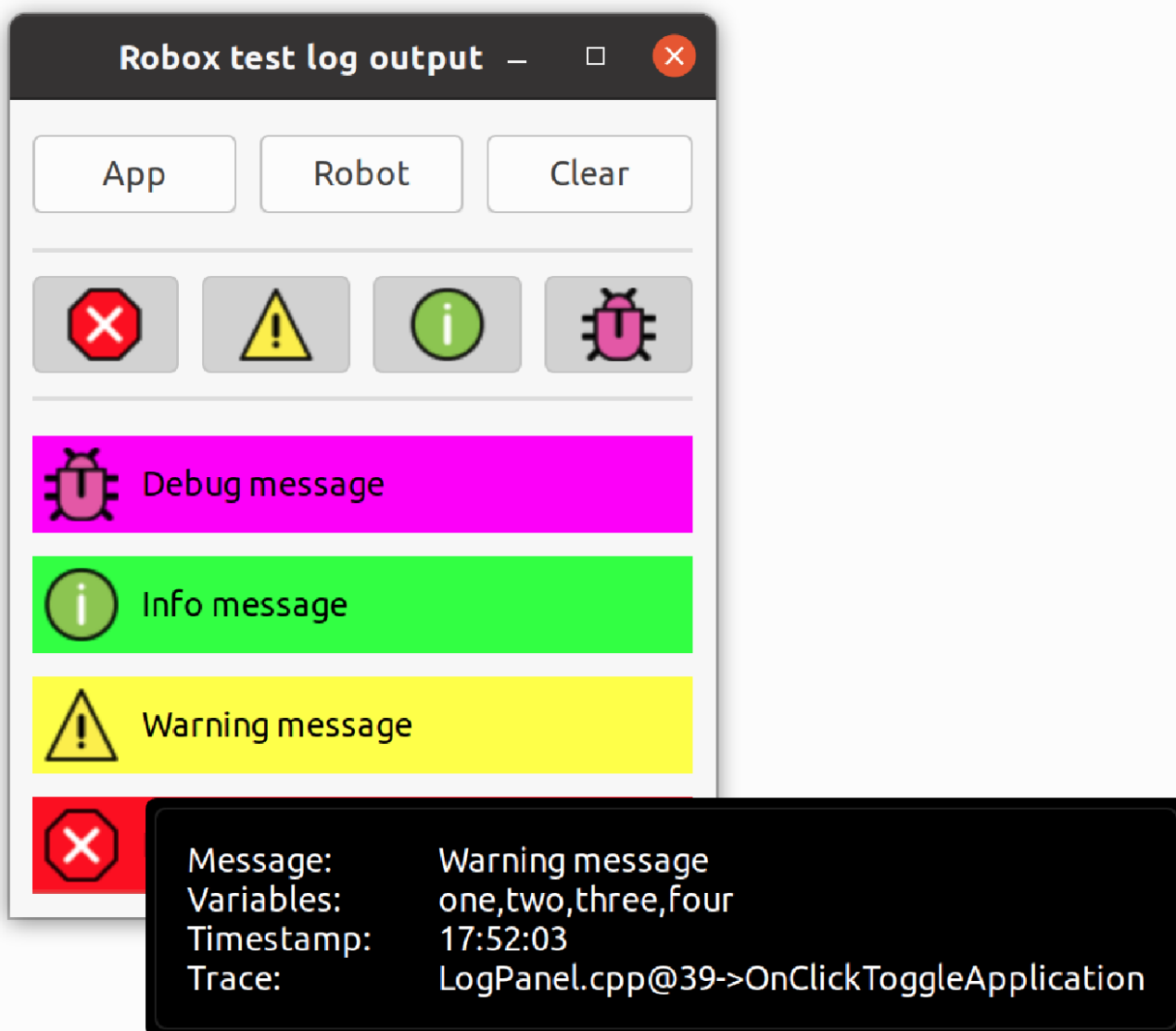
#	Probleem	Besluit	Argumenten
OK-01	Het parsen van de data moet op een centrale locatie plaatsvinden en toegankelijk zijn voor de gehele applicatie.	Het opzetten van een statische data container die zelf de berichten parsed.	Door dat de DataManager een statische instantie is is er altijd, maar een in de applicatie, doordat de DataManager zelf verantwoordelijk is voor het parsen van de berichten weet je zeker dat op dezelfde plek gebeurt.
OK-02	In toekomstige iteraties van de applicatie is het misschien noodzakelijk om meerdere seriële apparaten aan te sluiten.	Het loskoppelen van de functionaliteit van het uitlezen van de seriële bus.	Door de functionaliteit los te koppelen van de RobotDriver is het mogelijk om meerdere drivers op te zetten die gebruik maken van de seriële bus.
OK-03	Hoe verzekeren we dat de applicatie niet meerdere keren (foutieve) berichten naar de robot stuurt.	Het opzetten van een combinatie van een facade en singleton pattern.	Het facade pattern verstoort de achterliggende functionaliteit van de drivers. Het singleton pattern verzekert dat er altijd maar een locatie gebruik maakt van de RobotController.

Tabel 1 - Ontwerp keuzes - Driver

correspondeert met de severity van het bericht, zie bovenstaand. Verder wordt er een afbeelding toegevoegd, deze afbeeldingen zijn hieronder weergegeven.



Het daadwerkelijke log bericht wordt weergegeven op het paneel zelf. De andere aspecten zoals variabelen, tijd en locatie wordt in een *tooltip* verwerkt zodat het overzichtelijk blijft. In de onderstaande afbeelding zijn enkele voorbeelden te zien.



Afbeelding 6 - Voorbeeld van een widget-log

De klasse **Logger** is verantwoordelijk voor het daadwerkelijk verwerken van de log berichten. Omdat er vanaf allerlei plekken in de applicatie gelogd kan of moet worden, ook vanaf andere threads, is het noodzakelijk dat de logger een thread-safe, singleton klasse is. Dit houdt in dat de logger van welke thread dan ook aangesproken kan worden en nooit log berichten door elkaar naar de console of naar een widget schrijft. De Logger klasse implementeert drie verschillende design patterns; singleton, facade en strategy.

Eerder is al vermeld dat een log bericht uit meerdere elementen bestaat, het aanroepen van een log bericht, zeker tijdens het testen, is dan een moeizame actie. Om dit proces te versimpelen is er gebruik gemaakt van verschillende **macro's**. Deze macro's kunnen aangeroepen worden met een `std::string` waarde of een `std::string` waarde en een vector van een onbepaald type. De macro's roepen vervolgens de correcte methode aan het de bijhorende elementen.

Zonder macro	<code>Logger::Logger::GetInstance().Error("message", vec, __func__, __FILE__, __LINE__);</code>
Met macro	<code>ERROR("message", vars);</code>

De Logger klasse levert ook ondersteuning om verschillende berichten te filteren, een gebruiker kan bijvoorbeeld **DEBUG** berichten uitzetten wanneer de applicatie gereed is voor gebruik. Op deze manier hoeven niet alle debug *calls* weggehaald worden en is debuggen in een later stadium des te makkelijker.

4.2.1. Ontwerp keuzes

#	Probleem	Besluit	Argumenten
OK-01	Hoe kun je betrouwbaar loggen in een <i>multithreaded</i> applicatie?	Door een thread-safe singleton logger klasse te maken.	Het besluit zegt het al, een thread-safe singleton heeft als doel betrouwbaar gebruik in een multithreaded applicatie.
OK-02	Hoe kun je makkelijk wisselen tussen loggen naar de console en loggen naar bijvoorbeeld een bestand?	Door een logger op te zetten volgens een strategy pattern.	Door middel van een strategy pattern kan er makkelijk, zelfs run time, gewisseld worden van strategy.
OK-03	Hoe laat je duidelijk zien hoe belangrijk een bericht is voor de gebruiker?	Het toepassen van afbeeldingen en kleur.	Door opvallende afbeeldingen en kleuren te gebruiken die gebruikers makkelijk herkennen is het snel duidelijk hoe een bericht geïnterpreteerd moet worden.
OK-04	Hoe kun je de focus op een onderdeel van een bericht leggen zonder informatie weg te laten?	Door de 'onbelangrijke' data in een donkere kleur af te beelden, de belangrijke data kan vervolgens in een contrasterende kleur afgebeeld worden.	Door bijvoorbeeld de tijd en locatie onderdelen van een log bericht in het grijs af te beelden en de severity in kleur en het bericht in het wit is het meteen duidelijk wat de essentie van het bericht is.

#	Probleem	Besluit	Argumenten
OK-05	Hoe kun je makkelijk gebruik maken van een logger, maar toch de verschillende data velden vullen?	Door gebruik te maken van verschillende macro's.	Door gebruik te maken van macro's kunnen de data velden gevuld worden zonder dat de gebruiker de complete aanroep hoeft uit te typen.

Tabel 2 - Ontwerp keuzes - Logger

4.3. Package - Widget

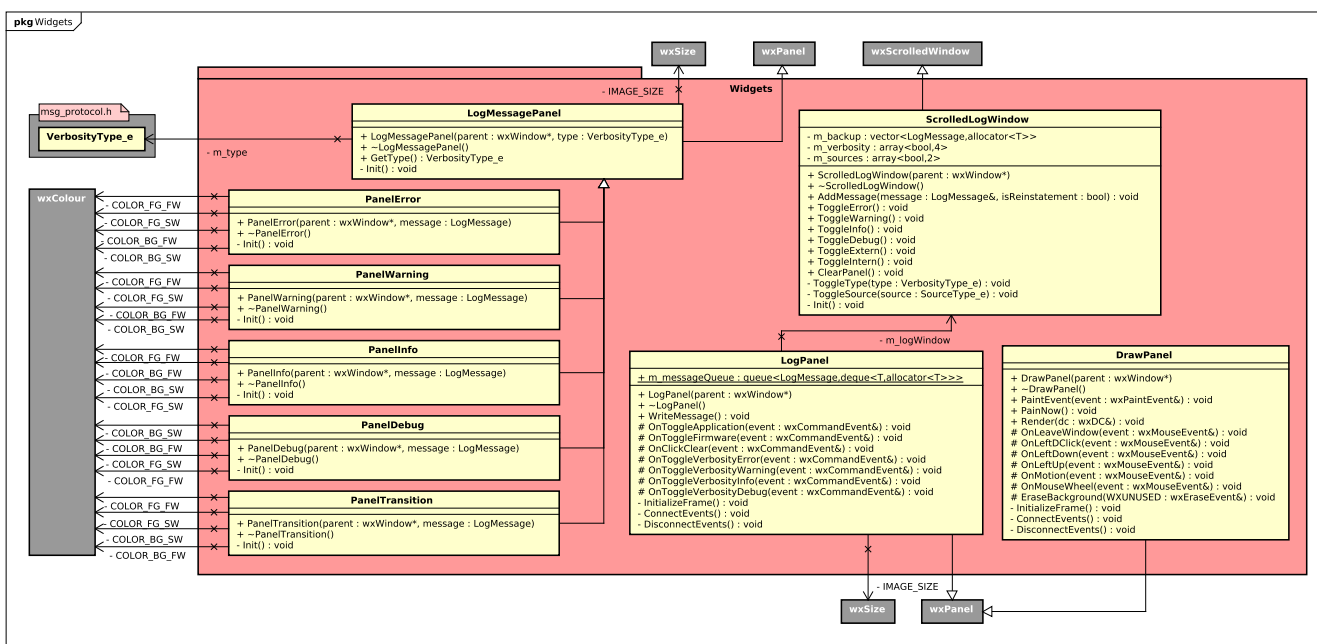
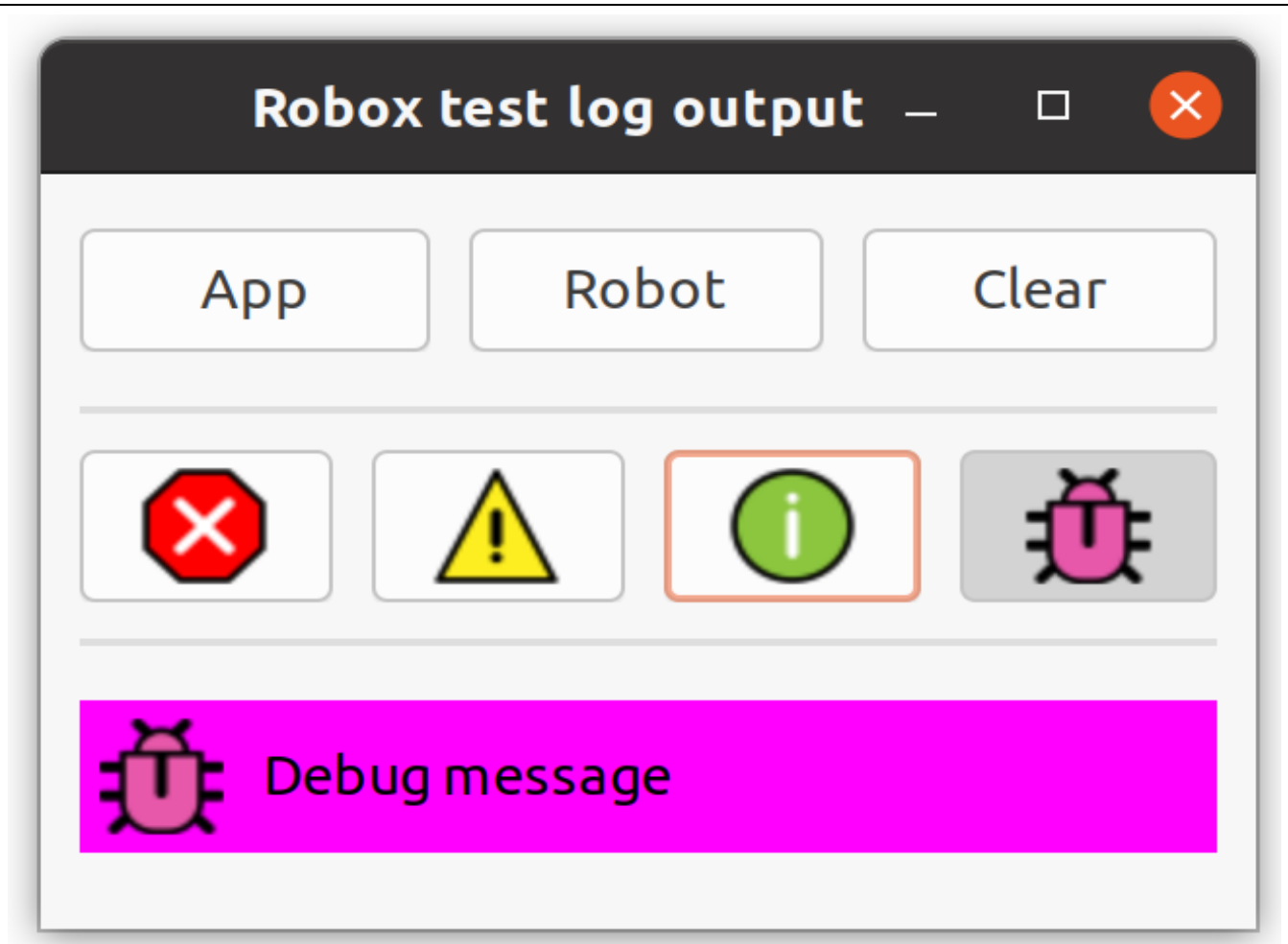






Diagram 4 - Class diagram - Widget package

De **Widget** package is een verzameling van verschillende klasse die fungeren als **wxWidgets** scherm onderdeel.

De klasse **LogMessagePanel** is de basis klasse voor een log message, de klassen **PanelError**, **PanelWarning**, **PanelInfo**, **PanelDebug** en **PanelTransition** zijn de daadwerkelijke implementaties van de log panels. Deze implementaties zetten de kleur, afbeelding en tooltip corresponderend met het log bericht. De klasse **ScrolledLogWindow** is de container die de verschillende log panels bevat. Deze klasse houdt een lijst met log berichten bij, wanneer er bijvoorbeeld gefilterd moet worden op een bepaald soort bericht kan deze klasse enkel de berichten laten zien die gevraagd zijn door de gebruiker. De klasse **LogPanel** is het onderdeel dat uiteindelijk op het scherm terecht komt. De klasse bevat een **ScrolledLogWindow** die weer de log panels bevat. De **LogPanel** klasse bevat enkele knoppen, in de onderstaande afbeelding en tabel worden deze onderdelen dieper toegelicht.



Afbeelding 7 - Log panel


'App'-knop	Wanneer de gebruiker op deze toggle knop drukt zullen all berichten afkomstig uit de applicatie uit de lijst met log berichten gefilterd worden.
'Robot'-knop	Wanneer de gebruiker op deze toggle knop druk zullen alle berichten afkomstig van een externe applicatie, de robot, uit de lijst met log berichten gefilterd worden.
'Clear'-knop	Door op deze knop te drukken zullen alle log berichten permanent verwijderd worden.
	De error toggle knop filtert alle error berichten uit de lijst. Bij het nogmaals klikken van deze knop komen de berichten weer terug.
	De warning toggle knop filtert alle warning berichten uit de lijst. Bij het nogmaals klikken van deze knop komen de berichten weer terug.
	De info toggle knop filtert alle info berichten uit de lijst. Bij het nogmaals klikken van deze knop komen de berichten weer terug.
	De debug toggle knop filtert alle debug berichten uit de lijst. Bij het nogmaals klikken van deze knop komen de berichten weer terug.



De klasse **DrawPanel** is een variant van de door wxWidgets aangeleverde '**BasicDrawPane**', deze klasse levert de mogelijkheid om op een paneel te tekenen met simpele vormen. Dit paneel wordt automatisch geüpdatet door de wxWidgets pipeline, waardoor het uitermate geschikt is voor bijvoorbeeld een simulatie.

4.3.1. Ontwerp keuzes





#	Probleem	Besluit	Argumenten
OK-01	Hoe zorg je dat er meerdere panelen weergegeven kunnen worden zonder de applicatie te schalen?	Door gebruik te maken van een scrolledWindow.	Een scrolled window is een wxWidgets klasse die wanneer de onderdelen in het window groter zijn dan het daadwerkelijke window er een scroll balk ontstaat.
OK-02	Hoe kun je een dynamische afbeelding weergeven in een wxWidgets paneel?	Door een klasse op te zetten die gebruik maakt van wxDC, bijvoorbeeld een variant van de BasicDrawPane.	Door een DrawPanel te implementeren is het mogelijk om een dynamische tekening, bijvoorbeeld een simulatie, weer te geven die automatisch geüpdatet wordt door de wxWidgets pipeline.
OK-03	Hoe kun je duidelijk onderscheid maken tussen interne en externe log berichten?	Door kleur codering te gebruiken bij het weergeven van log berichten	Door de kleur van interne berichten donkerder te zetten, maar wel dezelfde ondertoon is er een duidelijk verschil te zien tussen berichten van de applicatie en berichten van de externe machine.
OK-04	Hoe zorg je ervoor dat er geen overflow situatie ontstaat door alle log berichten?	Door een limit in te stellen voor het maximum aantal actieve berichten in de buffer.	Door een limit in te stellen, in dit geval 25, worden oudere minder belangrijke berichten niet meer weergegeven en is het dus niet mogelijk om een overflow situatie te creëren.

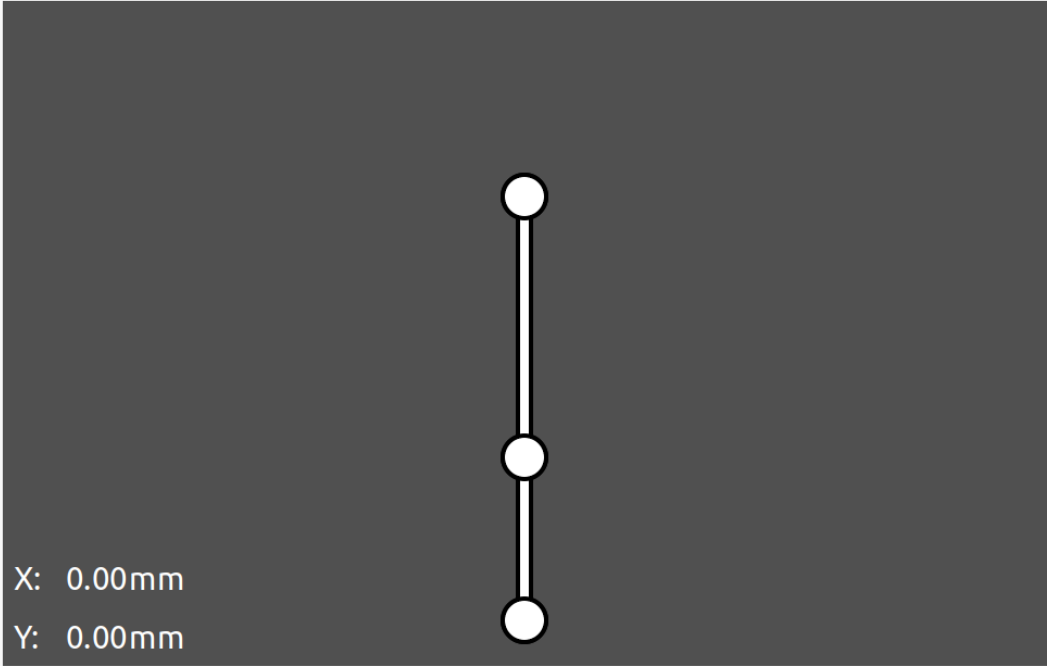
Tabel 3 - Ontwerp keuzes - Widget

Manual
Keyboard
BKE-ctrl
Settings
About
Connect
Activate
Run
STOP


Speed: % 
Accel: % 

Gripper 01
Gripper 02
Automatic run

Z: mm 
A1: ° 
A2: ° 
A3: ° 



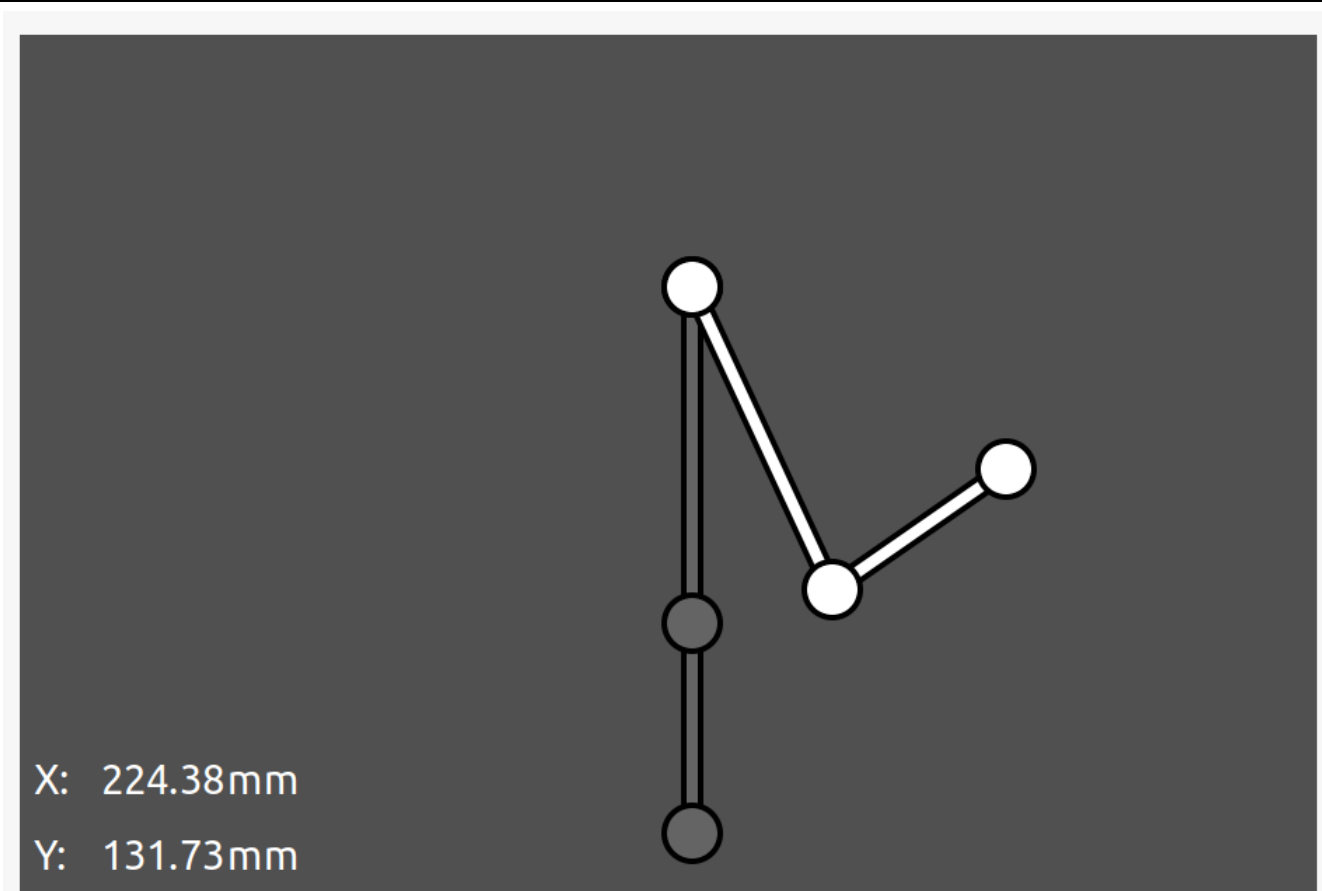
X: 0.00mm
Y: 0.00mm

Afbeelding 8 - Manual control screenshot

Zoals te zien in de afbeelding bevat deze uitwerking van de control method veel sliders, tekstvelden, knoppen en een simulatie. De onderdelen 'Speed' en 'Accel' zijn verantwoordelijk voor het weergeven en aanpassen van de huidige snelheid van de robot. Deze velden zijn werken volgens een percentage van het maximum aangegeven door de robot. Zodra de slider bewogen worden zal het tekstveld actief geüpdatet worden. Zodra in het tekstveld een nieuwe waarde wordt ingevoerd wordt de slider geüpdatet op het moment dat er op 'enter' gedrukt wordt.

De knoppen 'Gripper 01' en 'Gripper 02' zijn verantwoordelijk voor het aan en uit zetten van de gripper signaal pinnen. Hoewel de gripper een PWM-sigitaal ondersteund is er voor de uitwerking van manualCtrl gekozen voor een alles-of-niets uitwerking (aan/uit). De knop 'Automatic run' kan gebruikt worden als overschrijving van de eerder besproken 'Run'-knop. Als deze knop actief is hoeft de gebruiker niet telkens op run te drukken als de positie van de robot veranderd wordt. Als er nu een slider of tekstveld aangepast wordt zal er direct een bericht naar de robot gestuurd worden waardoor de robot direct naar de nieuwe positie verplaatst.

De onderdelen met de titels; 'Z', 'A1', 'A2' en 'A3' controleren de positie van de respectievelijke assen. Deze velden werken op dezelfde manier als de 'Speed' en 'Accel' velden. Het enige verschil is dat deze velden werken op basis van de daadwerkelijk door de robot opgegeven minimums en maximums. De minimums en maximums worden ingesteld op basis van de dynamische data verkregen van de robot (Zie Package - Driver).



Afbeelding 9 - *Simulation screenshot*

Het grijze blok onderin het scherm van de simulatie. De simulatie is opgezet door de klasse **RobotDrawPanel**. De RobotDrawPanel klasse erft van de DrawPanel klasse (zie Package - Widgets) en heeft daarom de mogelijkheid om een dynamische tekening weer te geven, in dit geval het bovenaanzicht van de robot. In de simulatie staan de cirkels voor de verschillende joints van de robot en de lijnen representeren de links tussen de joints. De tekening is een geschaalde variant van de robot. Door op de laatste joint te klikken met de muis (de onderste joint), kan deze geselecteerd worden. Wanneer dit gebeurt, zal er een rode rand om deze joint verschijnen en beweegt de joint met de muis mee. De rest van de joints en links wordt vervolgens geüpdatet volgens inverse kinematica (zie Package - Utils). Wanneer de robot bewogen wordt in de simulatie zullen de eerder genoemde velden 'A1' en 'A2' ook realtime geüpdatet worden, voor het updaten van hoeken op basis van een cartesische positie wordt gebruik gemaakt van forwards kinematica. Zodra de joint in de simulatie losgelaten wordt zal de positie opgeslagen worden net zoals het zou gebeuren voor de velden 'A1' en 'A2'.

De grijze variant van de robot representeert de huidige positie van de daadwerkelijke robot. Deze simulatie wordt geüpdatet aan de hand van de runtime data verstuurd door de robot (zie Package - Driver). Zodra de witte robot verplaatst wordt of door middel van de sliders/tekstvelden of door de simulatie en er wordt op run gedrukt zal de grijze robot naar de posities van de witte robot verplaatsen en zichzelf 'verstoppen' onder de witte robot. Hierdoor is het snel duidelijk wat de robot precies uitvoert zonder naar de fysieke robot te kijken. Links onderin het grijze blok zijn twee velden te zien; 'X' en 'Y'. Deze velden representeren de huidige positie van de gripper (de laatste joint) in cartesische coördinaten. Ook deze velden worden realtime geüpdatet. Deze velden zijn onderdeel van de klasse **Overlay**.

4.4.1. Ontwerp keuzes

#	Probleem	Besluit	Argumenten
OK-01	Hoe kun je zonder te veel aanpassingen in de applicatie wisselen van control method?	Door een strategy pattern te gebruiken	Door de control method te implementeren aan de hand van een strategy pattern kun je altijd meer control methods toevoegen zonder dat dit veel aanpassingen vergt van in de applicatie.
OK-02	Hoe zorg je ervoor dat ongeacht de huidige control method de essentiële knoppen zoals bijvoorbeeld de noodstop altijd zichtbaar zijn?	Door de control method in een los paneel te plaatsen	Door de control method in een los paneel te plaatsen is het makkelijk om van control method te wisselen en verzeker je dat de onderdelen in het MainFrame altijd zichtbaar zijn voor de gebruiker.
OK-03	Hoe kun je de robot besturen door middel van cartesische coördinaten en hoeken, tegelijkertijd?	Door het gebruik van tekstvelden, sliders en een simulatie te combineren en er voor te zorgen dat alle onderdelen onderling gelinkt zijn.	Door deze twee aspecten (inverse en forwards kinematica) te combineren is het mogelijk om beide aspecten feilloos samen te laten werken waardoor er een overzichtelijk en duidelijk besturingsmethode ontstaat.
OK-04	Hoe maak je een interactieve simulatie die het mogelijk maakt de robot te besturen?	Door de gebruiker met de muis de simulatie te laten bewegen	Door dat de gebruiker direct met de muis de simulatie kan besturen en de posities actief te berekenen is het mogelijk om een interactieve simulatie te realiseren.
OK-05	Hoe combineer je een interactieve simulatie met de daadwerkelijke posities van de robot?	Door een 'schaduw' simulatie te implementeren.	Door middel van een 'schaduw' simulatie is de daadwerkelijke positie van de robot te zien, deze simulatie wordt realtime geüpdatet met de bewegingen van de robot.
OK-06	Hoe verzeker je dat de robot nooit een start commando krijgt als deze al gestart is?	Door een toggle knop te gebruiken die zijn waarde (true/false) gebruikt om de robot of te starten of te stoppen.	Door een toggle knop te gebruiken kan er intern nooit iets fout gaan. De waarde van de knop is altijd gelijk aan de situatie van de robot.
OK-07	Hoe verzeker je dat de robot nooit een beweegcommando krijgt als deze niet gestart is?	Door all componenten omtrent beweging gedeactiveerd zijn tot het start commando verstuurd is	Door alle componenten te deactiveren is het voor de gebruiker niet mogelijk om (per ongeluk) een bewegingscommando naar de robot te versturen.
OK-08	Hoe verzeker je dat er nooit instellingen aangepast worden in de robot zonder dat er een seriële connectie is?	Door alles omtrent de communicatie met de robot te deactiveren totdat er een seriële connectie gerealiseerd is met de robot ('Connect'-knop)	Alle onderdelen van de applicatie zijn inactief tot er een seriële connectie is met de robot. Hierdoor kan een gebruiker nooit een commando versturen naar de robot zonder dat er een connectie is.

#	Probleem	Besluit	Argumenten
OK-09	Hoe geef je aan dat onderdelen niet geïmplementeerd zijn maar fungeren als placeholders?	Door een log bericht te genereren wanneer de gebruiker toch probeert deze placeholders te activeren.	Alle componenten die momenteel niet geïmplementeerd zijn in de applicatie zijn voorzien van een waarschuwing die aangeeft dat deze componenten nog niet geïmplementeerd zijn
OK-10	Hoe verzekeren we dat met gebruik van meerdere input methodes voor dezelfde data velden elk veld altijd up-to-date is?	Door elke input methode via dezelfde pipeline te updaten.	De verschillende input methodes volgen dezelfde pipeline waarin alle methodes gesynct worden, hierdoor is het mogelijk om meerdere input methodes te gebruiken die altijd dezelfde waarden weergegeven.

Tabel 4 - Ontwerp keuzes - Frame

4.5. Package - Base

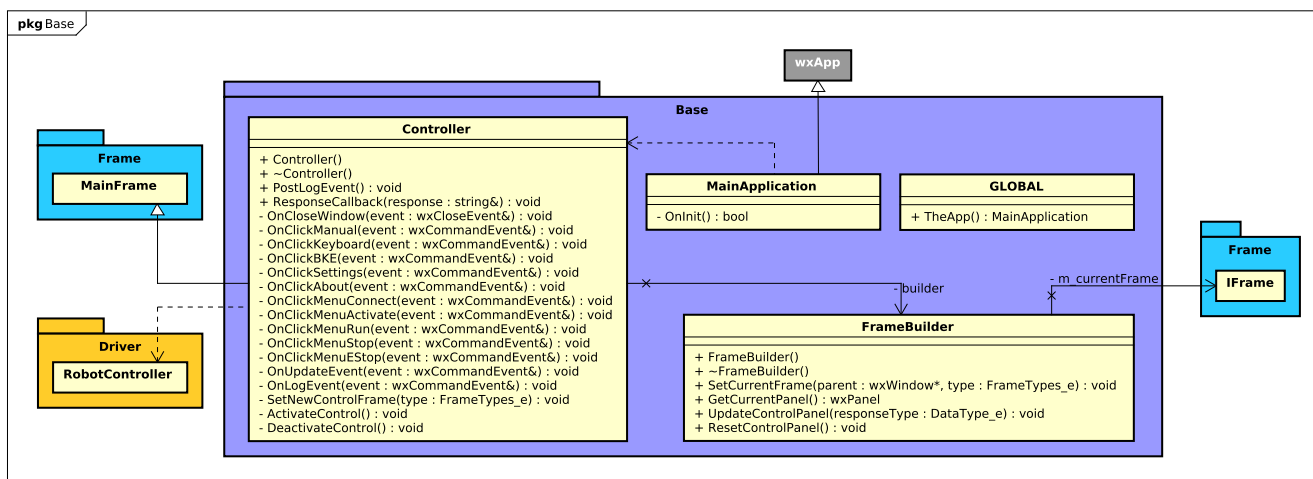


Diagram 6 - Class diagram - Base package

De package **Base** is het centrale punt van de applicatie. De klasse **Controller** erft van de klasse **MainFrame** (zie Package - Frame) en implementeert all functionaliteiten van de verschillende onderdelen verwerkt in de **MainFrame** klasse. Verder implementeert de **Controller** klasse ook de interactie tussen de driver (zie Package - Driver) en de control method. De klasse **FrameBuilder** is verantwoordelijk voor het aanmaken van de correcte control method. De **Controller** klasse kan aangeven dat het bijvoorbeeld een instantie van de **KeyboardCtrl** control method wil (momenteel niet geïmplementeerd), de **FrameBuilder** klasse maakt vervolgens een instantie aan en geeft deze weer in de applicatie. De klasse **MainApplication** initialiseert de applicatie, verder heeft het geen functionaliteit.

Het onderdeel **GLOBAL**, en hiermee de methode **TheApp** levert de mogelijkheid om een referentie naar de applicatie te krijgen. De **LogToWidget** klasse (zie Package - Logger) maakt hier bijvoorbeeld gebruik van. Door een 'externe' klasse toegang te bieden tot de applicatie kan er bijvoorbeeld van een andere thread geschreven worden naar het scherm, dat is wat deze methode doet.

#	Probleem	Besluit	Argumenten
OK-01	Hoe verzekeren we dat de seriële connectie is afgesloten voordat de applicatie wordt gestopt?	Door de applicatie een pop-up te laten weergeven voorafgaand aan het afsluiten.	Door een pop-up weer te geven kan de applicatie de tijd nemen voor het afsluiten van de seriële connectie. De applicatie stuurt eerst een noodstop bericht naar de robot, sluit de connectie en als dat allemaal klaar is sluit het de applicatie af.
OK-02	Hoe laat je de RobotController updates maken binnen de applicatie?	Door gebruik te maken van een callback die een update event aanroept binnen de Controller klasse.	Door gebruik te maken van een callback is het mogelijk om een wxWidgets event binnen de controller te triggeren, dat event zorgt er dan voor dat de applicatie en update uitvoert.
OK-03	Hoe laat je de Logger log berichten sturen naar frame elementen binnen de applicatie?	Door een methode te maken die een referentie naar de huidige applicatie weergeeft, aan de hand van die referentie kan er een log event getriggerd worden.	In essentie hetzelfde probleem als de voorgaande rij maar dan anders opgelost. Door een referentie te geven aan een externe instantie is het mogelijk de applicatie direct aan te spreken, Door deze interne link kan er dus een event getriggerd worden van buiten de applicatie.

Tabel 5 - Ontwerp keuzes - Base

4.6. Package - Utils

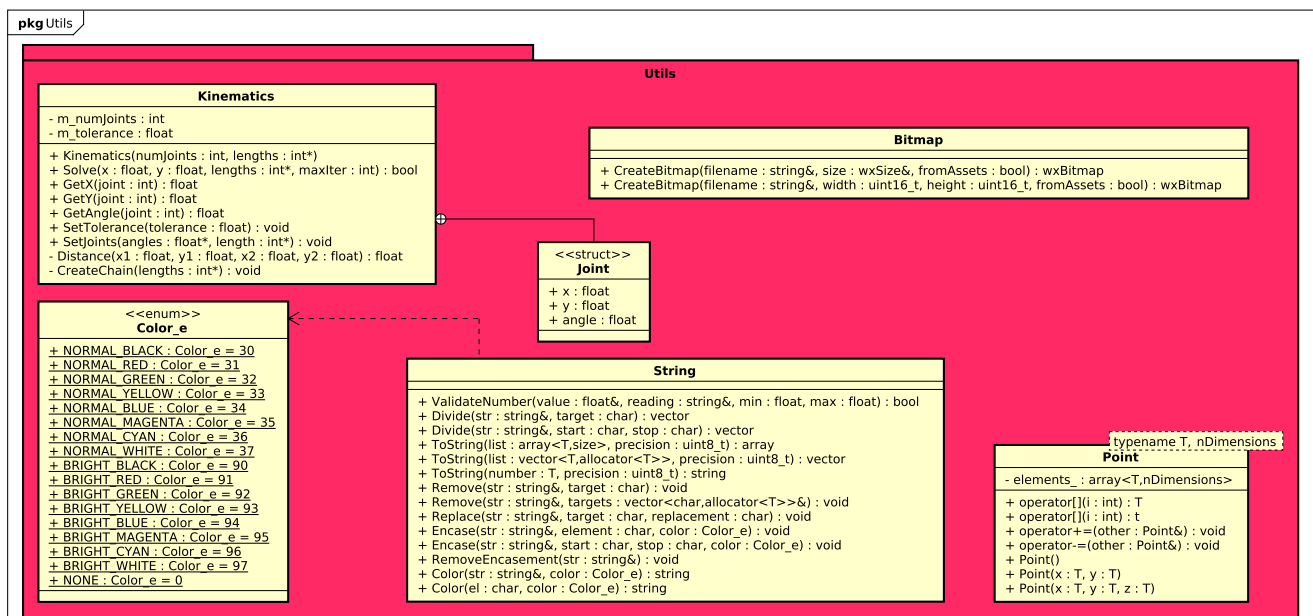


Diagram 7 - Class diagram - Utils package

De **Utils** package is een verzameling klasse die verschillende functionaliteiten uitvoeren. De klasse **Point** is een template klasse die het mogelijk maakt om een n-dimensionale coördinaat, van elk standaard data type, op te zetten. De klasse is geschreven en getest voor coördinaten tot drie dimensies, maar in theorie is er geen restrictie.

De klasse **Bitmap** levert de mogelijkheid om een geschaalde bitmap te krijgen. Binnen wxWidgets is het niet mogelijk om een bitmap te schalen. De bitmap moet eerst omgezet worden naar een image, deze kan vervolgens geschaald worden en tot slot moet deze image weer terug geconverteerd worden naar een bitmap. De applicatie maakt veel gebruik van bitmaps waardoor deze klasse uitermate handig is om de leesbaarheid op pijl te houden. De **String** klasse is een verzameling van verschillende string gerelateerde functionaliteiten. De methode **ToString** kan een standaard datatype converteren naar een string met een bepaalde precisie, erg handig tijdens het opstellen van een bericht voor de robot. de methodes **Divide**, **Encase** en **RemoveEncasement** worden allemaal gebruikt tijdens het parsen en het opstellen van berichten.

Tot slot de klasse **Kinematics**, De klasse is een voor C++ omgeschreven variant van de Arduino library **Fabrik2D**, geschreven door Henrik Söderlund. Deze Klasse wordt gebruikt voor de inverse en forwards kinematica gebruikt tijdens de simulatie (zie 'Package - Frame').

5. Literatuurlijst

Index	Source
1	Wikipedia-bijdragers. (2022, 10 februari). Universal serial bus. Wikipedia. Geraadpleegd op 27 mei 2022, van https://nl.wikipedia.org/wiki/Universal_serial_bus
2	Sourcemaking. (z.d.). Design Patterns and Refactoring. Singleton. Geraadpleegd op 30 mei 2022, van https://sourcemaking.com/design_patterns/singleton
3	Sourcemaking. (z.d.-b). Design Patterns and Refactoring. Facade. Geraadpleegd op 30 mei 2022, van https://sourcemaking.com/design_patterns/facade
4	Sourcemaking. (z.d.-c). Design Patterns and Refactoring. Strategy. Geraadpleegd op 30 mei 2022, van https://sourcemaking.com/design_patterns/strategy
5	Söderlund, H. (z.d.). GitHub - henriksod/Fabrik2DArduino: A FABRIK 2D inverse kinematics solver for Arduino. Fabrik2D. Geraadpleegd op 30 mei 2022, van https://github.com/henriksod/Fabrik2DArduino