Ohjelmointi Y2
Älykäs reseptikirja
Dokumentaatio

Ilkka Saarnilehto
351791
Informaatioteknologia, 3.vk

1. Yleiskuvaus

Toteutin projektityönä älykkään reseptikirjan. Ohjelma ylläpitää reseptikirjaa, josta voidaan kätevästi etsiä päivän ateriaksi ruokalaji joka täyttää sopivat kriteerit. Pohjana toimii käyttäjän jääkaappi, ”varasto” jossa on listattuna mitä kaikkia ruoka-aineita käyttäjällä on käytettävissään ja kuinka paljon.

Ohjelman avulla voidaan etsiä vain ne ruokalajit, jotka voidaan valmistaa käymättä kaupassa tai ruokalajit, jotka vaativat N puuttuvaa ainetta. Lisäksi voidaan hakea reseptejä jotka sisältävät tiettyjä aineita tai eivät sisällä tiettyjä aineita. Ohjelma tukee myös allergeenien tallentamista ja ruokien hakemista niiden perusteella. Näin allergiatapauksissa voidaan hakea ruokia jotka eivät sisällä määrättyjä aineita.

Jotkut raaka-aineet voidaan myös rakentaa itse reseptistä kuin ruokalajit. Esimerkiksi mehukeitto sisältää mehua, ja mehu valmistamiselle on olemassa oma resepti. Jos jääkaapissa ei ole mehua valmiina (tai sitä ei ole tarpeeksi), osaa ohjelma itse “tehdä” mehua lisää varaston raaka-aineista.

Ohjelman hallitsee myös muutokset erilaisten mittojen välillä. Jotkin aineet, kuten munat mitataan kappaleittain. Kaapissa oleva jauho ostetaan kiloittain, mutta mitataan resepteissä desilitroissa. Ohjelmassa on luokka, joka osaa hoitaa muutokset eri mittayksikköjen välillä. Näiden muutosten pohjana toimii kaikille aineille yhteinen tekijä, tiheys.

Oma toteutukseni ei sisällä graafista käyttöliittymää eikä reseptien interaktiivista lisäämistä ja muokkaamista käyttöliittymän kautta. Sen toteutus on siis vaikeusasteeltaan helppo.

2.Käyttöohje

Ohjelmassa ei ole graafista käyttöliittymää, eli sen käyttäminen suoritetaan konsolin kautta. Ohjelma käynnistetään ajamalla main_meny.py. Tämä vaatii toimiakseen pythonin version 3.x. Tämä tuo esiin päävalikon, joka tarjoaa listan eri vaihtoehdoista. Ohjelma siis keskustelee käyttäjän kanssa yksinkertaisten kysymysten ja niitä seuraavien käyttäjän antamien käskyjen avulla.

Seuraavana vaihe vaiheelta etenevä esittely ohjelmasta ja sen valikoista.

Päävalikko

```
1. Hae resepteja
2. Tarkastele tunnettuja resepeja
3. Tarkastele tunnettuja raaka-aineita
4. Tarkastele tamanhetkista varastoa
0. Sulje ohjelma
Miten haluat edetä?
```

“Miten haluat edetä?” on kysymys, joka esitetään jokaisella valikkosivulla jossa vaihtoehtoja on useita (eli päävalikossa ja sen alavalikoissa). Etenemiskäsky annetaan kokonaislukuna, valikon vasemman laidan määräämän luvun perusteella.

Hae reseptejä

```
1. Etsi kaikki reseptit, jotka valmistettavissa varastossa olevista raaka-aineista
2. Etsi kaikki reseptit, jotka sisältävät tiettyjä raaka-aineita
3. Etsi reseptit, jotka eivät sisällä tiettyjä aineita
4. Etsi reseptit, jotka eivät sisällä tiettyä allergeenia
5. Etsi reseptit, joista puuttuu korkeintaa N-määrä raaka-aineita varastosta
0. Takaisin
Miten haluat edetä?
```

Valikko sisältää vaihtoehdot reseptien hakemiselle. Etenemiskäsky annetaan kokonaislukuna, valikon alun määräämän luvun perusteella. Mikäli valinta vaatii ylimääräistä syötettä, kysytään se käyttäjältä. Alla esimerkki allergeenit karsivan haun lisäkysymyksestä ja tuloksesta:

```
Miten haluat edetä?  4
Mita allergeeneja et halua reseptin sisaltavan? Anna allergeenit pilkulla eroteltuna
mansikka, kala
PAISTETTU RIISI
Reseptin raaka-aineet:
riisi 500 g
vesi 1 l
Reseptin ohjeet:
Keita riisi
Paista riisi
Valmista tuotetta syntyy  4 portion
```

Esimerkkitiedostosta löytyi siis vain yksi resepti annetuilla ehdoilla, ja se tulostettiin konsoliin.

Tarkastele tunnettuja reseptejä

```
1. Etsi tietty resepti ja tulosta sen tiedot
2. Listaa kaikki reseptit ja niiden tiedot
0. Takaisin
Miten haluat edetä?
```

Eli voidaan hakea kaikki reseptit tai tietyn reseptin tiedot. Etenemiskäsky annetaan kokonaislukuna, valikon alun määräämän luvun perusteella. Alla esimerkki tietyn reseptin tietojen hakemisesta:

```
Miten haluat edetä?  1
Minka reseptin tiedot haluat nakyviin?
mehukeitto
MEHUKEITTO
Reseptin raaka-aineet:
mehu 5 dl
mansikka 300 g
Reseptin ohjeet:
Sekoita mehu ja marjat
Valmista tuotetta syntyy  1 l
```

Tarkastele tunnettuja raaka-aineita

Sisällöltään/toiminnaltaan käytännössä identtinen tunnetujen reseptien tarkastelun kanssa, haut vain suoritetaan raaka-aineille.

Tarkastele tämänhetkistä varastoa

Sisällöltään/toiminnaltaan käytännössä identtinen tunnetujen reseptien tarkastelun kanssa, haut vain suoritetaan varaston sisällölle.

3. Ohjelman rakenne

Ohjelma on pääosin jaettu selkeisiin osakokonaisuuksiin, ja luokkajako seuraile näitä osakokonaisuuksia. Tiivistettynä luokkia voisi kuvailla seuraavasti:

Ingredient, **Ingredient_holder** ja **Recipes :** Olioluokkia, joihin tallennetaan reseptit, raaka-aineet ja varaston sisältö listaksi olioita.
Find_recipes: Hoitaa reseptien haun. Sisältää oman metodin jokaiselle erityyppiselle haulle (esimerkiksi kaikki varaston pohjalta valmistettavissa olevat reseptit tai kaikki reseptit, joissa ei ole tiettyä allergeenia). Käyttää hyväkseen yllä määritettyihin olioihin tallennettuja tietoja.
Unit_transfer: Hoitaa raaka-aineiden yksikköjen muutokset. Tarvitaan esimerkiksi, kun ainesosa on eri yksikössä varastossa ja reseptissä.
IO: Input output luokka. Hoitaa tiedostojen lukemisen tiedostosta, ja saadun tiedon sijoittamisen oikeaan kohteeseen (yllä kuvattuihin olioihin)
Main: Pyörittää käyttäjälle näkyvää valikkoa, ja kutsuu kulloinkin tarvittavaa hakualgoritmia käyttäjän niin määrätessä.

Käytetty luokkajako perustuu melko suoraan ajatukseen siitä, että algoritmit (Unit_transfer ja Find_recipes), tietorakenteet (olioluokat Ingredient, Ingredient_holder ja Recipes) ja käyttöliittymä (Main) pitää saada eroteltua. Lisäksi IO rakentui loogisesti omaksi luokakseen. Tämän jakoperusteen pohjalta saatiin yllä olevan tyyppinen, selkeä ratkaisu.

4. Algoritmit

-Ohjelma on perusluonteeltaan melko virtaviivainen, eikä sisällä raskaita algoritmeja. Algoritmeja esiintyy käytännössä seuraavissa ohjelman osissa: yksikön muuntajassa (unit_transfer) ja erilaisissa hauissa, joilla haluttuja reseptejä etsitään (find_recipes).

Yksikkömuunnin

Yksikkömuuntimen toteutus jakautuu pääpiirteissään neljään osaan: Mass_to_mass, volume_to_volume, mass_to_volume ja volume_to_mass metodeihin. Lisäksi on määritelty “ylimetodi” unit_transfer, johon luokan ulkopuoliset kutsut käytännössä ohjataan. Tämä metodi tarkistaa, mitä yksikköä ollaan muuntamassa ja mihin muotoon, ja kutsuu sen pohjalta oikeaa yllämainituista funktioista.

Yksinkertaisemmat tapaukset toimivat metodien mass_to_mass ja volume_to_volume kautta. Tällöin ei tarvita enempiä kaavoja, vaan yksikön muuntaminen on käytännössä kertoimen muuttamista; esimerkiksi G(gramma)->KG(kilogramma) tai TBLSP(ruokalusikka) -> L(litra). Näiden kertoimien suhteet on määritelty moduulin alussa sanakirjan muodossa.

Vaikeammat tapaukset sisältävät muuntamista massan ja tilavuuden välillä, ja tapahtuvat metodien mass_to_volume ja volume_to_mass kautta. Nämä metodit hyödyntävät aineen tiheyttä ja sen peruskaavaa: p = m/V, jossa p on tiheys, m on massa ja V on tilavuus. Tästä kaavasta saadaan pyöritettyä halutut riippuvuudet massa ja tilavuuden välillä: m = V *p ja V = m/p. Tämä toteutus vaatii sen, että kaikille raaka-aineille on tallennettuna tieto niiden tiheydestä.

Haku

Hakualgoritmit on muodostettu pohjimmiltaan todella yksinkertaisella toteutuksella, jossa verrataan milloin itse reseptien, milloin niiden raaka-aineiden nimiä stringeillä. Tehokkaampiakin toteutustapoja on olemassa (esimerkiksi binäärihaku), mutta koska käsiteltävä materiaali on todella kevyttä tekstipohjaista tietoa eikä sitä ole paljoa (ainakaan oletettavasti), ei tehokkaamman hakualgoritmin kehittäminen ole relevanttia.

Hivenen vaativamman poikkeuksen muodostaa valmistettavissa olevien reseptien hakemisen pohjana toimiva metodi, `def check_for_ingredients(self, recipe, loop_counter = 0).` Se tarkistaa, kuinka monta raaka-ainetta reseptin vaatimista on saatavilla. Se käyttää apunaan metodia check_ingredients_amount, joka tarkistaa kunkin raaka-aineen määrän riittävyyden.

Koska myös raaka-aineet voivat sisältää reseptejä (raaka-aineita voidaan siis valmistaa muista varaston raaka-aineista), toimii funktio rekursiivisesti. Jos raaka-ainetta ei ole tarpeeksi ja raaka-aineella on resepti, funktio kutsuu itseään tällä reseptillä. Huomioitavaa, että funktio palauttaa kaksi arvoa; löydetyt raaka-aineet ja loop-counterin, jota tarvitaan rekursion laskemiseen. Loop counter siis kertoo, kuinka montaa raaka-ainetta on pyritty valmistamaan varastosta.
En koe että tätä toiminnallisuutta olisi pystynyt muulla tavalla toteuttamaan ilman turhaa toistoa.

Loput hakualgoritmeista pohjaavat, kuten yllä mainittu, huomattavasti yksinkertaisempaan stringien vertailuun. Alla esiteltynä pääpiirteittään yksi tämän toimintaperiaatteen metodeista. Koodi esitetty hyvän tavan vastaisesti kokonaisuudessaan, koska metakoodiin siiryminen voisi näin pitkän funktion tapauksessa haitata enemmän kuin selventää. Kukin rivi avattu tekstiksi luettavuuden parantamiseksi.

```
  def find_no_allergens(self, allergens):
    #Etsii reseptit, jotka eivät sisällä määrättyjä allergeeneja
    self.makeable_recipes = []      #Alustetaan lista, johon ehdot täyttävät raaka-aineet lisätään.

    for recipe in self.recipes_list:  #Silmukka, jossa kaydaan lapi kaikki tunnetut reseptit

        #Määritetään resepti alustavasti sallituksi
        recipe_allowed = True

        #Silmukka, jossa käydään läpi kaikki reseptin sisältämät raaka-aineet
        for recipe_ingredient in recipe.get_ingredients():

            #Silmukka jossa käydään läpi kaikki äskeisen silmukan löytämien raaka-aineiden allergeenit
            for allergen_recipe in recipe_ingredient.get_ingredients().get_allergens():

                #Silmukka jossa käydään läpi kaikki allergeenit
                for allergen_forbidden in allergens:

                    #Tarkistetaan onko reseptin sisältämä allergeeni kiellettyjen allergeenien joukossa
                    if allergen_recipe == allergen_forbidden:

                        #Jos allergeeni kielletty, hylätään resepti ja rikotaan loop
                        recipe_allowed =  False
                        Break

#Resepti tarkistettu kiellettyjen allergeenien varalta: lisätään valmistettavissa olevien reseptien listaan
        if recipe_allowed == True:
        self.makeable_recipes.append(recipe)

    #Palautetaan valmistettavissa olevat reseptit
    return self.makeable_recipes
```

Kuten näkyy, sisäkkäisiä silmukoita on paljon. Mutta kuten mainittu, käsiteltävät tiedot ovat oletettavasti kevyitä, eikä niitä ole suunnattoman paljon -> ongelmaa ei ole.

5. Tietorakenteet

Ohjelman tietorakenteet ovat erittäin vahvasti oliopohjaisia. Esimerkiksi kaikista resepteistä ja kaikista raaka-aineista luodaan oma olionsa.

Lisäksi ohjelmaan on toisteisuuden välttämiseksi luotu “ylimääräinen” luokka ingredient_holder. Tämä luokka pitää sisällään kulloinkin ajankohtaisen ruoka-aineen määrän ja yksikön. Luokka sisältää myös viittauksen itse raaka-aineeseen, eli sen kautta pystyy vaivattomasti hakemaan myös muut raaka-aineen kannalta relevantit tiedot, jotka eivät vaihtele sen mukaan missä yhteydessä raaka-aineesta puhutaan (esimerkiksi tiheys ja allergeenit).

Näitä olioita säilytetään ja liikutellaan luokkien välillä pääosin listoissa. Lista on muuttuvatilainen rakenne. Tässä toteutuksessa muuttuvatilaisuuden hyvät puolet eivät vielä kovin hyvin tule esiin. Mutta jos projektia oltaisiin jatkettu kohti keskivaikean tason toteutusta ja mukaan olisi tullut interaktiivinen reseptien muokkaaminen ja rakentaminen. Tällöin myös reseptien poistaminen olisi noussut ajankohtaiseksi. Tällöin muuttuvarakenteinen tietomuoto olisi noussut käytännössä välttämättömyydeksi, kun reseptejä oltaisiin haluttu poistaa myös keskeltä listaa.

Erikoisimmalla tavalla on tallennettu yksiköt ja niihin liittyvät tiedot. Ensinnäkin yksiköt on merkattu globaaleiksi muuttujiksi. Tämä idean taustana oli parempi hahmottaminen ja tiedon välittäminen. Tämän jälkeen yksiköt on vielä tallennettu vielä listaan, joka kertoo onko kyseessä massan vai tilavuuden muuttuja, ja sanakirjaan, joka kertoo kyseisen mitan suhdeluvun verrattuna muihin kategorian mittoihin (mass /tilavuus mitat).

6. Tiedostot

Ohjelman lukee testitiedostoja (.txt). Ohjelman toimintaa varten tarvitaan omat tiedostot raaka-aineille, resepteille ja varastoille.Ohjelman tapauksessa nämä tiedostot ovat Recipes.txt, Ingredients.txt ja Storage.txt. Nämä tiedostot ovat mukana palautuksessa.
Kunkin näistä tiedostoista rakenne esiteltynä alempana.

Reseptit

IO (eli input output) testaa aluksi, alkaako tiedosto tekstillä RECIPES. Jos ei ala, tiedostoa ei tunnisteta

Tiedosto voi sisältää käytännössä rajattoman määrän reseptejä. Uuden reseptin alkua tiedostossa merkitään rivillä #Recipe.

Tämän jälkeen ruvetaan lukemaan sisään reseptin attribuutteja. Reseptin tapauksessa kaikki attribuutit ovat pakollisia. Nämä attribuutit ovat:
Name: määrittää reseptin nimen
Instructions: määrittää reseptin ohjeet
Ingredient: rivi sisältää raaka aineen nimen, numeerisen määrän ja yksikön, tässä järjestyksessä
Outcome: .rivi sisältää valmistuvan tuotteen määrän ja yksikön, tässä järjestyksessä.
Mikäli yhdenkin näistä attribuuteista luku epäonnistuu, on tuloksena epäonnistunut luku, josta ilmoitetaan viestillä.

Alla muotoilua noudattava esimerkkitiedosto:

```
RECIPES
#Recipe
Name                : paistettu riisi
Outcome             : 4: portion
Instructions        : Keita riisi
Instructions        : Paista riisi
Ingredient          : riisi : 500 : g
Ingredient          : vesi : 1 : l
```

Raaka-aineet

IO testaa aluksi, alkaako tiedosto tekstillä INGREDIENTS. Jos ei ala, tiedostoa ei tunnisteta

Tiedosto voi sisältää käytännössä rajattoman määrän raaka-aineita. Uuden raaka-aineen alkua tiedostossa merkitään rivillä #Ingredient

Tämän jälkeen ruvetaan lukemaan sisään raaka-aineen attribuutteja. Nämä attribuutit ovat:
Name: määrittää raaka-aineen nimen
Density: määrittää raaka-aineen tiheyden
Allergen: määrittää raaka-aineen allergeenit, mikäli raaka aineella sellaisia on. Kukin allergeeni omalla rivillään
Recipe: määrittää raaka-aineen reseptin, mikäli raaka-aineella sellainen on. Palautuksen mukana tulevassa tietokannas esimerkiksi mehu on raaka-aine, mutta sitä voidaan valmistaa vedestä ja mehutiivisteestä. Tällöin sen reseptiksi on määritelty mehu.

Näistä attribuuteista pakollisia ovat Name ja Density. Mikäli yhdenkin näistä attribuuteista luku epäonnistuu, on tuloksena epäonnistunut luku, josta ilmoitetaan viestillä.

Alla muotoilua noudattava esimerkkitiedosto:

```
INGREDIENTS
#Ingredient
 Name        : kala
 Density     : 3
 Recipe      :
 Allergen    : kala
 Allergen    : ruodot
```

Varasto

IO testaa aluksi, alkaako tiedosto tekstillä STORAGE. Jos ei ala, tiedostoa ei tunnisteta.

Tiedosto voi sisältää käytännössä rajattoman määrän raaka-aineita. Uuden raaka-aineen alkua ei varsinaisesti merkitä, vaan yhdelle riville on aina tallennettuna yksi varaston komponentti. Tämän jälkeen ruvetaan lukemaan sisään varaston komponentteja (eli raaka-aineita). Toisin kuin raaka-aineilla ja resepteillä, varastolla ei ole omaa luokkaa. Sen attribuutit luetaan ingredient_holder olioihin (selitetty ylempänä).

Tiedostosta luettavat tiedot ovat:
Varastokomponentin nimi : Varastokomponentin määrä : Varastokomponentin yksikkö

Alla muotoilua noudattava esimerkkitiedosto:

```
STORAGE
riisi:10:kg
mehu:1:dl
vesi:10:l
mehutiiviste:5:l
```

7. Testaus

Suunnitelmassa esitetty järjestelmäntestaus

Alkuperäinen järjestelmäntestaussuunnitelmani oli melko pintapuolinen ja keskittyi käsittelemään ongelmia, jotka olivat ilmeisiä ennen ohjelman varsinaista toteutusta. Alla listattuna ranskalaisilla viivoilla silloin määrittämäni testattavat kriteerit, joiden alla lyhyt katsaus siitä miten tämän kyseessä oleva testaus toteutui lopullisessa ohjelmassa:

-Ruoka-aineita annetaan syötteenä tuloksia rajatessa. Näiden tulee olla ohjelman tunnistamia. Eli testataan typot, white-spacet ja ylipäätään virheelliset aineet.

White-spacet poistetaan, ja saadut tekstisyötteet myös karsitaan järjestelmällisesti isoista kirjaimista yhtenäisen kirjoitusasun saavuttamiseksi. Virheellisesti kirjoitetut raaka-aineet ja muut tekstisyötteet jätetään huomiotta. “Typojen” testaaminen olisi ollut ensinnäkin vaivalloista. Suurimmassa osassa hauista riittää, että syöte jätetään huomiotta eikä johda Erroriin. Mikäli esimerkiksi haetaan tiettyä reseptiä, jonka nimi on virheellisesti kirjoitettu, antaa ohjelma yksinkertaisen viestin “Haluttua reseptia ei loytynyt”.

- Muiden rajoittavien tekijöiden oikeellisuus, esim. puuttuvien tarvikkeiden määrä numerona.

Menu tarkistaa syötteenä saadut kokonaisluvut, esimerkiksi menussa annettavat etenemiskäskyt ja juuri puuttuvien tarvikkeiden määrän numerona. Mikäli syöte ei ole kokonaisluku, antaa ohjelma yksinkertaisen viestin “Annettu arvo ei ole kokonaisluku”.

- Mittojen kanssa painiminen. Joillekin aineille voi joutua määrittämään useita eri mittayksiköitä, ja mikäli yksikköä ei tunnisteta, tulee ohjelman reagoida siihen selkeästi.

Toteutettu puolittain. Tarkastusta ei tapahdu kun raaka-aineita lisätään, mutta kun niitä muunnetaan, ohjelma antaa varoituksen tunnistamattomasta tiedostotyypistä

- Ruokien ”sisäkkyys”. Esimerkiksi jauhelihataikina eivät voi sisältää lihapullia, vaan asian tulee olla toisinpäin, muuten vastassa tilanne jossa ruoka käytännössä sisältää itseään.

Reseptin raaka-aineiden saatavuutta selvittävä rekursiivinen funktio sisältää silmukoita laskevan muuttujan (loop_counter), joka estää tämäntyyppisestä asettelusta mahdollisesti seuraavat päättymättömät silmukat. Varsinaista vertailua ei ole, mutta toisaalta voidaan melko turvallisesti olettaa, että käyttäjän tiedostossa ei ole tämän tyyppisiä epäloogisuuksia.

- Ruokamäärä todennäköisesti helpoin määrittää henkilöiden määrän ja niihin pohjautuvien kertoimien perusteella. Tällöin täytyy pitää huolta, ettei varastoon jää mahdottomia määriä ruokaa, esimerkiksi 1/3 kananmunaa.

Päädyin lopulta lisäämään arvoille PIECE (kappale) ja PORTION (annos) omat suhdelukunsa massaan ja tilavuuteen. Näinollen ylä kuvaillun tyyppinen tilanne on mahdollinen. Koin tämän tarpeelliseksi sujuvuuden kannalta. Mikäli PECE ja PORTION eivät olisi sidonnaisia mihinkään tilavuuteen tai massaan, ei ohjelma pääsisi yli tilanteesta jossa niitä täytyisi muuntaa. Tai tätä varten pitäisi tehdä oma algoritminsa. Tästä lisää osiossa 9. Ohjelman tunnetut viat ja puutteet.

Testaus tekemisen aikana ja yksikkötestit

Ohjelman tekemisen aikana hahmottui nopeasti mitkä ovat ohjelman toiminnan kannalta tärkeimmät luokat. Nämä luokat yksikkötesteineen esiteltynä alapuolella.

IO

Suorittaa tiedon lukemisen tiedostoista ja tallentaa sen oikeisiin olioihin. Koska itse olioissa ei enää ole testausta virheellisen syötteen varalta, korostuu IO:n oikeaoppisen toimivuuden merkitys entisestään. Luokka reagoikin hyvin moneen erilaiseen virheellisen syötteeseen, ja sille on kattavat yksikkötestit.
Alkuperäiset testit luotiin generoimalla syötettä, jota ei oltu tallennettu mihinkään. Myöhemmin lisättiin testit, jotka lukevat tietonsa suoraan tiedostosta, ja mukailevat näin ollen huomattavan paljon enemmän lopullisen ohjelman toimintaa. Alla esiteltynä yksi näistä testeistä, raaka-aineiden lukemista testaava metodi.

```
    def test_read_ingredients_from_actual_file(self):
        f = open('Ingredients_test.txt', 'r')
        ingredients_list, successfull_reads, failed_reads =
self.IO.read_ingredients_from_file(f)
        f.close()
        self.assertEqual(2, successfull_reads,"Onnistuneiden lukujen maara ei tasmaa")
        self.assertEqual(1, failed_reads, "Epaonnistuneiden lukujen maara ei tasmaa")
        self.assertEqual(successfull_reads, len(ingredients_list), "Listan pituus ei vastaa
onnistuneita lukuja")
        if len(ingredients_list) > 0:
            ingredient = ingredients_list[0]
            self.assertEqual("Kala", ingredient.get_name(), "Raaka-aineen nimi ei tasmaa")
            self.assertEqual(3, ingredient.get_density(), "Raaka-aineen tiheys ei tasmaa")
            self.assertEqual(["Kala","Ruodot"], ingredient.get_allergens(), "")
```

Unit_transfer

Suorittaa raaka-aineiden yksikkömuunnokset. Kukin luokan sisältämistä, unit_transfer metodin alaisista metodeista on testattu yksikkötestin avulla. Alla esimerkki vaikeampaa kategoriaa (muutos massan ja tilavuuden välillä) edustavasta testifunktiosta

```
    def test_transfer_volume_to_mass(self):
        unit = L
        wanted_unit = G
        amount = 0.01
        density = 4

        received_amount = self.Unit_transfer.unit_transfer(unit, wanted_unit, amount, density)
        self.assertEqual(40, received_amount, "Saatu maara ei tasmaa")
```

Find_recipes

Suorittaa ohjelman kannalta oleellisimman osion eli haluttujen reseptien hakemisen. Sisältää myös toiminnaltaan monimutkaisimmat metodit. Jokaiselle metodille tehty oma yksikkötesti (lukuunottamatta apufunktiota check_ingredient_amount).

Alla esitettynä kaksi esimerkkitestiä.

Testi `def test_check_for_ingredients_more_ingredients_made_from_storage` testaa vaikeimmain tapauksen, joka liittyy valmistettavissa oleviin resepteihin, eli tilanteen jossa jotain raaka-ainetta pitää tehdä lisää varastosta. Esimerkkitapauksessa resepti, jonka raaka-aineita haetaan, on mehukeitto. Mehua ei kuitenkaan ole varastossa tarpeeksi, niinpä sitä valmistetaan lisää varastossa olevista muista raaka-aineista (mehutiiviste, vesi).
Loop_count kertoo, kuinka monta kertaa varastossa käytiin etsimässä reseptin sisältävän raaka-aineen komponentteja.

```
    def test_check_for_ingredients_more_ingredients_made_from_storage(self):
        self.Find_recipes = Find_recipes()
        recipe = self.recipes_list[2]  #mehukeitto
        ingredients_found, loop_count = self.Find_recipes.check_for_ingredients(recipe)
        self.assertEqual(2, ingredients_found,"Loytyneiden raaka-aineiden maara ei tasmaa")
        self.assertEqual(1, loop_count, "Raaka-aineiden, joita yritettiin valmistaa
varastosta, maara ei tasmaa")
```

Testi `def test_find_no_allergens` testaa valmistettavissa olevien reseptien määrän oikeellisuuden, kun tietty allergeeni (tässä tapauksessa tärkkelys) on kielletty. Myös muut tietyn ehdon valossa reseptejä etsivät metodit (esim. find_must_include, joka etsii kaikkia reseptit jotka sisältävät tiettyä raaka-ainetta) on testattu erittäin samantyyppisellä testillä kuin alla oleva.

```
    def test_find_no_allergens(self):
        self.Find_recipes = Find_recipes()

        allergens = ["tarkkelys"]
        makeable_recipes = self.Find_recipes.find_no_allergens(allergens)
        self.assertEqual(2, len(makeable_recipes), "Valmistettavissa olevien reseptien maara
ei tasmaa kun allergeeni tarkkelys on kielletty")
```

8. Ohjelman tunnetut viat ja puutteet

Ohjelmassa on muutama selkeä puute, jotka eivät kuitenkaan merkittävästi haittaa ohjelman toimintaa.

Ääkkösten puuttuminen luettavista tiedostoista. Päädyin tähän käytettyäni huomattavan paljon aikaa vian selvittämiseen keksimättä loogista syytä siihen, miksi ääkkösten luku ei onnistu. Tästä johtuen, esimerkiksi kun haetaan reseptejä allergeenilla tärkkelys, ei karsiminen toimi: allergeeni pitää esittää muodossa tarkkelys.
Ongelman selvittäminen vaatisi parempaa tutustumista ääkkösten käyttäytymiseen eri tiedostorakenteissa. Ongelman korjaamiseksi on melko varmasti olemassa yksinkertainen keino, en sitä vain itse keksinyt ja kiire vei voiton.

Yksikkömuuntimen (unit_transfer) tämänhetkisessä toteutuksessa mittayksiköt PORTION ja PIECE on sidottu massa- ja tilavuusyksiköihin (PORTION massaan ja PIECE tilavuuteen). Tämä voi tietyissä tilanteissa aiheuttaa omituisia ongelmia, kun esimerkiksi varastoon jää puolikas kananmuna, kun kananmunia on vähennetty painon mukaan. Toteutus kuitenkin mahdollistaa tämäntyyppisten laskujen suorittamisen; mikäli PORTION ja PIECE olisi määritetty täysin omiksi, riippumattomiksi yksiköikseen, voisi äskeisen kuvaillun tyyppinen yritys lamauttaa koko ohjelman.
Toinen mahdollinen toteutustapa olisi ollut estää äsken kuvaillun tyyppinen yritys jonkinlaisella hyvin muotoillulla virheilmoituksella, jonka jälkeen käyttäjän olisi itse annettu muodostaa päässään jonkinlainen riippuvuus kulloinkin kyseessä olevan “kappaleen” ja halutun massan/tilavuuden välille. Tämä olisi ollut mahdollisesti tyylikkäämpi toteutus. Sujuvuuden vuoksi päädyin kuitenkin kuvailemaani malliin.

Puutteeksi voi laskea myös tämänhetkisen hakujen toimintaperiaatteen. Jos haetaan reseptejä, jotka voidaan valmistaa varastossa olevista raaka-aineista, ohjelma toimii oikein ja tulostaa. Vastaavasti, jos halutaan esimerkiksi tulostaa reseptit, jotka eivät sisällä tiettyä allergeenia, ohjelma osaa tulostaa ne oikeaoppisesti. Ohjelma ei kuitenkaan yhdistä näitä kahta hakutulosta; allergeenihaulla saatu tulos siis sisältää kaikki olemassa olevat reseptit, jotka eivät sisällä kyseistä allergeenia.
Tämän korjaaminen olisi ollut todella yksinkertaista, käytännössä kahden olemassa olevan hakutuloksen yhdistämistä (tai tässä tapauksessa leikkauksen hakemista). Tämäntyyppistä toteutusta ei kuitenkaan tehtävänannossa pyydetty, enkä ollut itsekään kumpi toteutustapa olisi loppupeleissä se paras mahdollinen. Paras mahdollinen ratkaisu olisi lisätä vaihtoehto yhdistää eri hakuehtoja; tämä olisi kuitenkin vaatinut jo suuria muutoksia valmiiksi luotuun valikkoon, enkä ajanpuutteen takia tätä toteutusta kehittänyt.

9. Parhaat ja heikoimmat kohdat

Parhaat kohdat

Ohjelman yksikkötestaus on todella kattava, kaiken kaikkiaan 17 testiä. Nämä testit käyvät tehokkaasti läpi kaikki ohjelman toiminnan kannalta tärkeimmät metodit, keskittyen keskiössä olevien luokkien Find_recipes, Unit_transfer ja IO toimintaan.

Reseptien ainesosien saatavilla olevuutta testaava metodi `def check_for_ingredients` oli melko ylivoimaisesti ohjelman vaikein osa toteuttaa. Funktio käyttää hyväkseen melko lailla kaikkia ohjelmassa määritetyjä muita funktoita, ja on näinollen todella tärkeä yhtymäkohta ohjelmassa. Lopputulos johon pääsin on kuitenkin erittäin toimiva.
Funktio saa syötteenään reseptin, ja tarkistaa sitten kuinka monin sen vaatimista ainesosista on saatavilla varastossa. Koska joitain raaka-aineita voidaan myös valmistaa lisää varastosta, toimii funktio rekursiivisesti; mikäli jotain ainesosaa on saatavilla riittämättömästi, mutta kyseiselle ainesosalle on asetettu resepti, kutsuu funktio itseään tällä uudella reseptillä. Funktio palauttaa tiedon siitä, kuinka monta raaka-ainetta löydettiin varastosta, ja lisäksi tiedon siitä, kuinka montaa raaka-ainetta pyrittiin valmistamaan lisää varastossa olevista muista raaka-aineista.
Kaikki vaihtoehtoiset toteutukset olisivat todennäköisesti sisältäneet huomattavia määriä turhaa toistoa.

Heikoimmat kohdat

IO (input output) sisältää itsessään paljon virheiden varalta testaamista. Mikäli virhe kuitenkin pääsee livahtamaan näiden tarkistusten läpi ja päätyy olioon, se myös pysyy siellä; olioita määrittävät luokat eivät itsessään sisällä minkäänlaista virheiden tarkkailua.
Tämän olisi voinut korjata yksinkertaisesti lisäämällä virheiden tarkkailun. Aikani ei kuitenkaan tähän riittänyt.

Yksikkömuunnin on tietorakenteeltaan melko sekava. Sen sisällä yksiköt on määritelty kolmeen eri tiedostorakenteeseen. Globaaleiksi muuttujiksi (alkuperäinen idea, tekee koodista selkeämpää kun yksiköihin voidaan viitata suoraan niiden omalla nimellä esim. KG), listoihin (mass_units ja volume_units, ideana halutun muutoksen helppo eteenpäin siirtäminen määritellyille metodeille), sekä sanakirjaan (sisältää suhteet jolla yksiköt suhteutuvat toisiinsa). Tämän kaiken olisi voinut todennäköisesti toteuttaa myös yhdellä elegantilla sanakirja-tyyppisellä ratkaisulla, mutta aikani ei enää riittänyt olemassa olevan toteutuksen hiomiseen. Myös apufunktion `def unit_string_to_variable(self, unit):` olisi pystynyt toteuttamaan tyylikkäämmin sanakirja-tyyppisellä ratkaisulla

10. Poikkeamat suunnitelmasta

Eniten alkuperäisestä suunnitelmastani poikkesi työn toteuttamisjärjestys. Olin alkuperäisessä suunnitelmassa kuvaillut näin:

Ohjelman toteuttaminen kannattaa mielestäni aloittaa juuri perusrakenteen toteuttamisesta. Tämän ydinosan muodostavat luokat etsi_ruoka, lista, resepti ja raakaaine. Nämä ovat toistensa toiminnalla välttämättömiä luokkia, jotka muodostavat käytännössä koko toiminnallisuuden

Lähdin kuitenkin pitkän tyhjän näytön tuijottamisen jälkee liikkeelle siitä, että haluan saada aikaan edes jotain mikä toimii. Näinollen toteutin ensimmäisenä ohjelman käyttämät valikot. Tämän jälkeen siirryin ohjelman runkoon, eli luokkien Ingredients ja Recipes toteuttamiseen, sekä niitä täyttävän luokan IO (input output) implementointiin. Aikaisemmassa suunnitelmassa mainittu etsi_ruoka luokan toteutus oli lopulta viimeinen asia jonka tein; tämä on tavallaan ilmeistä, sillä eihän hakua voi tehdä ennen kuin haettavat komponentit on saatu muodostettua järkevästi. Tämä ei vain itselleni ollut vielä alustavassa suunnitteluvaiheessa täysin selvää. Aikataulutuksen kannalta oikeastaan ainoa asia joka piti kunnolla paikkansa oli mittamuuntimen toteuttaminen projektin loppupuolella.

Muutenkin suunnitelma muuttui huomattavasti. Esimerkiksi aikaisemmassa suunnitemassani mainittua luokkaa tulosta lista en toteuttanut lainkaan, vaan sisällytin eri lopputulosten tulostamisen Main luokkaan (joka siis pyörittää valikoita).

Lisäksi toteutin muutaman luokan joita ei alkuperäisessä suunnitelmassani ollut mainittu lainkaan. Näistä selvempi tapaus on IO, joka siis lukee tietoja tiedostosta ja tallentaa niitä haluttuihin paikkoihin. Tämän luokan tarpeellisuus ja olemassaolo oli kokonaan unohtunut alkuperäisestä suunnitelmastani.
Tämän lisäksi jaoin Ingredient luokan kahteen osaan, luokkiin Ingredient ja Ingredient_holder. Näistä ensimmäinen pitää sisällään raaka-aineeseen liittyvää muuttumatonta tietoa (tiheys), ja jälkimmäinen tietoa joka vaihtelee sen mukaan missä tilanteessa raaka-ainetta käsitellään. Esimerkiksi kun puhutaan raaka-aineesta reseptissä ja raaka-aineesta varastossa, ovat tiedot raaka-aineen määrästä erittäin erilaiset.

11. Toteutunut työjärjestys ja aikataulu

Kuten yllä mainittu, toteutunut työjärjestys erosi huomattavasti alunperin suunnitellusta. Alla pintapuolinen erittely siitä, missä järjestyksessä ja minä päivinä ohjelman osat valmistuivat:

20.4 : Toimiva menu

21.4 : Toimivat (lopulliseen toteutukseen asti päätyneet) pohjat luokille Ingredients, Recipes, ja Ingredients_holder

22.4 : IO:sista raakaversio kasassa

28.4 : IO:s toimintakunnossa käytännössä nykyisessä muodossaan

30.4 : Alustavasti toimiva yksikkömuunnin

5.5 - 6.5 : Haun toiminnan kannalta kriittiset funktiot toimivat (todella pitkän väännön jälkee)

12.5 : Käytännössä kaikki haut toimivat. Kulloinkin haluttavan ainesosan tulostaminen hiottu kuntoon

13.5 : Lisätty vielä hakuja, jotka eivät ole tehtävänannon kannalta välttämättömiä

Mainittujen päivämäärien lisäksi aikaa tuli projektin tekemiseen käytettyä myös muina päivinä; toisaalta kaikki mainitut päivät eivät välttämättä olleet aivan täysiä projektin kimpussa käytettyjä työpäivä. Melko rehellinen arvio projektiin käytetystä ajasta voisi olla 8-9 työpäivää.

Teknisessä suunnitelmassani olen jäsennellyt, että projektin vaikeusasteeltaan helppoon toteuttamiseen menis 5-8 työpäivää (best case scenario - worst case scenario). Näin ollen työhön loppujen lopuksi kulunut aika vastaa melko tarkasti alkuperäisen arvioini worst case scenariota.

Aikataulut kuitenkin menivät omalta osaltani pieleen projektin valmiusasteen arvioinnissa. Ohjelman hakutoimintojen ydinfunktoiden tekemiseen kului lopulta melkein kaksi päivää, vaikka olin kuvitellut niiden toteuttamisen olevan melko helppoa olemassaolevien, hyvien toteuttamis suunnitelmien perusteella.

12. Arvio lopputuloksesta

Niiltä osin kun sain toteutuksen kasaan olen siihe melko tyytyväinen. Koodin rakenne on ylipäätään toimiva; luokkajako on tehty hyvin, ja käytetyt metodit ovat pääsääntöisesti järkevästi rajattuja ja toimivia kokonaisuuksia. Tästä selkeästä ja toimivasta rakenteesta seuraa helppo laajennettavuus. Olen varma että tälle pohjalle olisi helppo rakentaa uusia toiminnallisuuksia, ja vaikkapa GUI:n implementointi ei olisi tuottanut ongelmia.

En jälkikäteen keksi montaa tapaa, joilla toteutettua luokkajakoa olisi voinut parantaa. Esimerkiksi haluttujen tulosten printtaamisen olisi voinut ulkoistaa omaksi luokakseen, mutta sisältöä tähän luokkaan ei tämänhetkisestä ohjelmasta olisi tullut paljoa.

Toisaalta olen pettynyt siihen että juuri GUI ja muut keskivaikean työasteen vaatimat ominaisuudet jäivät toteuttamatta. Varsinkin GUI:n tekeminen olisi ollut itselleni se projektin kiinnostavin osuus. Tästä on syyttäminen omaa aikataulutustani. Ensinnäkin en aloittanut työtä tarpeeksi aikaisin, toisekseen hahmotukseni jäljellä olevan työn määrästä petti pahasti. Näinollen asiat jäivät viime tippaan.

Muutenkin työhön jäi joitakin toteutuksia, jotka olisi voinut hoitaa paremmin, mutta aikataulu tuli vastaan. Näistä tarkemmin kappaleissaa 8. Tunnetut viat ja puutteet sekä 9. Parhaimmat ja heikoimmat kohdat. Opittavaa olisi siis ainakin aikataulutuksessa ja työn määrän hahmotuksessa.

14. Viitteet

https://docs.python.org/3/

http://stackoverflow.com/

15. Liitteet

Ohjelman lähdekoodi on palautettu gitin kautta.