Helsingin yliopisto
Tietojenkäsittelytieteen laitos

Cum laude laboratorio
Ohjelmistotuotantoprojekti Nobel

YLLÄPITODOKUMENTTI

Ilkka Forsblom, Risto-Matti Kääriäinen, Jussi Lahti,
Maarit Saaresto & Jarkko Salminen

versio 1.0

Ohjaaja:	**Teemu Kurppa**
Tilaaja:	**Tapio Elomaa**
jätetty:	3.5.2001

Erikoissanasto ja käytetyt lyhenteet

este robotin liikkeen estävä objekti fyysisessä maailmassa.

fyysinen reitti robotin fyysisessä maailmassa kulkema reitti.

hätäpysäytys robotin pysähtyminen sonarien antaman datan perusteella.

kartta koordinaattijoukko, joka rajaa yhden tai useamman alueen robotille kiellettyksi alueeksi tai alueiksi. Ohjelma saa koordinaatit syötteenään.

kielletty alue alue, jolle robotin ei toivota kulkevan navigoinnin aikana. Kaikkien robotin ympäristössä olevien esteiden tulisi sijaita kiellettyjen alueiden sisäpuolella.

lasketut reitit reitit, jotka on laskettu maalipisteeseen sisäisellä kartalla.

liikkuminen robotin fyysinen liikkuminen.

maamerkki robotin ympäristöstään tunnistama hahmo, jonka koordinaatit suhteessa karttaan on sen tiedossa.

maalipiste kartan koordinaattipiste, johon ohjelmisto pyrkii robotin ohjaamaan.

etäisyys , joka vähintään jää navigoitaessa kiellettyjen alueiden ja robotin väliin.

Martti **m**obiili **a**utonominen **r**obo**tti**

monitorointitietokone robotin ulkopuolinen tietokone, johon robotti pitää yhteyttä joko modeemilla radiolinkin tai ethernet–kaapelin välityksellä.

navigointi liikkuminen, reitinlaskenta ja paikanmääritys yhdessä.

odometria kuljetun matkan ja rintamasuunnan arviointi pyörien pyörimisen perusteella

ohjelman keskeytys ohjelman keskeyttäminen SIGINT:llä (näppäimistöltä annettu keskeytys, CONTROL+C), joka ainoastaan pysäyttää robotin liikkumisen ja lopettaa ohjelman suorituksen.

ohjelman lopettaminen käyttäjän pyytämä (esim. johonkin näppäimeen sidottu) ohjelman hallittu lopettaminen, joka pysäyttää robotin. Pysäytyksen yhteydessä tehdään mm. tallennukset (kuvat, logit) ennen suorituksen lopettamista.

robotti tässä tapauksessa sonarein ja askelmoottorein varustettu tietokone.

segmentointi kuvan värialueiden erottaminen.

sensori havaintolaite, jonka avulla robotti tutkii ympäristöään. Projektin yhteydessä sensoreina käytetään odometriaa, ultraääniantureita ja videokameraa.

sisäinen kartta ohjelman sisäinen tietorakenne, joka sisältää (marginaaleilla laajennetut) kielletyt alueet sekä kaikki lasketut reitit maalipisteeseen

paikanmääritys sijainnin selvittäminen maamerkkien avulla.

pysäytysetäisyys robotin ylläpitämä minimietäisyys esteisiin. Robotin etäisyyttä esteisiin havainnoidaan sonareiden avulla. Esteeseen reagoidaan kun se havaitaan pysäytysetäisyyden sisäpuolella.

sonar ääntä heijastavien pintojen ja niiden etäisyyksien havainnointiin käytetty ultraääniluotain.

SVM **S**upport **V**ector **M**achine, tukivektorikone. Koneoppimismenetelmä, jota tuotettavassa ohjelmistossa käytetään maamerkkien hahmojen tunnistamiseen.

yllättävä este pysäytysetäisyyden sisällä oleva sonareiden havaitsema este.

Sisältö

1 Ohjelmiston yleiset periaatteet

1.1 Yleisarkkitehtuuri

Arkkitehtuuri on suunniteltu siten, että yksi osajärjestelmä (`RobotControl`) toimii keskuksena, joka keskustelee kaikkien muiden osajärjestelmien kanssa, ja tekee päätöksiä ohjelman suorituksen kulusta (minkä osajärjestelmän palveluita kutsutaan milloinkin) saamiensa palautteiden ja syötteiden perusteella. Muut osajärjestelmät (`Localization`, `Map`, `Movement`) ovat itsenäisiä, eivätkä tarvitse toisiltaan palveluita omien toimintojensa suorittamiseen. Näiden osajärjestelmien lisäksi ovat lisäksi pakkaukset `Nobel`, `External` ja `Input`. `Nobel` on kokoelma erilaisia pieniä olioita (mm. itsekonvertoivia mittayksiköitä), joita käytetään jokaisessa osajärjestelmässä sekä sisäisen toteutuksen apuna että erityisesti rajapintojen välisinä viestiolioina. `External` tarjoaa osajärjestelmille keinot käyttäjän kanssa kommunikoimiseen tulostuspalveluiden ja syötteiden luvun muodossa. `Input` tarjoaa keinon lukea käyttäjän näppäimistökomentoja. Inputia käytetään vain `RobotControl`–osajärjestelmässä.

1.2 Osajärjestelmien sisäinen arkkitehtuuri yleisesti

Osajärjestelmät eivät ole suunnitteluperiaatteiltaan täysin yhtenevät toistensa kanssa. `RobotControl` on siitä poikkeuksellinen osajärjestelmä, että se sisältää käytännössä vain yhden luokan/olion, joka pitää yhteyttä kaikkiin muihin osajärjestelmiin ja pitää yllä tietoa ohjelman ja robotin toimintatilasta. Muilla osajärjestelmillä (`Localization`, `Map`, `Movement`) on kaikilla yhteistä se, että niillä kaikilla on yksi ko. osajärjestelmää ohjaava olio, joka toteuttaa osajärjestelmän ulkoisen rajapinnan. Tämä rajapinta on määritelty osajärjestelmän abstraktissa rajapintaluokassa, jonka osajärjestelmän ohjausolio siis perii ja toteuttaa rajapintaluokan määräämät palvelut.

`Movement`– ja `Map`–osajärjestelmien ohjausolioiden konstruktorit ottavat parametreinaan viitteet tarvitsemiensa palveluiden tarjoaviin olioihin. Näin pystyy helpommin korvaamaan yhden palan ko. osajärjestelmistä toisella, saman rajapinnan toteuttavalla koskematta itse osajärjestelmässä muuhun koodiin kuin itse lisäämäänsä. `Localization`–osajärjestelmässä puolestaan jokainen luokka itse luo kaikki tarvitsemansa oliot, ja myös itse vastaa tarvittavien parametrien hakemisesta Externalilta.

Kaikki ohjelman käyttämät kirjastot ja niiden funktiot (paitsi ohjelmointikielen ja käyttöjärjestelmän tarjoamat) on eristetty ohjelman varsinaisen logiikan omaavasta osasta rajapintaluokkien ja –funktioiden taakse. Tällä tavoin käytetyn kirjaston vaihtaminen toiseen, hiukan erilailla toimivaan mutta samat palvelut tarjoavaan kirjastoon edellyttää muutoksen tekemisen vain rajapintaluokkiin ja –funktioihin.

1.3 Luokkien ja olioiden arkkitehtuuri yleisesti

1.3.1 Handle–Body–jaottelu

Kaikki ohjelman oliot (paitsi useimmat poikkeukset) käyttävät Handle–Body–jaottelua. Siinä olion julkiset metodit esitellään varsinaisessa olion header–tiedostossa (olion Handle). Yksityiset tietojäsenet ja mahdolliset yksityiset apufunktiot sen sijaan on sijoitettu erilliseen olioon ja headertiedostoon (olion Bodyyn). Handle sisältää pelkästään tiedon Body–luokan nimestä (jos varsinainen luokka on nimeltään Olio, on Body–luokka nimeltään OlioRep), ja ainoa Handlen tietojäsen on osoitin Body–olioon. Näin jokainen ohjelman olio, jota käytetään useammasta eri paikasta, on koko elinkaarensa ajan samankokoinen. Tästä on se etu, että vaikka jonkin luokan sisäinen toteutus muuttuu, ei muita luokkia (lähdekooditiedostoja), jotka käyttävät muutettua luokkaa, tarvitse kääntää uudelleen, kunhan rajapinnat pysyvät muuttumattomina.

Käytännössä siis jokainen olio on toteutukseltaan jakautunut neljään tiedostoon: Olio.h (julkinen rajapinta, Handle), OlioRep.h (yksityinen rajapinta, Body), Olio.cc (julkisen rajapinnan toteutus) ja OlioRep.cc (yksityisten apumetodien ja –funktioiden toteutus). Joskus OlioRep.cc–tiedosto on jätetty pois, ja kaikki metodit on siis toteututettu Olio.cc–tiedostossa sekä mahdollisesti joitain apumetodeja inline–muodossa OlioRep.h–tiedostossa.

Luokkia, jotka käyttävät Handle–Body–jaottelua, käytetään kuten mitä tahansa luokkia, eli käytetään luokan julkisen rajapinnan toteuttavaa header–tiedostoa. Ainoastaan luokan toteuttajan tarvitsee murehtia Body–osiosta, mistään muualta sitä ei tarvitse eikä pidä käyttää.

1.3.2 Itse itsensä testaavat oliot: SelfTestingObjectin perintä

Käytännössä kaikki ohjelman luokat (paitsi poikkeukset) ovat `NobelNS::SelfTestingObject`–luokan aliluokkia. `SelfTestingObject`–luokan aliluokkien tulee toteuttaa `invariant()`–metodi, jolla tarkistetaan, onko olio yhä eheässä tilassa. Tämän avulla voidaan helposti havaita joitakin yksinkertaisia virheitä nopeasti. Lisäksi SelfTestingObjectin headereissa esitellään makroja, joilla olio voi testata omaa tilaansa sekä tarkistaa metodikutsuissa tiettyjä alku– ja loppuehtoja niin, että ohjelmaa kehitettäessä ohjelman suoritus lopetetaan assertilla, mutta julkaistavassa versiossa assertit voidaan korvata poikkeuksilla pelkästään SelfTestingObjectin headeria muuttamalla (määrittelemällä `NDEBUG` ohjelmasta käännetään julkaisuversio, määrittelemällä lisäksi `OPTIMIZATION` ohjelmasta käännetään optimoitu versio, jossa mikään testimakro ei tee mitään).

SelfTestingObjectissa esitellyt makrot ovat:

INV() testaa olion tilan `invariant()`–metodilla ja keskeyttää ohjelman assertilla (ohjelman kehitysversio) tai heittää poikkeuksen `InvariantException` (julkaisuversio), jos olio ei ole eheässä tilassa

INVDB() kuten INV(), paitsi että tarkistus poistetaan kokonaan ohjelman julkaisuversiosta (esim. liian paljon aikaa vievä tarkistus)

TEST(testi, exception) jos `testi` on epätosi, keskeytetään ohjelma assertilla (kehitysversio) tai heitetään poikkeus `exception` (julkaisuversio)

TESTDB(testi) jos `testi` on epätosi, keskeyttää ohjelman assertilla (kehitysversio), julkaisuversiossa testiä ei suoriteta ollenkaan

1.3.3 Ohjelman poikkeukset

Jokainen ohjelman poikkeus on periytetty standardikirjaston poikkeuksesta `std::runtime_error`. Näin kaikki ohjelman poikkeukset pystyy nappaamaan ohjelman ylimmällä tasolla yhdellä `catch`–lauseella. `Header`–tiedostoissa funktioiden määrittelyissä on pyritty ilmoittamaan `throw`–määreellä funktioiden heittämät poikkeukset, jos funktio voi heittää jonkin oman kyseiseen luokkaan liittyvän poikkeuksen. Mahdollisia `InvariantExceptioneita` ei ole erikseen mainittu. Kaikki funktiot, jotka voivat muuttaa olion tilaa, saattavat heittää `InvariantExceptionin`.

2 `Nobel`–pakkaus

`Nobel`–pakkaus sisältää pieniä apuluokkia, joita käytetään ohjelman toteutuksessa etenkin rajapintojen välillä kulkevina viestiolioina, mutta myös varsinaisen logiikan toteutuksen apuna. `Nobel` sisältää itsekonvertoivia mittayksiköitä (`Angle`, `Length`, `RobotVelocity`, `RobotAcceleration`), vektorin (`Vector`) ja sijoittautumisen (sijainti+suunta, `Pose`). Lisäksi `Nobel` sisältää poikkeuksen `BlockException`, joka oikeastaan kuuluisi `Movement`–osajärjestelmään, sekä `SelfTestingObject`–luokan. Kaikki Nobelin luokat, paitsi `SelfTestingObject`, käyttävät Handle–Body–menetelmää toteutuksessaan. Pakkaus sijaitsee nimiavaruudessa `NobelNS`.

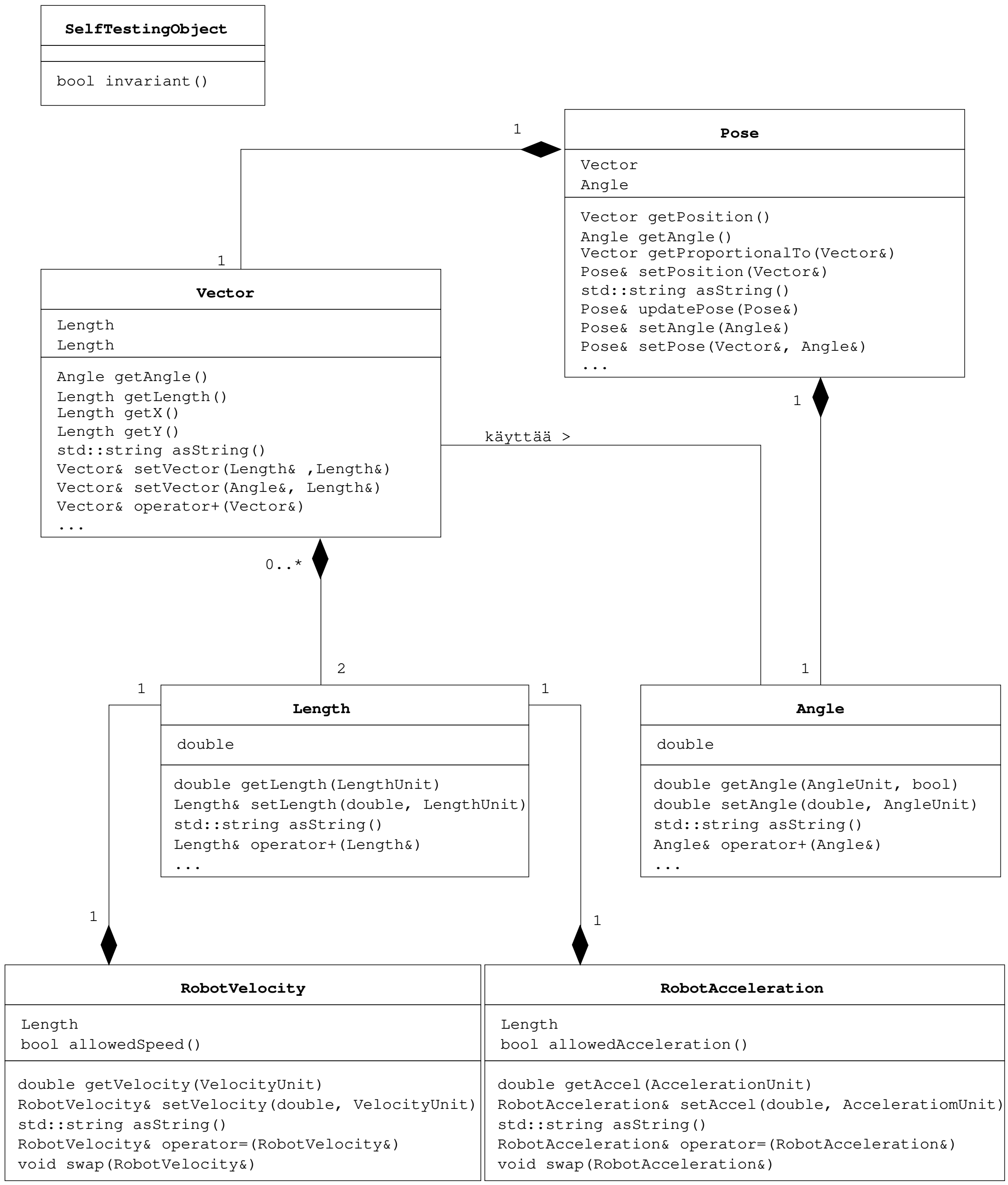
2.1 `Angle`–luokka

`Angle` on itsekonvertoiva mittayksikkö. Sillä on metodi `double getAngle(AngleUnit)`, jolla voi pyytää siltä talletetun kulman eri mittayksiköissä. Parametri `AngleUnit` on lueteltu tyyppi, jonka perusteella metodi konvertoi talletetun kulman haluttuun mittayksikköön. Palautettu kulma on normalisoitu alueelle]-180, 180] astetta, paitsi jos aksessorille antaa toisena parametrina totuusarvon `true`, jolloin palautettu kulma normalisoidaan alueelle [0, 360[astetta.

Anglen konstruktorin `Angle(double=0, AngleUnit=Deg)` parametrit ovat kulman suuruus ja mittayksikkö. Jo olemassaolevaa Angle–oliota voi muuttaa `setAngle(double, AngleUnit)`–metodilla.

Angle sisältää ylikuormitetut operaattorit kahden kulman yhteen– ja vähennyslaskulle sekä kulman ja double–luvun kerto– ja jakolaskulle. Ylikuormitetuista operaattoreista on versiot sijoituksella ja ilman.

Sisäisesti kulma talletetaan double–arvona, joka on kulman suuruus asteina, normalisoituna alueelle]-180, 180]. Normalisointia varten on apufunktiot



Kuva 2.1: Nobel–osajärjestelmä

`double normalize360(double)` ja `double normalize180(double)`, esitelty ja määritelty tiedostossa `AngleRep.h`.

2.2 `Length`–luokka

`Length` on itsekonvertoiva pituusyksikkö. Sillä on metodi `double getLength(LengthUnit)`, jolla voi pyytää siltä talletetun pituuden eri mittayksiköissä. Parametri `LengthUnit` on lueteltu tyyppi, jonka perusteella metodi konvertoi talletetun pituuden haluttuun mittayksikköön.

Lengthin konstruktori on `Length(double=0, LengthUnit=CM)`. Sisäisesti Length on talletettu double–lukuna, joka tulkitaan pituudeksi senttimetreissä.

Length sisältää ylikuormitetut operaattorit kahden Lengthin yhteen– ja vähennyslaskulle sekä Lengthin ja double–luvun kerto– ja jakolaskulle. Operaattoreista on olemassa versiot sekä sijoituksella että ilman.

2.3 `Vector`–luokka

`Vector` on vektori–luokka, jonka ilmentymiä käytetään ilmaisemaan sekä sijaintia kartalla että liikevektoria (suunta ja matka, joka liikutaan). Sisäisesti `Vector` talletetaan kahtena `Length`–oliona, jotka ilmaisevat vektorin X– ja Y–komponentteja.

Vectorilla on seuraavat metodit talletetun informaation hakemiseen: `Length getLength()` hakee vektorin pituuden, `Angle getAngle()` hakee vektorin kulman, `Length getX()` hakee X–komponentin ja `Length getY()` hakee Y–komponentin.

Vectorilla on kaksi konstruktoria. `Vector(Length=Length(), Length=Length())` ottaa parametreinaan X– ja Y–komponenttien pituudet. `Vector(Length, Angle)` ottaa parametreinaan vektorin pituuden ja kulman. Jo olemassaolevan Vectorin voi asettaa vastaavat parametrit ottavilla metodeilla `setVector(Length, Length)` ja `setVector(Length, Angle)`.

Vector sisältää ylikuormitetut operaattorit kahden Vectorin yhteen– ja vähennyslaskulle, sekä sijoituksella että ilman.

2.4 `Pose`–luokka

`Pose` ilmaisee robotin sijoittautumisen (sijainnin ja suunnan). Sitä käytetään sekä robotin oikean sijoittautumisen ilmaisemiseen että kertomaan, mihin robotti on liikkunut edellisestä pisteestä (suhteellinen sijoittautuminen).

Pose sisältää seuraavat metodit talletetun informaation hakemiseen: `Vector getLocation()` palauttaa sijainnin ja `Angle getAngle()` palauttaa suunnan.
`Vector getProportionalTo(Vector)` palauttaa parametrina annetusta suuntavektorista (joka olettaa robotin katsovan kartan X–akselin suuntaan) käännetyn version,

jossa robotin oikea suunta (joka on talletettu `Pose`–olioon) on otettu huomioon.

`Pose`–olion muokkaamiseen on seuraavat metodit: `setPosition(Vector)` asettaa uuden sijainnin, `setAngle(Angle)` asettaa uuden suunnan, `setPose(Vector, Angle)` asettaa uuden sijainnin ja suunnan yhdellä kertaa. Metodi `updatePose(Pose)` päivittää sijoittautumisen seuraavasti: parametrina saatu sijoittautuminen on sijoittautuminen, jossa robotti on koordinaatistossa, jonka origo on päivitettävän sijoittautumisen sijainti ja X–akseli päivitettävän sijoittautumisen suunnan suuntainen.

Posella on konstruktori `Pose(Vector=Vector(), Angle=Angle())`, jonka parametrit ovat sijainti ja suunta. Sisäisesti Pose esitetään `Vector`– ja `Angle`–olion avulla.

2.5 `RobotVelocity`–luokka

`RobotVelocity` on luokka, jonka ilmentymät ovat robotin nopeuksia. `RobotVelocity`–oliot osaavat kertoa talletetun nopeuden eri mittayksiköissä. `RobotVelocity`–oliot myös pitävät huolen siitä, että talletettu nopeus on sallituissa rajoissa.

`RobotVelocity`–olion tietosisällön voi hakea metodilla `double getVelocity(VelocityUnit)`. Tietosisällön voi asettaa metodilla `setVelocity(double, VelocityUnit)`. Konstruktori on muotoa `RobotVelocity(double, VelocityUnit)`. Metodi setVelocity ja konstruktori voivat heittää poikkeuksen `std::range_error`, jos annettu nopeus ei ole sallituissa rajoissa.

Sisäisesti RobotVelocityn tietosisltö talletetaan `Length`–oliona.

2.6 `RobotAcceleration`–luokka

`RobotAcceleration` on luokka, jonka ilmentymät ovat robotin kiihtyvyyksiä. `RobotAcceleration`–oliot osaavat kertoa talletetun kiihtyvyyden eri mittayksiköissä. `RobotAcceleration`–oliot myös pitävät huolen siitä, että talletettu kiihtyvyys on sallituissa rajoissa.

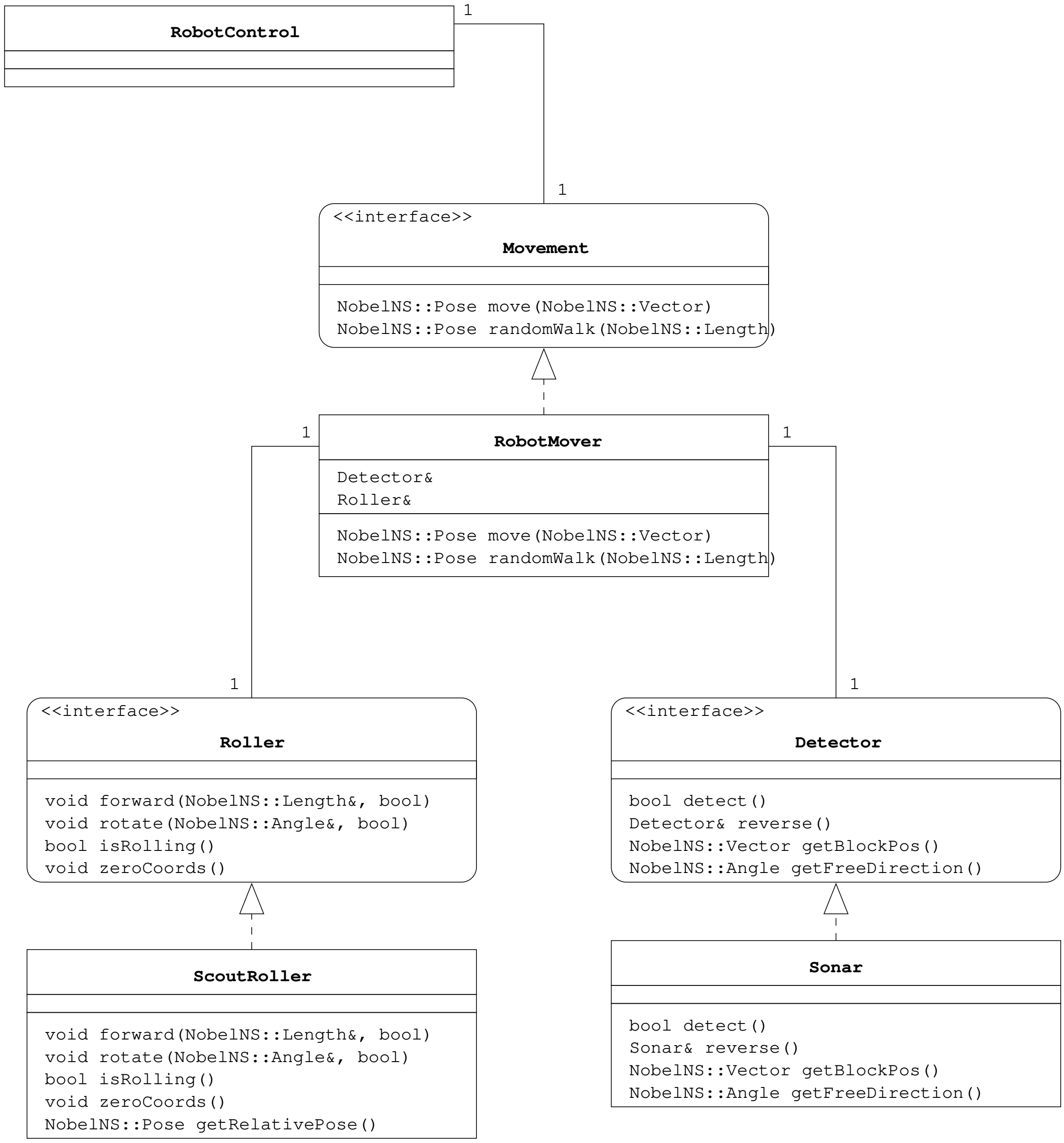
`RobotAcceleration`–olion tietosisällön voi hakea metodilla
`double getAccel(AccelerationUnit)`.

Tietosisällön voi asettaa metodila
`setAccel(double, AccelerationUnit)`.

Konstruktori on muotoa `RobotAcceleration(double, AccelerationUnit)`.

Metodi `setAccel()` ja konstruktori voivat heittää poikkeuksen `std::range_error`, jos annettu kiihtyvyys ei ole sallituissa rajoissa.

Sisäisesti RobotAccelerationin tietosisältö talletetaan `Length`–oliona.



Kuva 3.2: `Movement`–osajärjestelmä

2.7 `BlockException`–poikkeus

`BlockException`–poikkeus heitetään, kun robotti liikkuessaan havaitsee esteen. `BlockException` sisältää seuraavat metodit poikkeuksen tietosisällön hakemiseen: `Vector getUnfinishedMovement()` palauttaa kulkematta jääneen matkan, `Vector getBlockLocation()` palauttaa havaitun esteen sijainnin, `Pose getRelativePose()` palauttaa robotin suhteellisen sijoittautumisen liikkumisen jälkeen.

3 `Movement`–osajärjestelmä

`Movement`–osajärjestelmä hoitaa robotin liikkumisen. Sen luokkia ovat `Movement`, `Detector`, `Roller`, `RobotMover`, `Sonar` ja `ScoutRoller`. Näistä luokista Handle–Body–menetelmää käytetään RobotMoverin, Sonarin ja ScoutRollerin toteuttamiseen. Muut luokat ovat abstrakteja rajapintaluokkia. Osajärjestelmä sijaitsee nimiavaruudessa `MovementNS`.

3.1 `Movement`–rajapintaluokka

`Movement`–luokka on järjestelmän ylin luokka. Se on abstrakti rajapintaluokka, joka määrittelee osajärjestelmän tarjoamat palvelut. Näitä ovat
`NobelNS::Pose move(NobelNS::Vector)`, jolla laitetta liikutetaan yhden liikevektorin verran, sekä
`NobelNS::Pose randomWalk(NobelNS::Length)`, joka liikuttaa robottia satunnaiseen suuntaan tietyn matkan verran.

3.2 `Detector`–rajapintaluokka

`Detector` on abstrakti rajapintaluokka, joka määrittelee, mitä palveluita tarvitaan esteet havaitsevalta luokalta. Näitä ovat `bool detect()`, joka kertoo onko havaittu este; `void reverse()`, joka kääntää havaintojen teon eteen– ja taaksepäin liikumista varten; `NobelNS::Vector getBlockPos()`, joka palauttaa viimeksi havaitun esteen suhteellisen sijainnin; ja `NobelNS::Angle getFreeDirection()`, joka hakee suunnan, jossa on eniten tilaa liikkua).

3.3 `Roller`–rajapintaluokka

`Roller` on abstrakti rajapintaluokka, joka määrittelee, mitä palveluita tarvitaan, jotta laitetta voidaan liikuttaa. Näitä ovat `void forward(Length)`, joka liikuttaa laitetta eteen/taaksepäin; `void rotate(Angle)`, joka kääntää laitetta paikallaan; `bool isRolling()`, joka kertoo liikkuuko laite; ja `void zeroCoords()`, joka nollaa laitteen sisäiset koordinaatit.

3.4 `RobotMover`–luokka

`RobotMover` on luokka, joka toteuttaa Movementin määräämän rajapinnan. RobotMoverin toteutus tarvitsee viitteet rajapintojen `Detector` ja `Roller` toteuttavien luokkien olioihin. Nämä annetaan RobotMoverin konstruktorissa, ja niitä pidetään tallessa `RobotMoverRep`–oliossa. Muuta tietosisältöä RobotMover ei tarvitse, eikä myöskään suoria yhteyksiä mihinkään muihin kirjastoihin.

RobotMoverin `move()`–metodin toteutus pyrkii minimoimaan robotin liikkeen siten, että kääntyminen suoritetaan siihen suuntaan, johon tarvitsee kääntyä vähiten, jotta robotti olisi suunnattu oikein. Käytännössä move() ei siis ikinä pyöritä robottia paikallaan 90 astetta enempää. Jos kääntyminen tehtiin "väärin päin", robottia liikutetaan takaperin, minkä oletetaan olevan yhtä nopeaa kuin eteenpäin liikkuminen. Paikallaan kääntymisen oletetaan onnistuvan aina, eikä silloin siis välitetä Detectorin havaitsemista esteistä (oikeastaan Detectorilta ei edes kysytä havaitseeko se esteitä). Samoin liikuttaessa hyvin lyhyitä matkoja (alle yksi senttimetri) ei esteistä välitetä. Eteen– ja taaksepäin liikuttaessa pollataan Detectorin `detect()`–metodia ja Rollerin `isRolling()`–metodia. Jos detect() palauttaa totuusarvon true, keskeytetään silmukkojen suoritus. Silmukkojen jälkeen käsketään robotin pysähtyä kutsumalla ScoutRollerin `forward()`–metodia 0–pituudella.

RobotMoverin `randomWalk()` on hyvin yksinkertainen: kysytään Detectorilta missä suunnassa on eniten tilaa, luodaan vektori saadusta kulmasta ja parametrina annetusta pituudesta, ja käsketään `move()`–metodin liikuttaa robottia lasketun vektorin verran.

3.5 `Sonar`–luokka

`Sonar`–luokka toteuttaa Detector–rajapinnan käyttäen Nomad Super Scout –robotin sonareita `Nclient`–kirjaston kautta. `Sonar`–luokan toteutus on erittäin riippuvainen `Nclient.h`–tiedostossa määritellystä sonarien rajapinnasta. Luokassa käytetään seuraavia kyseisen kirjaston funktioita ja tietoalkioita: `get_sn()`–funktiolla päivitetään sonarien lukemat, jotka puolestaan luetaan `State`–taulukosta, indekseistä `STATE_SONAR_0`–`STATE_SONAR_(SONARS-1)`, joiden oletetaan olevan State–taulukossa järjestyksessä peräkkäin. Lukemien mittayksikön oletetaan olevan tuumissa. Lisäksi konstruktori ja destruktori käyttävät `scout.h`–tiedoston funktioita `turn_sonars_on()` ja `turn_sonars_off()` sonarien kytkemiseen päälle ja pois, vastaavasti. Funktiot `turn_sonars_on()` ja `turn_sonars_off()` käyttävät Nclient.h:ssa esiteltyä funktiota `conf_sn(int rate, int order[16])`. Käytettyjen komentojen mahdollisesti palauttamaa totuusarvoa komennon onnistumisesta ei tutkita, komentojen oletetaan onnistuvan aina.

Konstruktorin parametrina annetaan taulukko (`std::vector`), jonka koko on `SONARS` (määritelty Nclient.h:ssa, SuperScoutilla arvo on 16). Taulukon alkiot ovat pysäytysetäisyyksiä (`NobelNS::Length`–olioita), yksi per sonar. Arvot talletetaan `SonarRep`–olioon tuumina (koska kirjaston antamat sonar–lukemat ovat myös tuumissa). `SonarRep` tallettaa myöskin viimeksi havaitun esteen suhteellisen sijainnin `NobelNS::Vector`–oliona. `Sonar`–olion elinaikana (aina konstruktorin kutsumisesta destruktorin kutsun päättymiseen) oletetaan yhteyden robottiin olevan olemassa.

3.6 `ScoutRoller`–luokka

`ScoutRoller`–luokka toteuttaa Roller–luokan Nomad SuperScout –robotille käyttäen `Nclient`–kirjastoa. Luokka tarvitsee `Nclient.h`–tiedoston rajapintoja seuraavasti: funktio `get_rv()` päivittää robotin pyörien liikkeen tilan; `get_rc()` päivittää odometriatiedot, jotka luetaan `State`–taulukon indekseistä `STATE_CONF_X`, `STATE_CONF_Y`, `STATE_CONF_STEER`; funktioita `zr()`, `dp(int x, int y)` ja `da(int th, int unused)` käytetään odometriatietojen nollaamiseen; funktiota `ws(unsigned char r_ws, unsigned char l_ws, unsigned char unused, unsigned char timeout)` käytetään robotin pysähtymisen odottamiseen. Lisäksi käytetään `scout.h`–tiedostossa esiteltyjä makroja `scout_pr(trans, steer)` robotin varsinaiseen liikuttamiseen, `scout_ac(value)` ja `scout_sp(value)` robotin nopeuden ja kiihtyvyyden asettamiseen, sekä funktiota `bool isMoving()` robotin liiketilan tutkimiseen. Kirjastojen käyttämien mittojen oletetaan olevan tuuman kymmenesosia ja asteen kymmenesosia. `ScoutRoller`–oliota käytettäessä oletetaan yhteyden robottiin olevan olemassa. Käytettyjen komentojen mahdollisesti palauttamaa totuusarvoa komennon onnistumisesta ei tutkita, komentojen oletetaan onnistuvan aina.

Konstruktorin parametreina annetaan robotin maksiminopeus (`NobelNS::RobotVelocity`) ja kiihtyvyys (`NobelNS::RobotAcceleration`).

4 `Map`–osajärjestelmä

4.1 `MapInterface`–rajapintaluokka

`MapInterface` on abstrakti luokka, joka määrittelee operaatiot, jotka koko reitinlaskentaosajärjestelmän on toteutettava. `RobotControl` käyttää osajärjestelmää tämän rajapinnan kautta.

4.2 `Map`–luokka

Luokka `Map` toteuttaa `MapInterface`–rajapinnan. Sen päätehtävänä on laskea lyhin reitti robotin sijaintipisteestä maalipisteeseen. Reitti lasketaan dijkstra–algoritmilla (löytyy mm. Weissin tietorakenteiden ja algoritmien oppikirjasta [1]), joka antaa tietystä kartan solusta lyhimmät reitit kaikkiin muihin soluihin.

Algoritmi käyttää aputietorakenteenaan STL–kirjaston prioriteettijonoa.

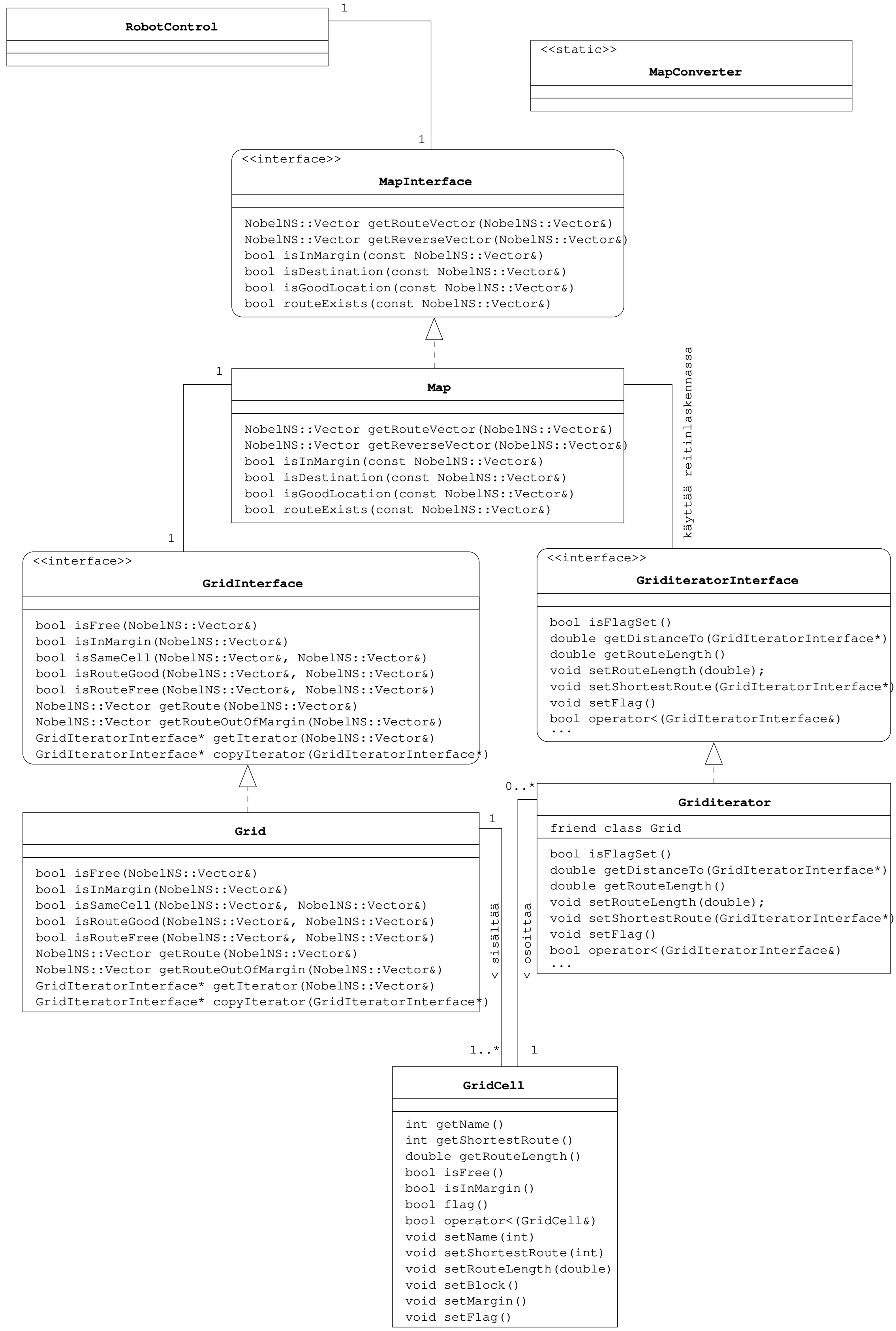
Jos kartta on erityisen iso, Map–olio pyrkii antamaan arvion dijkstran vaatimasta suoritusajasta, joka on luokkaa O(n*log n), missä n on kartan vapaiden solujen lukumäärä. Lausekkeen arvo kerrotaan lopuksi kokeellisesti valitulla vakiolla, joka ei kuitenkaan päde yleisesti. Prosessorin muuta kuormitusta tai mahdollista levymuistin käyttöä ei oteta huomioon, joten arvio on hyvin karkea.

Reitit lasketaan siten, että robotti ei liikkuisi marginaalin sisälle. Jos kuitenkin robotti joutuu odometriavirheen seurauksena marginaaliin, palautettavat reittivektorit kulkevat ulos marginaalista.

`getReverseVector()`–funktio palauttaa vektorin, joka johtaa suorinta reittiä ulos marginaalista. Tämä operaatio on hyödyllinen, kun robotti on joutunut niin lähelle karttaan merkittyä estettä, että sonarit havaitsevat kyseisen esteen. Tällöin on järkevää liikkua heti pois esteen läheisyydestä, eikä pyrkiä maaliin johtavalle optimaaliselle reitille.

Map heittää `BadDestinationExceptionin`, jos maalipiste kuuluu marginaaliin tai kartan kiellettyyn alueeseen.

Reitit kartan vapaista soluista maaliin lasketaan dijkstralla kerralla valmiiksi Map-olion konstruktorissa. Sen sijaan reittien optimointi tapahtuu `getRouteVector()`–funktion kutsun yhteydessä. Optimointi tapahtuu laskemalla yhteen peräkkäisiä dijkstralla laskettuja vektoreita. Summausta jatketaan, kunnes reitti kulkee esteiden tai marginaa-



Kuva 4.3: Map–osajärjestelmä

lialueiden läpi. Optimoinnin ansiosta reitti lyhenee. Muuten robotti liikkuisi vain dijkstran laskemia vektoreita, jotka kulkevat vain vaaka- pysty- tai diagonaalin suuntaan.

`MapRep`–olio sisältää Mapin sisäiset tietokentät: osoitin `Grid`–olioon ja maalipisteen sijainnin.

4.3 `GridInterface`–rajapintaluokka

`GridInterface` on abstrakti luokka, joka määrittelee operaatiot kartan toteuttavalle luokalle.

4.4 `Grid`–luokka

Luokka `Grid` pitää sisällään kartan tietorakenteen. Se on kaksiulotteinen ruudukko. Luokan tarkoituksena on tarjota korkeamman tason operaatioita `Map`–oliolle, jotta kartan käsittely on helppoa ja turvallista.

4.5 `GridIteratorInterface`–rajapintaluokka

`GridIteratorInterface` määrittelee operaatiot, jotka iteraattorin pitää toteuttaa.

4.6 `GridIterator`–luokka

GridIteratorit ovat osoittimia kartan ruutuihin. Iteraattoreiden on tarkoitus tehdä kartan ruutujen selaaminen yksinkertaiseksi ja turvalliseksi. Niiden operaatiot on suunniteltu `Map`–luokan dijkstra–algoritmin tarpeita ajatellen: iteraattoreilla voi käydä läpi kartan ruutujen vierusruutuja ja tallentaa lasketut reitit.

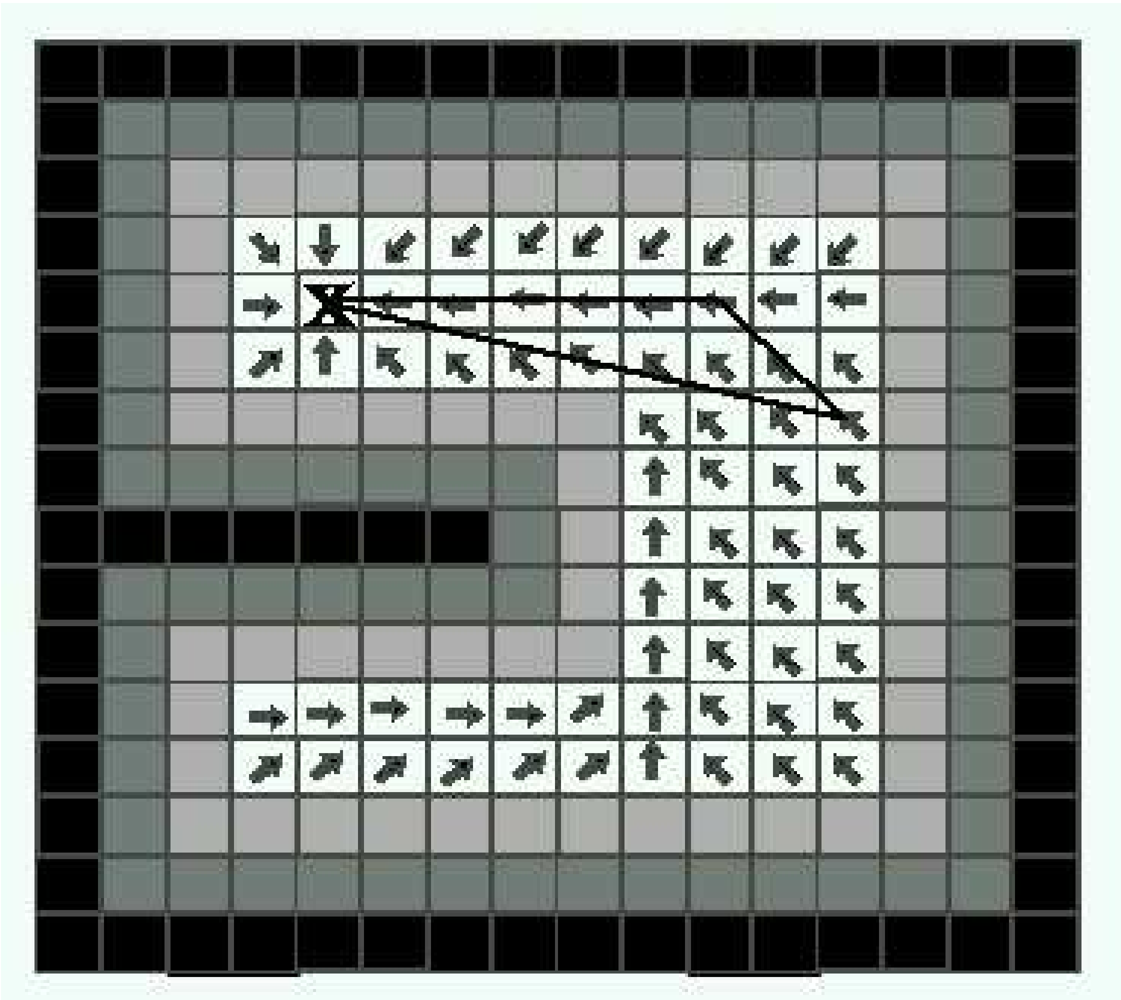
4.7 `GridCell`–luokka

`GridCell`–oliot ovat kartan alkioita, jotka voivat olla kolmessa tilassa: este, marginaali tai vapaa. `GridCell`–olioihin tallennetaan dijkstralla lasketut reitit.

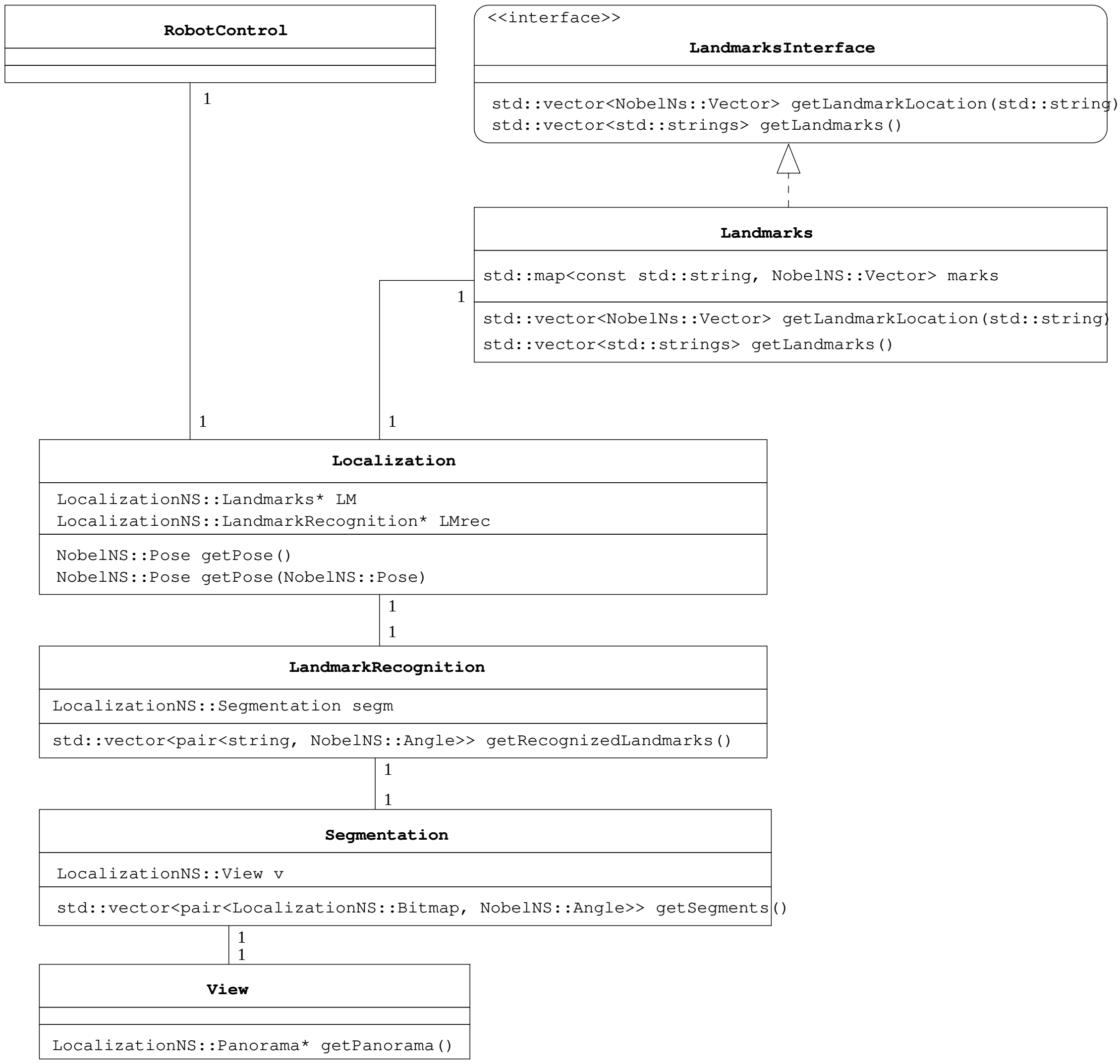
4.8 Esteiden kasvatus ja marginaali

Kuva 4.4 havainnollistaa reitinlaskentaosajärjestelmän sisäistä esitystapaa kartalle.

Kuvassa 4.4 uloimpana olevat tummimmat alueet ovat syötteenä annettun kartan esteitä. Seuraava vaaleampi vyöhyke on myös estettä, sillä reitinlaskenta ymmärtää robotin pistemäiseksi ja tämän takia esteitä pitää kasvattaa robotin säteen verran. Kolmas



Kuva 4.4: Järjestelmän sisäinen kartta



Kuva 5.5: `Localization`–osajärjestelmä

alue on marginaali, jolla liikkumista pyritään välttämään. Valkoiset ruudut ovat vapaita alueita. Nuolet kuvaavat dijkstra-algoritmilla laskettuja reittejä, jotka lasketaan ainoastaan vapaille ruuduille, ei marginaalille. Maalipiste on merkitty X:llä. Sen sijaan reitinlaskenta ei tallenna mitään tietoa robotin sijainnista. Kolme kapeaa viivaa esittävät optimointia: Diagonaalin suuntainen ja vaakasuora reitinpätkä ovat dijkstran mukaisia pisimpiä reittivektoreita. Kolmas suora on optimoinnilla laskettu reitti, joka karsii turhat mutkat.

5 `Localization`–osajärjestelmä

`Localization` on ohjelmiston osajärjestelmistä suurin. Se toteuttaa robotin paikanmäärityksen visuaalisten maamerkkien avulla. Maamerkit tunnistetaan robotin kameran ottamista kuvapanoraamoista erottamalla maamerkin väriset värialueet ja tunnistamalla niiden hahmot hahmontunnistuskoneilla. Robotin sijainnin laskemiseen kartalla tarvitaan kolme tunnistettua maamerkkiä, niiden kulmat suhteessa robotin kulkusuuntaan ja maamerkkien absoluuttiset sijainnit kartalla.

Paikanmääritysosajärjestelmä ei salli saman maamerkin käyttämistä useissa eri paikoissa.

Kuvien ottamisesta huolehtii `View`–luokka, segmentoinnista `Segmentation`–luokka, maamerkkien tunnistuksesta `LandmarkRecognition`–luokka, maamerkkien sijaintitietoja hallitsee `Landmarks`–luokka ja sijainnin laskemisen toimittaa `Localization`–luokka, joka samalla on koko `Localization`–osajärjestelmän rajapinta ulospäin. Lisäksi osajärjestelmässä on neljä ns. kirjastorajapintaa, jotka piilottavat todelliset laitteistojen kirjastorajapinnat, jotta osajärjestelmän luokat eivät olisi suoraan riippuvaisi mistään tietystä kirjastosta. Kirjastorajapinnat koostuvat kokoelmista staattisia funktioita, jotka todellisen kirjaston piilottamisen lisäksi myös yksinkertaistavat kirjastojen monimutkaisemman käytön selkeiksi, lukumääriltään vähäisiksi funktiokutsuiksi. Kirjastorajapintoja ovat `Camera`, joka hoitaa yhteyden kuvankaappauskortin kirjastoon; `PanTilt`, joka hallitsee kameran liikuttamisen pantilt–yksiköllä; `BasicSegmentation`, joka tekee perussegmentoinnin ja `ShapeRecognition`, joka tunnistaa hahmot. Kaikki `Localization`–osajärjestelmän luokat ja kirjastorajapinnat sijaitsevat `LocalizationNS`–nimiavaruudessa.

5.1 `Localization`–luokka

`Localization`–luokan ilmentymiä on tarkoitus olla olemassa ainoastaan yksi sovelluksen ajon aikana. Sen tarkoitus on `Landmarks`– ja `LandmarkRecognition`–luokkien ilmentymiltä saamiensa tietojen avulla laskea robotin sijainti ja asento.

Metodi `NobelNS::Pose getPose()` laskee robotin sijainnin ja suunnan. Se pyytää `LandMarkRecognition`–oliolta havaittuja maamerkkejä (nimi ja kulma). `Landmarks`–oliolta pyydetään havaittujen maamerkkien sijainteja nimien perusteella. Mahdollisesti kahdessa eri paikassa havaitut maamerkit karsitaan pois. Robotin sijainti lasketaan `triangulate`–funktiolla. Jos jotain menee pieleen, getPose() heittää poikkeuksen `LocalizationException`.

Konstruktori `Localization()` luo Landmarks– ja `LandmarkRecognition`–luokista ilmentymät.

Destruktori `Localization()` tuhoaa konstruktorin luomat `Landmarks`– ja `LandmarkRecognition`–luokkien ilmentymät.

Funktio `NobelNS::Pose triangulate(Angle, Angle, Angle, pair<double,double>, pair<double,double>, pair<double,double>)` on yksityinen metodi. Se toteuttaa Cohenin ja Kossin triangulointiartikkelissa [2] esitetyn geometrisen triangulointimenetelmän algoritmin. Se palauttaa robotin sijoittautumisen koordinaatistossa, käyttäen sijaintina kameran sijaintia. `getPose()` korjaa lasketun sijainnin bobotin keskustaan (eroa kameraan noin 10 cm) yksityisellä funktiolla
`NobelNS::Pose LocalizationRep::fixLocation(NobelNS::Pose)` ennen sijoittautumisen palauttamista `RobotControl`:lle.

Ennen teriangulaation laskemista pooistetaan `LandmarkRecognition`–olion antamasta nähtyjen maamerkkien vektorista (katso luku 5.4) kaikki useammin kuin kerran esiintyvät maamerkit yksityisen aliohjelman `void LocalizationRep::stripDoubles(std::vector< std::pair<std::string, Angle> >&)` avulla. Poisto on tarpeen tehdä, sillä jokaisen maamerkin vaatimusten nojalla [3] on oletettu olevan ainutkerainen.

5.2 `LocalizationException`–poikkeus

LocalizationException sisältää tiedon nähdyistä maamerkeistä. Nähtyjen maamerkkien määrän saa selville metodilla `int getLandmarkCount()`. Nähtyjen maamerkkien nimet saa selville metodilla `std::vector<std::string> getSeenLandmarks()`.

5.3 `Landmarks`–luokka

Metodi `std::vector<NobelNS::Vector> getLandmarkLocation(std::string)` palauttaa pyydetylle maamerkille taulukon sijainteja, jos maamerkillä ei ole yksikäsitteistä sijaintia. `Landmarks`–luokan tämänhetkinen toteutus kuitenkin ymmärtää vain yksikäsitteisiä maamerkkejä (eli palautetussa taulukossa on aina korkeintaan yksi alkio). Useammassa paikassa sijaitseva merkki hylätään kokonaan.

Konstruktorissa luetaan parametrina annettua maamerkkitiedostoa alusta kunnes tulee syntaksivirhe, jolloin luku lopetetaan. Onnistuneesti luetut maamerkit tulostetaan. Syntaksivirheen jälkeen ei jatketa maamerkkitiedoston lukua, vaan konstruktorista palataan. Maamerkkitiedostoon ei sallita kommentointeja.

5.4 `LandmarkRecognition`–luokka

`LandmarkRecognition`–luokan tehtävä on tunnistaa bittikarttahahmot maamerkeiksi. Sen ainoa toiminnallinen julkinen funktio on `getRecognizedLandmarks()`, joka käyttää hyväkseen `ShapeRecognition`–kirjastorajapintaa, jonka staattisia funktioita ovat `initMachines(vector<string>)` ja `recognize(Bitmap)`. Välillisesti, ShapeRecognitioninin kautta, `LandmarkRecognition`–luokka on riippuvainen `SVMFu`–kirjastosta, joka sisältää välineet bittikarttahahmojen tunnistamiseksi hahmontunnistuskoneilla. Kirjastoa on tätä sovellusta varten hieman muunnettu, SvmFuSvmTest.cpp-tiedostosta on poistettu yksi tulostus esteettisistä syistä. Käytännössä ohjelmisto toimii myös alkuperäisellä kirjastolla.

Hahmontunnistuskoneet on koulutettava tunnistamaan maamerkit ennen ohjelman suoritusta. Koulutukseen voidaan käyttää `SVMFu`–kirjastoon sisältyvää `SVMFuTrain`–sovellusta ja koneiden testaukseen `SVMFuTest`–sovellusta. Koulutukseen tarvitaan materiaalia positiivisista ja negatiivisista hahmoista suhteessa jokaiseen maamerkkiin.

`LandmarkRecognition`–luokan konstruktori saa parametrikseen vektorin, joka sisältää hahmontunnistuskoneiden tiedostojen nimet. Konstruktori kutsuu ShapeRecognitionin `initMachines(vector<string>)`–funktiota, joka saa parametrikseen maamerkkien tunnistuskoneiden nimet hakemistopolkuineen vektorissa. Hahmontunnistuskoneet on sijoitettava `SVM`–nimiseen hakemistoon, jota etsitään lähtien siitä hakemistosta, josta ohjelma käynnistetään. Lisäksi `LandmarkRecognition`–ilmentymän luonnin yhteydessä luodaan ilmentymän privaatti paikallinen ilmentymä `Segmentation`–luokasta.

`LandmarkRecognition`–luokan `getRecognizedLandmarks()`–funktio on parametriton ja se palauttaa kaikki sillä hetkellä robotin ympäristöstä tunnistetut maamerkit ja niiden suuntakulmat suhteessa robotin ajosuuntaan. Funktio kutsuu `Segmentation`–luokan `getSegments()`–funtiota, joka palauttaa vektorin sisältäen Pair–tietorakenteita, joissa on kaikki tietyn väriset ja kokoiset bittikarttasegmentit ja niiden suuntakulmat suhteessa robotin kulkusuuntaan. Jokainen bittikarttasegmentti yritetään tunnistaa ShapeRecognitionin `string recognize(Bitmap)`–funktiolla, joka palauttaa koneen nimen, jos hahmo on tunnistettu kyseisellä koneella. Koneiden ja maamerkkien nimet vastaavat toisiaan, joten tuloksista voidaan muodostaa vektori, joka sisältää tunnistetut maamerkit ja niiden suuntakulmat suhteessa robotin kulkusuuntaan.

Johtuen siitä, että panoraama koostuu kuvista, jotka limittyvät reunoiltaan viereisiin kuviin, saattaa maamerkki näkyä kahdessa kuvassa saman aikaisesti. Tällöin samasta maamerkistä syntyy kaksi segmenttiä ja kaksi tunnistusta. Tämän johdosta duplikaatit, joiden kulmat suhteessa robottiin ovat lähekkäin (kulmien erotus $<$ `DIFF_ANGLE`), yhdistetään ja uudeksi kulmaksi tulee duplikaattien kulmien keskiarvo. Yhdistäminen on erityisen tärkeää, koska ylemmällä tasolla `Localization`–luokka poistaa duplikaatit kokonaan moniselitteisyyden takia so. kaksi tunnistusta samasta maamerkistä tarkoittaa Localizationille, että toinen tunnistuksista on väärä. `Localization` ei voi kuitenkaan tietää kumpi tunnistuksista on väärä, joten kummatkin täytyy hylätä. Jos duplikaattimaamerkkien erotuskulma on suurempi tai yhtä suuri kuin `DIFF_ANGLE`, `LandmarkRecognition` ei tee mitään ja vastuu siirtyy ylöspäin eli metodin kutsujalle.

Tulosvektori palautetaan `getRecognizedLandmarks()`–funktion kutsujalle. Funktio heittää eteenpäin mahdolliset `Segmentation`–luokan `getSegments()`–funtion heittämät `CaptureException`– ja `PanTiltException`–poikkeukset.

5.5 `ShapeRecognition`–kirjastorajapinta

`ShapeRecognition`–kirjastorajapinnan kaksi funktiota ovat `initMachines(vector<string>)` ja `recognize(Bitmap)`. Funktio `initMachines(vector<string>)` tallettaa koneiden nimet omaan tietorakenteeseensa. Funktio `recognize(Bitmap)` testaa parametrina annettua bittikarttasegmenttiä kaikilla `initMachines(vector<string>)`–funktiokutsulla annetuilla koneilla. Positiivisen tuloksen, koneen nimen, funktio palauttaa, jos yksi ja vain yksi kone tunnistaa annetun segmentin, muussa tapauksessa palautetaan tyhjä String–arvo.

5.6 `Segmentation`–luokka

`Segmentation` tarjoaa palveluna segmentoinnin metodina `std::vector<std::pair<Bitmap, NobelNS::Angle> > getSegments()`. `Segmentation` itse on hyvin yksinkertainen, sillä varsin paljon toiminnallisuudesta on muiden luokkien ja funktioiden palveluita, joita `Segmentation` käyttää. `Segmentation` käyttää `BasicSegmentation`–kirjastorajapintaa, ja `View`–oliota, jolta se saa `Panorama`–olioita.

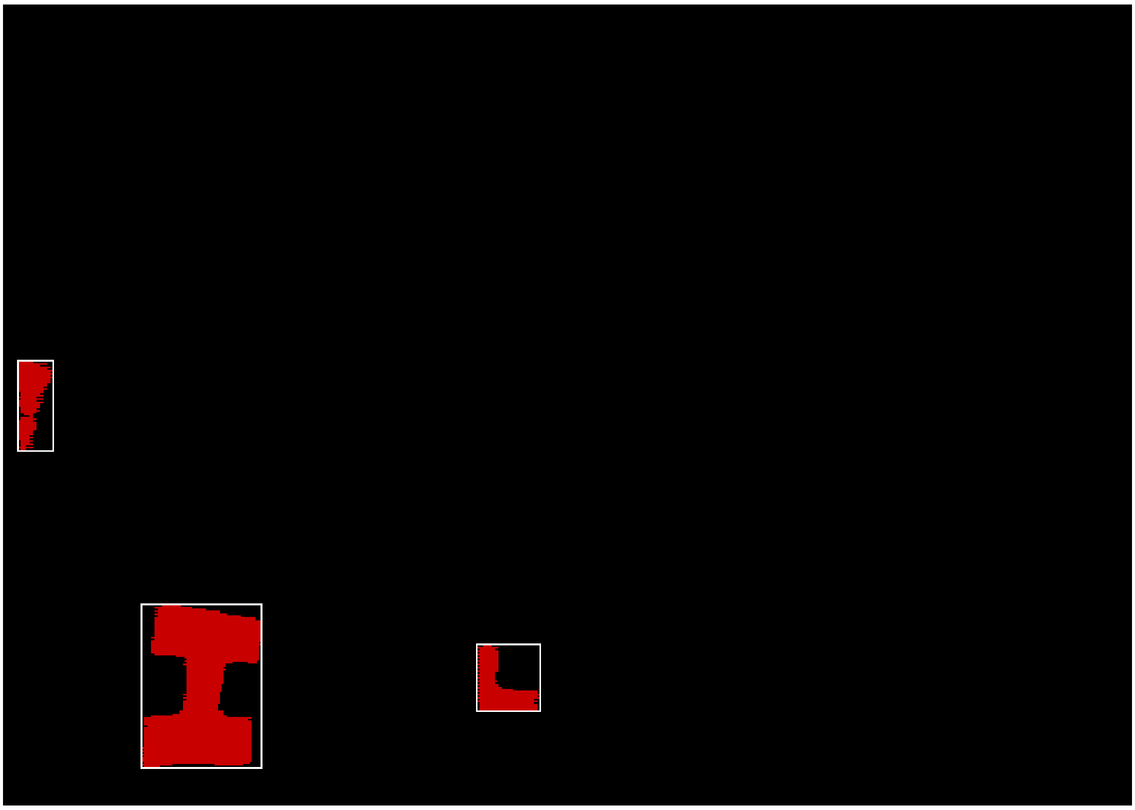
Kuva 5.6: Robotin kameran ottama kuva

Metodi `getSegments()` pyytää `View`–oliolta panoraaman (`Panorama`–olion tietojen perusteella tiedetään jokaiselle kuvalle kulma, jossa se on otettu). Jokaiselle panoraaman kuvalle se pyydetään BasicSegmentationilta segmentit ja segmenttien sijainnit kuvassa. Jokaiselle segmentille lasketaan sen suunta. Saadut (segmentti, kulma) –parit palautetaan taulukossa.

5.7 `BasicSegmentation`–kirjastorajapinta

`BasicSegmentation` on kirjastorajapinta kuvien matalan tason segmentoinnin toteuttavalle `HELVision`–kirjastolle. Lisäksi `BasicSegmentation` toteuttaa segmenttien normalisoinnin sopiviksi maamerkkitunnistusta varten.

`BasicSegmentation` alustetaan käyttöön metodilla `initSegmenter(stdd::string)`, jolle annetaan parametriksi segmentoinnin tarvitseman värimäärittelytiedoston nimi. Kuvat segmentoidaan metodilla
`std::vector< pair<Bitmap,double> getSegments(YUV422Image)`,
joka saa parametrinä yhden kuvan, (kuva 5.6), ja palauttaa sellaiset segmentit, (kuva 5.7), joiden alkuperäinen koko on riittävä. Segmentit on normalisoitu 32x32–kokoisiksi bittimatriiseiksi. Lisäksi kustakin segmentistä palautetaan sen keskipisteen suhteellinen sijainti alkuperäisessä kuvassa.



Kuva 5.7: Segmentoitu kuva

5.8 `View`–luokka

`View`–luokka huolehtii robotin ympäristön kuvaamisesta kameralla. Luokan ainoa julkinen metodi, `Panorama getPanorama()`, ottaa kameralla kahdeksan kuvan panoraaman kääntämällä kameraa. Panoraaman kuvat peittävät koko 360 asteen horisontin niin, että robotille ei jää yhtään sokeaa pistettä. Kameran kääntämiseen luokka tarvitsee `PanTilt.h`–tiedostossa määriteltyä `PanTilt`–kirjastorajapintaa ja itse kuvien ottamiseen `Camera.h`–tiedostossa määriteltyä `Camera`–kirjastorajapintaa.

`View`–luokan konstruktori on parametriton ja se ainoastaan määrittelee `View`–luokan ainoalle sisäiselle tietorakenteelle, `int`–tyyppiselle direction muuttujalle, arvon -1. Directionin ainoat arvot ovat -1 ja 1 ja se määrittelee käänetäänkö kameraa vasemmalta oikealle vai päinvastoin, kun panoraama kuvataan. Kameran kääntymissuunta vaihtuu joka kerta panoraaman luomisen jälkeen. Näin kameran ei tarvitse tehdä turhia liikkeitä ja säästetään ajassa.

`View`–luokan julkinen metodi, `getPanorama()`, palauttaa `Panorama`–luokan ilmentymän, joka sisältää kahdeksan kappaletta `YUV422Image`–luokan ilmentymää. Ennen jokaista kuvanottamista kameraa käännetään käyttäen PanTiltin `panTilt(Angle)`–funktiota. `Panorama`–ilmentymään sijoitetaan kuva `insertImage(YUV422Image, Angle)`–funktiolla ja kuva saadaan `Camera`–kirjastorajapinnassa määriteltyä `getImage()`–funktiota hyväksikäyttäen. Kuvan mukana panoraamaan talletetaan kuvanottokulma suhteessa robotin kulkusuuntaan. Lisäksi panoraamalle sijoitetaan kulma funktiollaä `setAngleOfView(Angle)`, joka on yhden kameran ottaman kuvan näkymän laajuus. Jokaisen panoraamanottokerran jälkeen `View`–luokan `direction`–muuttujan arvo muutetaan arvoksi `direction*(-1)` eli käytännössä kameran kääntämissuunta vaihdetaan. Funktio `getPanorama()` heittää eteenpäin `panTilt(Angle)`– ja `getImage()`–funktion heittämät `PanTiltException`– ja `CaptureException`–poikkeukset.

5.9 `Camera`–kirjastorajapinta

`Camera`–kirjastorajapinta on määritelty tiedostossa `Camera.h`. Se sisältää kolme staattista metodia, `initCapture()`, `releaseCapture()` ja `getImage()`. `Camera` on suorassa yhteydessä `Capture.h`–tiedostossa määriteltyyn kuvankaappauskortin hallintaan liittyvään rajapintaan. Funktio `initCapture()` alustaa kuvankaappauskortin. Se kutsuu Capture.h:n `videoConnect()`–funktiota määritellen kuvan resoluution ja väripaletin. Funktio `releaseCapture()` vapauttaa kortin kutsumalla `videoDisconnect()`–funktiota. Funktio getImage() luo `YUV422Image`–olion, sijoittaa sen sisäiseen tietorakenteeseen kuvan `videoCopyBuffer()`–funtiolla ja palauttaa sen funktion kutsujalle. Funktiot `initCapture()`– ja `getImage()`–funktio heittävät `CaptureException`–poikkeuksen, jos alustaminen tai kuvan kaappaus epäonnistuu.

5.10 `PanTilt`–kirjastorajapinta

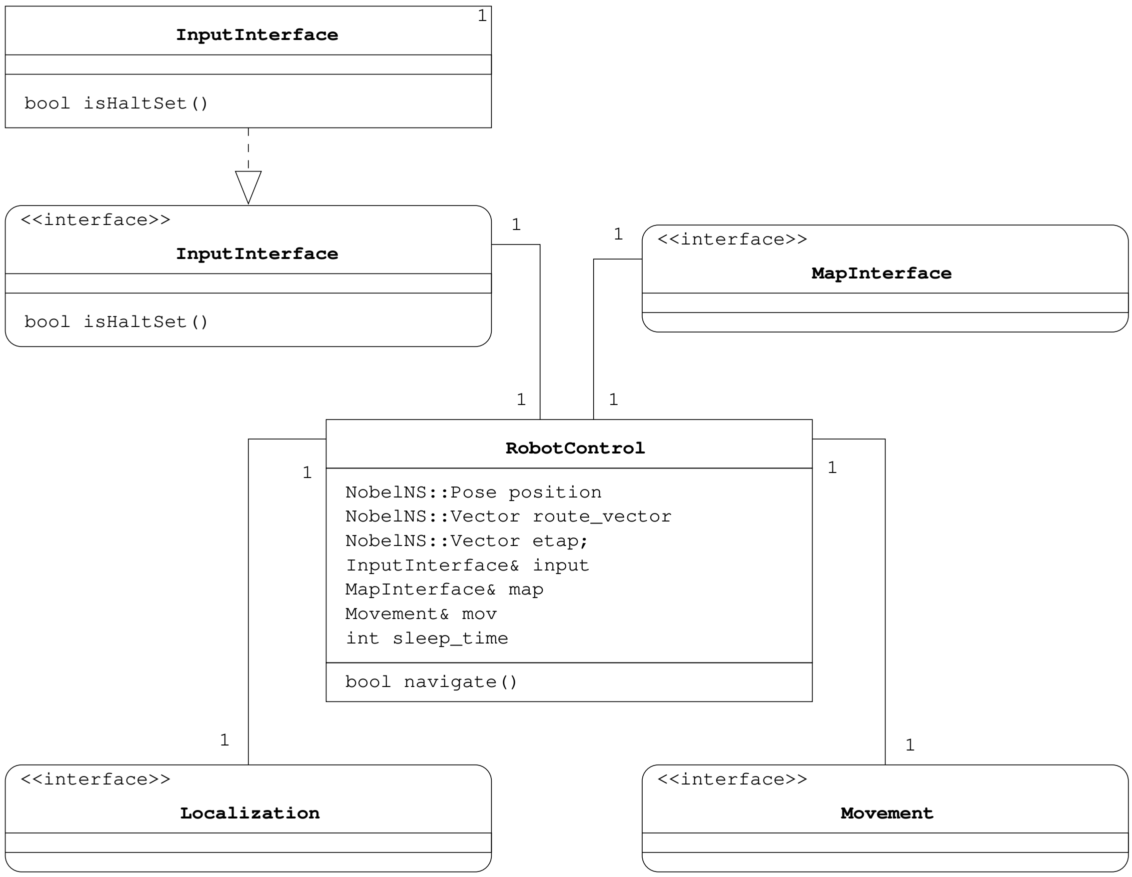
`PanTilt`–kirjastorajapinta on määritelty tiedostossa `PanTilt.h`. Sen ainoa metodi (staattinen) on `panTilt(Angle,Angle)`, joka kääntää kameran jalustan haluttuun vaaka– ja pystykulmaan. `PanTilt` käyttää hyväkseen `pantilt.h`–tiedostossa määriteltyä rajapintaa. Aluksi se alustaa pantilt–yksikön (`pt_init()`–funktio), määrittää nopeudet kääntymisen pysty– ja vaakasuuntaiselle nopeudelle, kääntää jalustaa pysty– (`pt_move_tilt_abs(int)`) ja vaakasuuntaan (`pt_move_pan_abs(int)`), odottaa liikkeen loppumista ja lopuksi vapauttaa pantilt–yksikön. Funktio `panTilt(Angle, Angle)` heittää `PanTiltException`–poikkeuksen, jos pantilt–yksikön initialisointi epäonnistuu.

5.11 `Panorama`–luokka

`Panorama`–luokka on talletusrakenne yhdelle kuvapanoraamalle. Se sisältää kuvat ja niiden kulmat suhteessa robottiin sekä kuvien optisen laajuuskulman. Kuvat ovat `YUV422Image`–olioita.

5.12 `Bitmap`–luokka

`Bitmap` on rasterikuvan talletusrakenne. Tällä erää sitä käytetään vain normalisoitujen bittisegmenttien tallettamiseen. `Bitmap` tukee kuitenkin myös suurempia värisyvyyksiä.



Kuva 6.8: `RobotControl`–osajärjestelmä

5.13 `YUV422Image`–luokka

`YUV422Image` on YUV–kuvien talletusrakenne. Kameran kuvankaappaukset tehdään YUV–muotoon, koska HELVision–segmentoija käyttää syötteenään tätä formaattia.

Kaapatut kuvat tallennetaan myös levylle YUV–muodossa, jolloin tiedostonimi on uyvy–päätteinen.

6 `RobotControl`–osajärjestelmä

RobotControl–osajärjestelmä koostuu yksinomaan RobotControl–luokasta. Luokka pyytää palveluita tarpeen mukaan muilta osajärjestelmiltä. Kaikki päätöksenteko robotin liikkumiseen ja navigointiin liittyen tapahtuu sen sisällä. Palvelukutsuja varten RobotControlilla on käytössään osoitin kuhunkin kolmen muun osajärjestelmän rajapintaolioon. RobotControl tarjoaa ainoastaan palvelun textttnavigate(), joka siis ohjaa robotin maalipisteeseen muiden osajärjestelmien palveluja hyväksikäyttäen.

RobotControlin konstruktori saa parametreina osoittimet MapInterface, Movement, `LocalizationInterface`– ja `InputInterface`ů–rajapintaluokat toteuttaviin olioihin. Osoittimet talletetaan `RobotControlRep`–olioon.

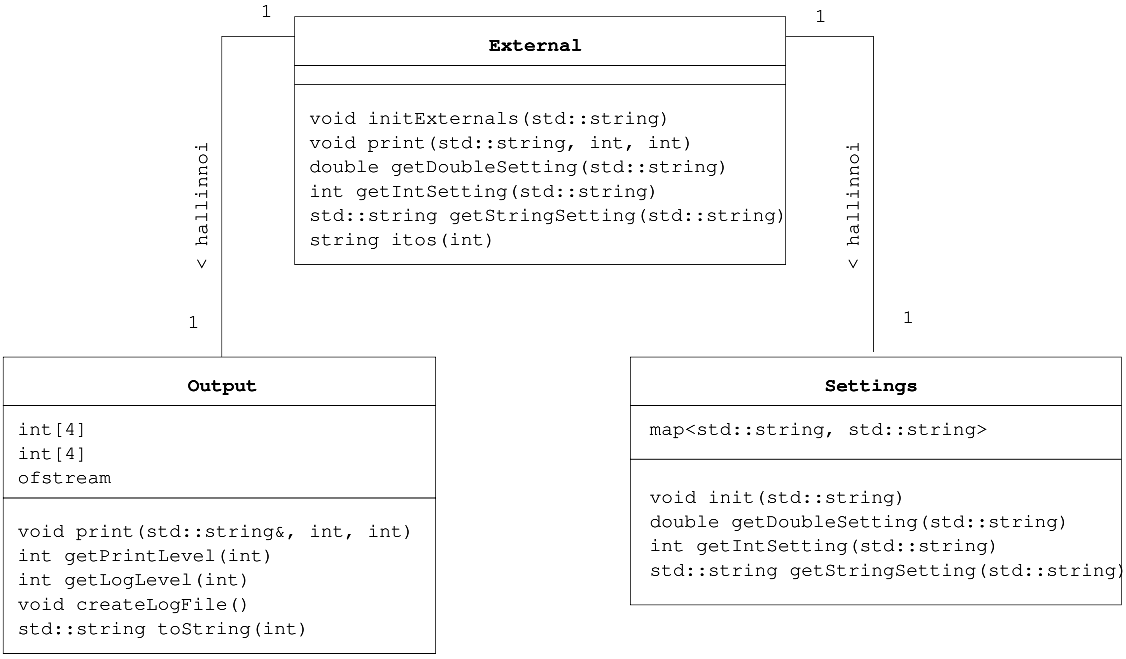
RobotControl toimii pääpiirteittäin seuraavasti:

- Kun navigointi aloitetaan, lähtöpisteessä suoritetaan ns. pakollinen paikanmääritys: Jos paikanmääritys epäonnistuu ensimmäisellä kerralla, suoritetaan satunnaisvektori ja uusi paikanmääritys `RANDOMVECTOR_COUNT`–kertaa. Jos paikanmääritys edelleen epäonnistuu, lopetetaan ohjelma.
- Kun on navigoitu ilman paikanmäärityksiä ajoparametrin `ETAP_LENGTH` verran, suoritetaan paikanmääritys (etappitarkistus). Jos paikanmääritys epäonnistuu, poikkeukseen ei kuitenkaan reagoida ja jatketaan navigointia normaalisti.
- Jos saadaan sonarihavainto, liike pysäytetään. Odotetaan parametrin `WAIT` ajan ja yritetään kerran liikkua uudelleen. Jos pystytään liikkumaan alle viisi senttiä, suoritetaan pakollinen paikanmääritys.
- Jos paikanmäärityksen antaman sijainnin mukaan robotti ei ole marginaalissa, on havaittu yllättävä este, jota ei ole merkitty karttaan. Yritetään kerran liikkua uudelleen ja jos pystytään liikkumaan vain alle viisi senttiä, ohjelma lopetetaan.
- Jos paikanmäärityksen antaman sijainnin mukaan robotti on marginaalissa, on havaittu karttaan merkitty este ja liikutaan suorinta reittiä ulos marginaalista.
- Jos robotti on maalissa tai käyttäjä on antanut lopetuskomennon, ohjelma lopetetaan.

6.1 Ongelmallisia tilanteita

Järjestelmä ei selviä seuraavista tilanteista:

- Robotti alkaa liikkua kesken reitin suorituksen satunnaisvektoreita ja ajautuu marginaalialueen yli umpikujaan, josta ei reitinlaskennan mukaan ole reittiä maaliin. Kun satunnaisvektorit on suoritettu ja paikanmääritys on onnistunut, ohjelma ei enää tarkista onko maalipisteeseen olemassa reittiä. Tämän jälkeen reitinlaskenta palauttaa vain nollavektoreita, ja ohjelma päättyy poikkeukseen, jossa ilmoitetaan, että "A bug was found! The routevector was zero."Satunnaisvektoreiden suoritus voi siis johtaa tilanteeseen, jossa maalipisteen saavuttaminen on mahdotonta. Robottihan ei suostu tarkoituksella liikkumaan ulos umpikujasta marginaalin läpi. Robotti voi joutua myös umpikujaan, jos paikanmääritys tunnistaa virheellisesti jonkin roskan ja antaa väärän sijainnin, joka on esteiden ympäröimä. Ongelmaa voi ratkoa tarkistamalla useammin routeExists-funktiolla, onko reittiä olemassa. Paikanmäärityksen antamaa sijaintia voi myös verrata odometrian mukaiseen sijaintiin. Jos sijainti poikkeaa epätavallisen paljon, tulos voidaan hylätä.
- Robotti voi joutua ikuiseen silmukkaan seuraavassa epätavallisessa tilanteessa: Saadaan sonarihavainto, odotetaan, yritetään liikkua uudelleen, liikkuminen epäonnistuu, suoritetaan paikanmääritys, paikanmääritys epäonnistuu esteen läheisyydessä, liikutaan satunnaisvektoreita ja yritetään paikanmääritystä uudelleen, paikanmääritys onnistuu kauempana esteestä, liikutaan reitin suuntaisesti jälleen



Kuva 7.9: `External`–osajärjestelmä

kohti estettä, joka ei siirry sekä saadaan jäälleen sonarihavainto, ja silmukka alkaa alusta. Siis este on niin iso, että paikanmääritys epäonnistuu aina sen läheisyydessä ja este peittää reitin maaliin.

- Jos robotti on marginaalialueella ja se saa sonarihavainnon, robotti aina olettaa yksinkertaisesti havainneensa karttaan merkityn esteen. Tällöin robotti pyrkii liikkumaan lyhintä reittiä ulos marginaalista. Ongelmia seuraa, jos este peittää suoran reitin ulos marginaalista. Tällöin robotti joutuu ikuiseen silmukkaan eikä reagoikaan yllättävään esteeseen järkevästi. Ongelman ratkaisuehdotus: `BlockException`–poikkeus sisältää tiedon havaitun esteen suhteellisesta sijainnista. Jos Mappiin lisätään funktio, jolla voi kysyä, onko kyseisessä pisteessä este, RobotControl voi päätellä oikein, onko este merkitty karttaan vai onko kyseessä yllättävä este.

- `RobotControl` saattaa virheellisesti keskeyttää ajon liikkuessaan ahtaista oviaukoista ja ilmoittaa havainneensa yllättävän esteen, vaikka kyseessä onkin oven karmi. Ongelman syy: Että robotti voi mahtua oviaukoista, on marginaali asetettava pienemmäksi kuin pysäytysetäisyys, jonka järkevä alaraja on 20 cm. RobotControl päättelee virheellisesti havainneensa yllättävän esteen ja lopettaa navigoinnin, koska robotti ei ole marginaalissa. Kyseessä on siis sama ongelma kuin edellisessä kohdassa. Lisäksi pelkkä marginaalin asettaminen pienemmäksi kuin pysäytysetäisyys voi aiheuttaa virheitä. Reitinlaskenta saattaa suunnitella reitit siten, että saadaan turhia sonarihavaintoja.

7 `External`–pakkaus

`External`–pakkaus tarjoaa olioille tulostus– ja asetuspalvelut.

7.1 `External`–luokka

Itse `External`–luokka on staattinen, kaikkien tuntema "manageri", jonka kautta pääsee käsiksi `Settings`– ja `Output`–palveluihin. Näin muille olioille ei tarvitse välittää viitteitä `Settings`– ja `Output`–olioihin missään vaiheessa.

`Settings` ja `Output` ovat Externalin staattisia muuttujia. Nämä alustetaan kutsumalla Externalin metodia `initExternals(string)`, joka ottaa parametrina ajotiedoston nimen.

7.2 `Settings`–luokka

`Settings` lukee ja jäsentää ajotiedoston. Muut oliot kysyvät Settingsiltä (Externalin kautta) ajotiedostossa määriteltyjen asetusten arvoja.

`Settings` kerää asetukset `map`–säiliöön, jossa avaimena on asetuksen tunniste ja arvona asetuksen arvo merkkijonona. Kutsuttaessa `string getStringSetting(string)`–metodia palautetaan asetuksen merkkijonoarvo, jos annettu asetustunniste löytyy, muuten heitetään poikkeus. Metodit `int getIntSetting(string)` ja `double getDoubleSetting(string)` toimivat muuten samoin, mutta niiden palautusarvo lasketaan merkkijonoarvosta käyttämällä `atoi`/`atof`–funktioita.

7.3 `Output`–luokka

Tulostaa näytölle ja kirjoittaa lokiin olioiden tulosteet annettujen tulostusasetusten mukaan. Ajotiedostossa on määritelty kullekin osajärjestelmälle tulostustaso. Tämän mukaan olioiden tulostuspyynnöt ohjataan lokiin ja/tai monitorille tai hylätään. Tulostustasoja on neljä, joista numero 0 vastaa ns. debuggaus–tasoa, joka sisältää ohjelmiston testaukseen liittyvää tulostetta. Tason 0 tulosteita ei tulosteta ruudulle valmiissa ohjelmistossa. Taso 1 on luokka–taso eli sisältää luokkien sisäistä tulostusta. Taso 2 on askeleen korkeampaa tulostetta ja on tarkoitettu osajärjestelmille. Taso 3 on korkein tulostustaso ja sisältää kaikista yleisluonteisimmat tulosteet. Se on tarkoitettu lähinnä `RobotControl`–luokalle.

Luokka luo uuden lokitiedoston sillä nimellä, joka on määritelty ajotiedostoon. Nimen perään lisätään .log–pääte. Jos määritelty nimi on jo käytössä, lisätään nimen perään juokseva numero kunnes ensimmäinen vapaa tiedostonnimi löytyy.

`Output`–olion kautta tulostettaessa täytyy kutsujan antaa parametrina kutsujan osajärjestelmätunnus ja haluttu tulostustaso, jotta Output osaisi tulostaa tai olla tulostamatta.

8 `Input`–pakkaus

`Input`–pakkaus sisältää luokat `InputInterface` ja `Input`. Luokka `Input` on toteutettu käyttäen Handle–Body–menetelmää. Inputin tarkoituksena on lukea käyttäjän antama lopetuskomento.

8.1 `InputInterface`–rajapintaluokka

`InputInterface` on abstrakti luokka, joka määrittelee operaatiot, jotka syötteitä lukevan olion on vähintään toteutettava.

8.2 `Input`–luokka

`Input` toteuttaa `InputInterface`–luokan. Syöte luetaan stdinistä. Inputin konstruktori avaa sen ei-blokkaavana, ettei koko prosessi blokkaannu stdiniä luettaessa, jos käyttäjä ei ole antanut lainkaan syötteitä.

9 `Main` - järjestelmän pääohjelma

Pääohjelma sisältää robotin hätäpysäytystoiminnon törmäysten estämiseksi. Hätäpysäytys on toteutettu tekemällä oma käsittelijä signaalille `SIGINT`. Käsittelijä pysäyttää robotin liikkeen välittömästi ja terminoi koko ohjelman suorituksen.

Pääohjelmassa alustetaan myös yllättäville poikkeuksille (eli poikkeuksille, joita funktio on luvannut olla heittämättä, mutta heittää silti) oma käsittelijä, joka tulostaa järkevän virheilmoituksen (erityisen hyödyllinen, jos poikkeus on `std::exception` tai sen aliluokka). Tästä käsittelijästä ei sinänsä ole paljoa hyötyä, sillä ohjelmassa on pyritty takaamaan, ettei yllättäviä poikkeuksia esiinny.

Pääohjelmassa myös luodaan `Movement`– ja `Map`–osajärjestelmiä ohjaavien olioiden konstruktoreiden tarvitsemat oliot sekä myös itse ohjaavat oliot. Lisäksi luodaan `Localization`, `Input` ja lopuksi `RobotControl`.

Kirjallisuutta

[1] M. Weiss. *Data Structures and Algorithm Analysis.* The Benjamin/Cummings Publishing Company, USA, 1995.

[2] Charles Cohen and Frank V. Koss. A comprehensive study of three object triangulation. In *Proc. of the 1993 SPIE Conf. on Mobile Robots*, pages 95–106, November 1993.

[3] Ilkka Forsblom, Risto-Matti Kääriäinen, Jussi Lahti, Maarit Saaresto, and Jarkko Salminen. Vaatimusdokumentti. February 2001.