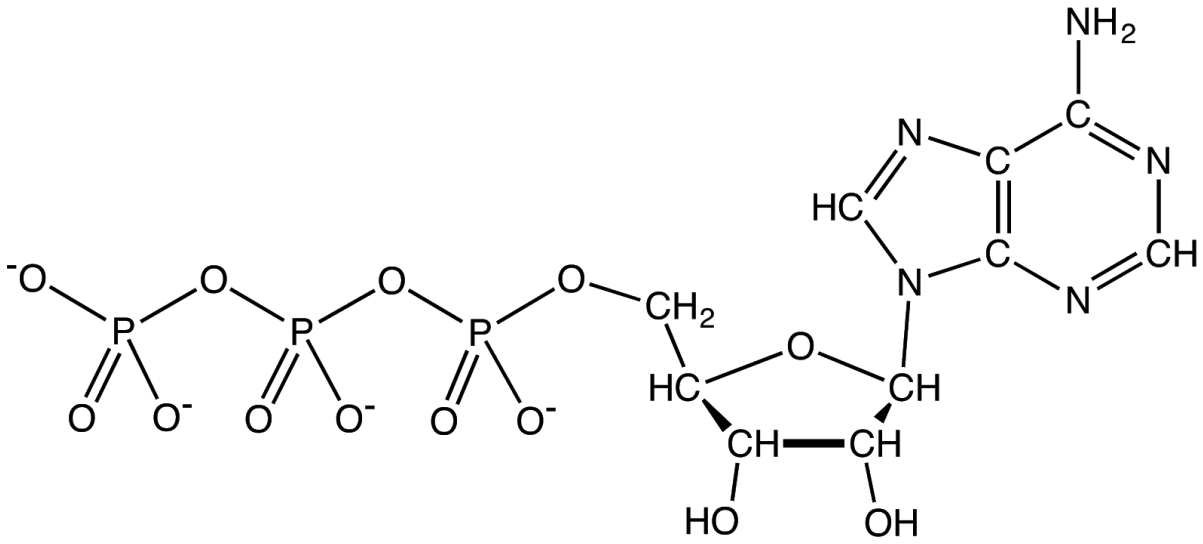
TESTRAPPORT
THE LEMONATOR



An ATP (Adenosine triphosphate) molecule.

BESTANDSINFORMATIE

02-06-2019

Versie 1.0

OPGESTELD DOOR

Wiebe van Breukelen: 1696056

Mike Hilhorst: 167609

Peter Bonnema: 1584656

Inhoudsopgave

Inleiding

In dit testrapport wordt beschreven hoe er een bepaalde mate van kwaliteit van de Lemonator is behaald is de hand van de uitgevoerde (en niet uitgevoerde) tests.
Ook beschrijft het rapport hoe deze tests zijn uitgevoerd en met welke reden. Tenslotte bevat dit document een motivering voor afwijkingen van het eerder opgestelde testplan.

Rapportage: infrastructuur

Dit hoofdstuk licht alle infrastructuur toe die relevant is voor de geautomatiseerde testen. Dit is van belang om de reproduceerbaarheid van het testen te vergroten.

Continuous integration

Er is gebruik gemaakt van Continuous Integration (CI). Dit is tijdsbesparend en garandeert dat code aanpassingen altijd getest worden. CI biedt als grootste voordeel dat alle branches worden gebouwd en getest op een build-server na elke git push. Dit biedt direct feedback over eventuele problemen met de code-implementatie.

Tooling

De applicatie is ontwikkeld met Python 3.7.2 32-bits en C++ 17, TDM-GCC-32 (32-bits GCC `5.1.0-3` compiler). Deze versies zijn compatible met `hwlib`.

Voor versiebeheer is gekozen voor git in combinatie met GitHub[1]. GitHub is een veelgebruikt platform voor onder andere kleinere projecten zoals de onze en biedt voldoende flexibiliteit en mogelijkheden.

Travis-CI wordt gebruikt als CI tool. Travis integreert goed met GitHub en is relatief eenvoudig op te zetten. Jobs en omgevingen worden gedefineerd in een file genaamt `.travis.yml`. Hier wordt vrijwel alle configuratie in geplaatst. Travis levert ook een badge met de build-status. Deze is geplaatst is op de readme van het project.

De Travis-CI build status in te zien op de volgende URL:
https://travis-ci.com/PBonnema/lemonator

De Python `coverage` module wordt gebruikt om de code coverage voor Python te meten. Dit rapport wordt ook gegenereerd door de Travis build job. `Coveralls` wordt gebruikt om de rapporten te visualiseren. De build job stuurt het gegenereerde coverage rapport aan het eind naar coveralls. Coveralls levert ook een badge met het coverage percentage. Deze geplaatst is op de readme van het project.
De coverage rapporten zijn in te zien op de volgende URL:
https://coveralls.io/github/PBonnema/lemonator

[1] De GitHub repository is te vinden op: https://github.com/PBonnema/lemonator/

Travis-CI Configuration

Travis-CI start een build job als er gepushed wordt naar een branch op GitHub en als de file .travis.yml aanwezig is in die branch. Deze file definieert een build job.
Het bestand is aanwezig in iedere branch. In de configuratie op Travis-CI wordt aangeven welke repositories gemonitord moeten worden.

De job runt in een Linux Xenial OS. Dit is nodig om Python versie => `3.7` uit te voeren. Een job bestaat uit verschillende stappen. Hiervan worden alleen de stappen `install`, `script` en `after_success` gebruikt. Deze worden nu toegelicht:

Install

Hier worden alle benodigde Python Pip pakketten geïnstalleerd. Deze staan gedefinieerd in de requirements.txt bestand in de repository.
Er zijn tools, zoals `PipReqs`, die aan de hand van alle module imports in de code die gedaan worden automatisch een `requirements.txt` bestand kan genereren zodat deze daarna door Pip kan worden gebruikt om de benodigde dependencies te installeren.
Dit gaf echter problemen waarbij de zelf-geschreven python modules die geïmporteerd werden, inclusief de files van de simulator, ook gezien als dependencies. Daarom is afstand gedaan van een dergelijke tool en worden nu de dependencies handmatig bijgehouden in de `requirements.txt` file.

Script

Tijdens deze stap worden alle tests uitgevoerd. Het gekozen unit test framework is de standaard unit test module. Er wordt gebruik gemaakt van test discovery die automatisch alle uit te voeren testen in alle directories ontdekt en uitvoert. Ook wordt er tegelijk een code coverage rapport gegenereerd met de coverage module. Niet alleen de coverage van de regels wordt gemeten maar ook de coverage van de branches (if-statements etc). Dit is een ingebouwde feature in de coverage module.

After success

Uiteindelijk wordt het rapport opgestuurd naar `Coveralls`. Hiervoor wordt de Python module *coveralls* gebruikt. Dit is een officieuze Python library die interacteert met de web API die *coveralls* aanbiedt op hun servers om het rapport op te sturen.

Branching strategy

Bij CI hoort ook een branching strategie. Deze strategie houdt in dat er een centrale master branch en een centrale development branch is. De master branch bevat in principe alleen stabiele versies van de software. De development branch bevat meer recente en eventueel onstabielere versies. Af en toe wordt de development branch in de master branch gemerged als deze op een zeker moment als stabiel wordt aangemerkt. Er wordt op geen enkel ander moment en vanaf geen enkele andere branch gemerged naar de master branch.

Wanneer er een nieuwe feature wordt gebouwd of een bug wordt opgelost, wordt dit gedaan in een aparte feature branch. Deze wordt van de development branch afgetakt. Zodra de

feature klaar is, moet de developer zelf een pull request aanmaken om de feature terug naar de development branch te mergen. Op deze manier kunnen verschillende features tegelijk worden geïmplementeerd zonder elkaar in de weg te zitten.

Bij iedere pull request wordt het hypothetische resultaat van de merge eerst gebouwd door Travis voordat deze definitief wordt gemerged. Hiermee kan op voorhand al problemen worden ontdekt. Ook worden voor deze zogenaamde *integration builds* alle tests uitgevoerd en de coverage bepaald. Daarnaast stellen pull requests het team in staat om code reviews uit te voeren.

Al deze voordelen verhogen de kwaliteit en betrouwbaarheid van de software.

Mappen structuur

De mappenstructuur van de repository is als volgt:
.\simulator
Deze map bevat een kopie van de inhoud van de simulator map van the lemonator simulator repository.

- `.\hw_dummy`
- `.\hw_server`
- `.\library`
- `.\pc_cpp`
- `.\pc_python`

Deze folders zijn gekopieerd van de `vkatp-lemonator` repository.

- `.\test`: Bevat alle test files
- `.\test\system_tests`: Bevat alle system tests.
- `.\test\unit_tests`: Bevat alle unit tests.

Testing

De Python tests zijn ontwikkeld met de ingebouwde `unittest` module en de bijbehorende unittest.mock module. Test discovery wordt op twee momenten gedurende development ingezet. De eerste is tijdens de Travis build job. De tweede is lokaal op ontwikkelmachines tijdens het normale ontwikkelproces. De test discovery tool kijkt alleen naar mappen die een package zijn (dus een `__init__.py` file bevatten). Deze is daarom aanwezig in de `.\tests` map en alle mappen binnen die map.
.
De tests moeten de Python controller module en de modules van de simulator kunnen vinden wanneer deze worden geïmporteerd. Ook moeten de modules elkaar vinden wanneer deze elkaar importeren. Daarom wordt de map `.\simulator` toegevoegd aan `PYTHONPATH` in de `__init__.py` bestanden. Dit `__init__.py` bestand wordt namelijk uitgevoerd wanneer de test discovery tool de tests package importeert.

Reproduceerbaarheid van testen

Er wordt gebruik gemaakt geautomatiseerde testen, zoals bij het kopje *Rapportage: infrastructuur* beschreven.
Deze tests maken gebruik van een nieuwe instantie van de simulator en de controller. Hierdoor beïnvloeden de testen elkaar niet.

Omdat we bij elke test een nieuwe instantie van de simulator gebruiken wordt in elke test, een testsituatie opgesteld. De simulator heeft basiswaarden. Hier is meer over te lezen onder het kopje “Situatie van de simulator”. Elke test maakt gebruik van asserts. Hier wordt een variabele vergeleken met een ‘hardcoded’ (vaste) waarde. Wanneer een waarde afwijkt van het gewenste resultaat faalt de test.

Hierdoor hoeft een tester, na het opzetten van de testomgeving, alleen de testen uit te voeren. Houdt gedachte dat de simulator basis waardes heeft, dus alleen de waardes die in de test worden aangepast wijken van deze *basis* waarde af. Hieronder een voorbeeld van een test:

```
def test_controller_start_press(self):
    # Press the A/start.
    self.keypad.push('A')

    # Update the simulator once.
    updateSim(self.sim)

    # Test the state.
    self.assertEqual(
        self.ctl.state, CustomController.States.WAITING_FOR_CUP)
```

Hier wordt een keypress gesimuleerd. Wordt één simulator update cyclus uitgevoerd en er wordt getest of de state is veranderd naar de gewenste state.

Startwaarden Lemonator

De lemonator heeft standaard waarden waar rekening mee gehouden moet worden, deze waardes zijn constant tussen testen. Dit zijn deze waarde:

Object	Status
Pump Siroop	Uit
Pump Water	Uit
Valve Siroop	Dicht
Valve Water	Dicht
Cup	Niet aanwezig

Vat siroop inhoud	2000 ml
Vat water inhoud	2000 ml
Display buffer	Leeg
Keyboard buffer	Leeg
Fouten	Geen (fault: `NONE`)
State	Wachten (state: `IDLE`)

Test verantwoording

De tests zijn verdeeld in verschillende categorieën. Ten eerste is er een onderscheid gemaakt tussen system tests, die functionele dingen testen, en unit tests, die de verschillende software units testen. Er zijn geen integratie tests meer geschreven wegens tijdsgebrek.

Unit tests

Er zijn unit tests van de SimulatorInterface. Deze tests verifiëren dat de interface naar behoren werkt. De simulator wordt volledig weggemocked waardoor het mogelijk is om te verifiëren dat de juiste dingen worden aangeroepen. Dit is belangrijk omdat, als hier bugs in zitten, dan kan dat in potentie heel moeilijk op te sporen problemen in de controller opleveren. Als de interface incorrect is, lijkt het namelijk alsof de controller zelf incorrect is.

Er zijn geen unit tests geschreven van de simulator. Tijdens de development bleek de simulator goed te werken. Wel is het zo dat er op een aantal vlakken duidelijke verschillen bleken te zijn tussen de hardware en de simulator. Dit hebben we in stand gehouden omdat dit het functioneren van de controller niet hinderde.

System tests

De system tests zijn als volgt onderverdeeld.

Main menu

Deze tests gaan over of de initiële staat van de controller correct is, of dat alle hardware onderdelen in de goede stand staan en of dat de juiste dingen van het hoofdmenu op het scherm staan.
De hardware die gechecked wordt is beide pompen, beide valves en de heater.

Het is belangrijk om controle te houden over de beginstaat van de software en van de hardware zodat je altijd weet in welke situatie het systeem zich bevindt. Dan is er minder code nodig die elke keer controleert wat de huidige staat is van het systeem. Dit reduceert ook de kans op fouten in de software.

Dispense menu

Deze tests gaan over het menu dat je krijgt als je op ‘A’ drukt. Dit is het menu waarin je kan instellen hoeveel limonade je wil en hoe sterk.
Deze tests verifiëren of dat de tekst op het scherm correct is, dat er correcte waarden ingevuld kunnen worden en dat die ook door de controller worden gebruikt. Ook wordt er gecontroleerd dat er geen incorrecte waarden kunnen worden ingevoerd en dat dan de juiste foutmelding op het scherm komt.
Deze functionaliteit is vrij belangrijk omdat het invoeren van de temperatuur behoort tot de kern van het systeem. Het verifiëren dat er geen incorrecte waarden kunnen worden ingevoerd is iets minder belangrijk omdat dit alleen bijdraagt aan de gebruiksvriendelijkheid maar zonder deze checks zou het systeem ook werken. Dit maakt de tests ook minder kritisch.

Heater menu

Deze tests gaan over het menu dat je krijgt als je op ‘C’ drukt. Dit is het menu waarin je de gewenste temperatuur va de limonade kan instellen.
Deze tests verifiëren of dat de tekst op het scherm correct is, dat er correcte waarden ingevuld kunnen worden en dat die ook door de controller worden gebruikt. Ook wordt er gecontroleerd dat er geen incorrecte waarden kunnen worden ingevoerd en dat dan de juiste foutmelding op het scherm komt.
Deze functionaliteit is vrij belangrijk omdat het invoeren van de temperatuur behoort tot de kern van het systeem. Het verifiëren dat er geen incorrecte waarden kunnen worden ingevoerd is iets minder belangrijk omdat dit alleen bijdraagt aan de gebruiksvriendelijkheid maar zonder deze checks zou het systeem ook werken. Dit maakt de tests ook minder kritisch.
Ook zijn deze tests iets minder belangrijk omdat de heater functionaliteit optioneel was.

Stats menu

Deze tests testen het stats menu waarin te zien is hoeveel water en siroop er nog in de tonnen zit. Deze tests verifiëren of dat de juiste dingen op het scherm komen te staan en dat de terug (#) knop werkt.

Dispensing

Deze tests gaan over het proces waarbij er een drankje wordt geschonken voor de gebruiker.
Eerst wordt de happy-flow geverifieerd. Er wordt gechecked dat de inhoud van de vessels met water en siroop ook echt dalen en dat de beker gevuld wordt.
Daarna wordt de unhappy-flow getest. Dit houdt in dat het gedetecteerd wordt wanneer de beker wordt verwijderd en dat er dan een melding op het scherm komt te staan.
Er zijn ook veel tests die nagaan of dat de pompen, valves en LED’s goed worden aangestuurd.
Dit hele proces verifiëren is heel belangrijk. Dit is namelijk de kern van functionaliteit.

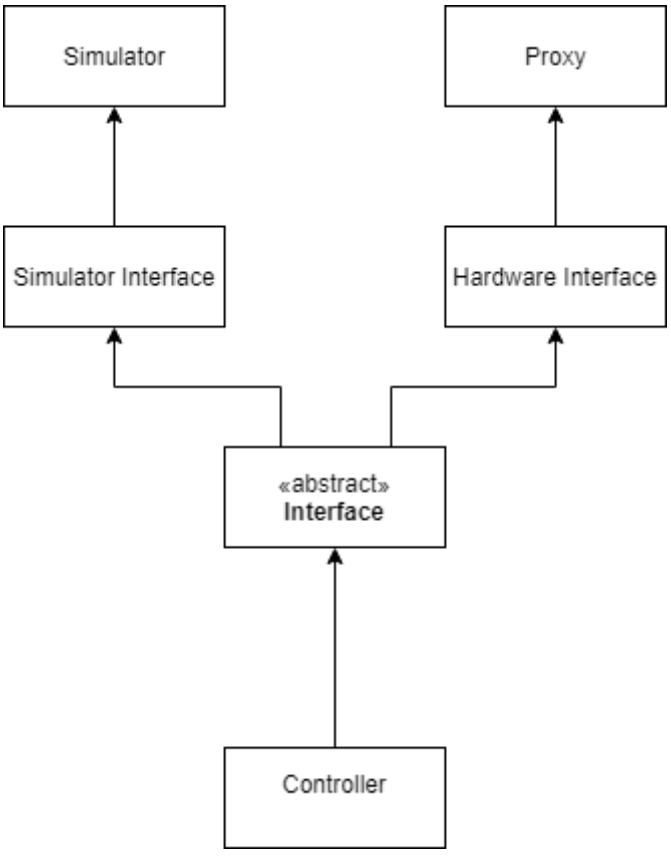
Uitgevoerde Testen

De uitgevoerde code testen zijn in te zien op GitHub:
https://github.com/PBonnema/lemonator/tree/dev/test

Alle testen moet slagen om te voldoen aan de test eis.

Unit tests

Binnen de repository is de map *tests/unit_tests* aangemaakt. Binnen deze map zijn de unit tests gedefinieerd. Er is besloten om tests te schrijven voor de *Simulator Interface*. Dit is de interface tussen de simulator en de controller. Zie het onderstaande diagram.



Alle tests zijn ingericht om de actuatoren en sensoren, ook wel modules genoemd, van de Lemonator te testen. De tests zijn hierbij ingericht als white-box. Belangrijk om te vermelden is dat deze tests niet met een interne state werken. Het testen met een interne state is juist van belang bij een systeem test. Ook laten we de implementatie van de Simulator buiten beschouwing, dit is in ons geval dus een black-box.

Elke module bevat tenminste de volgende tests:

- Het correct initialiseren van de module: `test_can_create`.

- Een test per functie, welke gedeclareerd zijn binnen een module.

De modules en bijbehorende unit testen zijn te vinden in de volgende tabel:

Module	**Getest**
(Abstract) Effector	• Inschakelen en uitschakelen effector. Effector wordt ingeschakeld en er wordt gecontroleerd of de functie werkelijk is aangeroepen. • Uitlezen van huidige effector status, uit of aan. Effector wordt aan- of uitgezet en huidige waarde wordt uitgelezen.
LED's	• Testen van schakelen (toggle). Er wordt getest of de functie daadwerkelijk is aangeroepen.
LCD	• `pushString`: schrijven van waarde naar het display. Er wordt getest of de functie daadwerkelijk is aangeroepen.
(Abstract) Sensor	• Lezen van sensor; er wordt een dummy waarde aan het object aangeboden. Uitgelezen waarde wordt achteraf vergeleken met ingevoerde waarde. De sensor zelf bevat geen waarde bewerkingen; de pure leeswaarde wordt teruggeven. Het testen van interne berekeningen is dat ook niet van toepassing.
PresenceSensor	• Lezen van sensor; er wordt een dummy waarde aan het object aangeboden, in dit geval `False` (geen beker aanwezig). `PresenceSensor().readValue()` dient aangeroepen te zijn en de huidige waarde terug te geven, in dit geval *False*.
Keypad	• Keypad wordt gelezen door middel van een `Keypad.pop()` aanroep. Functie `pop` dient aangeroepen te worden. • Keypad wordt gelezen door middel van een `Keypad.popAll()` aanroep. In het geval van deze test is er geen invoer vanuit de gebruiker, alleen het `NULL` karakter (`0x00`) is aanwezig. Dit is namelijk ook het geval bij de proxy implementatie wanneer er geen nieuwe toets acties beschikbaar zijn. Er wordt vergeleken of de return-waarde gelijk is aan een lege string. • Keypad wordt uitgelezen. In de buffer staat `abc\x00`. De return-waarde dient `'abc'` te zijn. • Keypad wordt uitgelezen. In de buffer staat `a\x00abc\x00`. De return-waarde dient `'a'` te zijn.

Systeem testen

De testen zijn opgedeeld in 12 groepen/blokken per groep wordt er een uitleg gegeven en een selectie van de bijbehorende test worden beschreven.

Defaults

In deze groep worden de test **standaardwaarden** getest. Het is zeer belangrijk om dit te doen, omdat hiermee afgedwongen wordt dat alle testen vanaf dezelfde gewenste basis werken. Als hierin een afwijking tussen testen zou bestaan kunnen de testen ook gaan afwijken zonder een duidelijke reden. Dit is niet wenselijk.

De controller, een state-machine voor de aansturing van de Lemonator, maakt gebruik van standaardwaarden. Neem als bijvoorbeeld; de pompen staan uit en een beker is aanwezig.

Hier wordt getest of de actuatoren en effectoren de gewenste basiswaarden hebben.

```
def test_controller_init_actuator_and_effector_state(self):
```

Hier wordt getest of de variabelen de gewenste basiswaarden hebben.

```
def test_controller_init_state_vars(self):
```

Hier wordt getest of het LCD goed wordt geleegd.

```
def test_lcd_is_clear(self):
```

Deze test valt ook in de groep *faults*, hier wordt getest of er voor de machine initialisatie, dus direct na het aanmaken van de state-machine, geen fouten in het systeem registreert staan.

```
def test_controller_init_fault_state(self):
```

Zoals eerder beschreven deze test zorgen ervoor dat we zeker zijn dat alle tests hierna de gewenste begin waardes hebben.

Dispense logic

In deze groep wordt de logica van het vloeistofpompen getest. Dit is kernfunctionaliteit van de Lemonator. Er wordt bijvoorbeeld getest of pompen van water goed gaat wanneer de siroop pomp uit staat en de water sluis open staat.

De functies die het pompen van water en siroop afhandelen, ondersteunen het tegelijkertijd pompen van water en siroop. De functie kan ook forceren dat er alleen water of alleen siroop gepompt wordt. Dit wordt aangegeven met `only_one_can_be_on`, zoals de naam suggereert, zal deze notatie alleen gebruikt worden wanneer er alleen water of siroop gepompt moet worden.

Hier wordt de waterpomp logica getest `PompA` uit en `ValveA` dicht

```
def test_start_water_pump_only_one_can_be_on_true(self):
```

`PompA` aan en `ValveA` dicht

```
def test_start_water_pump_pumpA_already_on_only_one_can_be_on_true(self):
```

`PompA` uit en `ValveA` open

```
def test_start_water_pump_valveA_already_on_only_one_can_be_on_true(self):
```

`PompA` aan, `ValveA` dicht *en* kan meer dan een pomp aan

```
def test_start_water_pump_pumpA_already_on_only_one_can_be_on_false(self):
```

Kan meer dan een pomp aan

```
def test_start_water_pump_only_one_can_be_on_false(self):
```

Hier wordt de siroop pomp logica getest

```
def test_start_syrup_pump_only_one_can_be_on_true(self):
```

`PumpA` aan

```
def test_start_syrup_pump_pumpA_already_on_only_one_can_be_on_true(self):
```

`PumpB` aan

```
def test_start_syrup_pump_pumpB_already_on_only_one_can_be_on_true(self):
```

`ValveB` open

```
def test_start_syrup_pump_valveB_already_on_only_one_can_be_on_true(self):
```

Kan meer dan een pomp aan

```
def test_start_syrup_pump_only_one_can_be_on_false(self):
```

Hier wordt het stoppen van de pompen getest.

```
def test_shutfluids_pumpA_on_pumpB_on(self):
```

Omdat het aansturen van de pompen zo belangrijk is voor het systeem worden deze uitgebreid getest. Deze testen bevestigen dat de kernfucntionatiet van het systeem werk. Daarom dragen deze testen veel bij aan het systeem.

Invoer

In deze groep wordt de afhandeling van verkeerde gebruiker interactie getest. Bijvoorbeeld als iemand “per ongeluk” 100 liter invult in plaats van 100 milliliter of de cup verwijderd tijdens het dispencen, dan moet het systeem een fout geven en het niet daadwerkelijk uitvoeren.

Hier wordt getest of de afhandeling van te lage hoeveelheden van vloeistoffen werkt.

```
def test_controller_select_zero_amount(self):
```

Hier wordt getest of the afhandeling van te hoge temperaturen werkt.

```
def test_controller_select_too_high_heater(self):
```

Hier wordt getest of het verwijderen van een beker goed wordt afgehandeld.

```
def test_controller_dispensing_fault_cup_removed(self):
```

Hier wordt getest of de afhandeling van te hoge hoeveelheden water werkt.

```
def test_controller_select_wrong_amount_water(self):
```

Hier wordt getest of de afhandeling van te hoge hoeveelheden siroop werkt.

```
def test_controller_select_wrong_amount_syrup(self):
```

Het belang van deze testen is dat er bevestigd wordt dat het systeem goed kan omgaan met verkeerde invoer van de gebruikers en niet kapot loopt.

LED's

In deze groep wordt de aansturing van de LED's getest. De LED's zijn niet erg van belang in het systeem. Als er per ongeluk een verkeerde led aan gaat draait de rest van het systeem door. Het is echter wel vereiste functionaliteit van het systeem. Wij achten het dus van belang om deze te testen.

Oorspronkelijk was de gedachte dat de Lemonator zes aanstuurbare LED's had, aangezien de simulator ook zes aanstuurbare LED's tot zijn beschikking had. Na besturing van de hardware bleek echter dat er twee LED's aanstuurbaar waren. De testen zijn wel zo ingericht dat deze alle zes LED's tests. Dit is dus een verschil met de proxy aansturing.

De LED's worden beïnvloed door de status van de actuatoren en effectoren, namelijk: `PumpA` (water), `ValveA` (water), `PumpB` (siroop), `ValveB` (siroop) en de `PresenceSensor` (beker plaatsing sensor).

LED condities:

- Als `pompA` aan staat en `ValveA` uit staat dan gaat `GreenLedA` branden
- Als `pompB` aan staat en `ValveB` uit staat dan gaat `GreenLedB` branden

Als deze voorwaarden niet worden voldaan dan gaan de respectieve led rood branden.

Als beide pompen succesvol aan staan en de cup aanwezig is dan zal de onderste led (GreenLedM) branden, anders de gele led (*YellowLedM*). De testen die hier voorgeschreven zijn testen of de juiste LED's gaan branden wanneer de status van de actuatoren en effectoren worden beïnvloed.

Hier is een voorbeeld van een van deze testen:

```
def test_update_leds_pumpA_on_valveA_off_pumpB_on_valveB_off_cup_true(self):
    self.ctl.pumpA.switchOn()
    self.ctl.valveA.switchOff()
    self.ctl.pumpB.switchOn()
    self.ctl.valveB.switchOff()
    self.ctl.cup.set(True)
    self.ctl.updateLeds()

    self.assertEqual(self.ctl.ledYellowM.isOn(), False)
    self.assertEqual(self.ctl.ledGreenM.isOn(), True)
```

In deze tests worden de pompen aangezet en kleppen dichtgezet en er wordt een (virtuele) beter geplaatst. Dit zijn de eisen voor het groene led om te branden, dus deze zal ook moeten branden.

Display

In deze groep wordt het LCD display getest. Het display geeft tekst weer tijdens het invoeren en bij fouten. Het is belangrijk dat de gebruiker zich bewust is van fouten. Vandaar is het van belang om te testen of het display de (juiste) fouten correct laat zien.

Hier wordt getest of het display de juiste waarde laat zien in de `IDLE` state van het systeem.

```
def test_display_idle_state(self):
```

Hier wordt getest of de het display de juiste waarde laat zien in de `WAITING_FOR_CUP` state van het systeem.

```
def test_display_waiting_for_cup_state(self):
```

Hier wordt getest of de het display de juiste waarde laat zien als de cup vroegtijdig wordt verwijderd.

```
def test_display_dispensing_cup_removed_fault(self):
```

Hier wordt getest of de het display de juiste waarde laat zien wanneer het water vat leeg is. Als een gebruiker een verkeerde hoeveelheid aan water invoert dan zal deze fout weergegeven moeten worden.

```
def test_display_dispensing_water_shortage_fault(self):
```

Hier wordt getest of de het display de juiste waarde laat zien wanneer het siroop vat leeg is.

```
def test_display_dispensing_syrup_shortage_fault(self):
```

Hier wordt getest of de het display de juiste waarde laat zien wanneer er een te hoge temperatuur wordt geselecteerd.

```
def test_display_selection_temp_too_high_fault(self):
```

Hier wordt getest of de het display de juiste waarde laat zien wanneer er een niet gespecificeerde invoerfout optreed

```
def test_display_selection_invalid_fault(self):
```

Het LCD geeft bijvoorbeeld fouten in het systeem weer, zoals “te weinig siroop”. Hierdoor weet de gebruiker dat er iets mis gaat. Dit is een belangrijk deel van de Lemonator. Daarom zijn de testen die valideren dat het LCD goed functioneert ook belangrijk voor het systeem.

Cup validation

in deze groep wordt de cup validatie functie getest, er wordt in het systeem vaak gekeken of de cup niet vroegtijdig wordt verwijderd. Het is dus van belang dat dit goed gaat, omdat het systeem anders door kan gaan met dispencen tewijl de cup er niet meer staat of stop met dispencen wanneer de cup er wel staat.

Hier wordt getest of de cup wordt gevalideerd wanneer die er wel staat.

```
def test_validate_cup_appearance_cup_true(self):
```

Hier wordt getest of de cup wordt gevalideerd wanneer die er niet staat.

```
def test_validate_cup_appearance_cup_false(self):
```

Zonder het valideren van de cup zou het systeem kunnen door pompen zonder beker. Dit is niet de bedoeling, dus deze test zorgen er voor wat we zeker weten dat we de cup status goed uitlezen.

Verantwoording afwijking van testplan

Een opsomming van afwijkingen ten opzichte van het testplan)

Wijzingen ten opzichte van het eerder opgestelde testplan:

1. Er is niet gebruik gemaakt van *integratietesten*. Integratietesten worden uitgevoerd om de functionering van interfaces van systeemcomponenten te valideren. Er is besloten om uiteindelijk om geen integratietest uit te voeren.
2. De echte hardware bleek gebreken te hebben. De kleursensor werkt niet (correct). Hierdoor hebben wij er voor gekozen om deze sensor niet te gebruiken.
3. Ook bleek de ultrasone sensor erg inaccuraat. Het toepassen van een *mean* of *average* filter is geprobeerd, maar biedt helaas geen oplossing; de meetwaarden hebben onderling een te groot verschil. Hierdoor is er voor gekozen om deze sensor niet te gebruiken.
4. De sensor die de aanwezigheid van de beker hoort te detecteren werkt niet. Hierdoor hebben wij er voor gekozen om deze sensor niet te gebruiken. De implementatie van *PresenceSensor* geeft nu altijd waar terug. De beker is in dit geval altijd aanwezig.