Metropolia

Sunnittelumallit Harjoitustehtävät / Simo Silander
Sisältää suunniteltuja tehtäviä, mutta voi päivittyä kurssin aikana.

Ohjelmointitehtävät koodataan pääsääntöisesti Javalla. Netistä löytyviä vapaasti käytettäviä ratkaisuja voi esittää, kunhan ne osaa selittää.

1. **Factory Method:** Työtilan Tehtäväpohjat-kansiossa on factorymethod.zip, joka sisältää koodia, jossa AterioivaOtus aterioi. Esimerkkiohjelman Juoma luodaan AterioivanOtuksen Opettaja-aliluokassa. Juomanluontimetodi on tehdasmetodi. Kirjoita Opettaja-luokalle kaksi rinnakkaista luokkaa, joissa kummassakin luodaan sopiva juoma.

   Luo testiohjelmassasi jokaista otustyyppiä oleva olio ja laita ne aterioimaan. (Voit halutessasi laajentaa ohjelmaa ottamalla mukaan Ruoka-rajapinnan.)

2. **Abstract Factory:** Jasper Java-koodaaja on koko opiskeluaikansa pukeutunut Adidas-merkkisiin vaatteisiin lippis ja kengät mukaan lukien. Kirjoita ohjelma, jossa Jasper pukee päällensä farmarit, t-paidan, lippiksen ja kengät. Tämän jälkeen Jasper luettelee ylpeänä, mitä hänellä on päällään (vaatekappaleet osaavat toString-metodeissaan kertoa kaiken oleellisen itsestään).

   Valmistuttuaan insinööriksi Jasper siirtyy käyttämään yksinomaan Boss-tuotteita ja osaa edelleen kertoa, mitä hänellä on päällään.

   Koodaa ohjelma siten, että Jasperiin ei tarvitse tehdä juuri muutoksia Adidas->Boss siirtymän takia, vaan ainoa muutos on tuoteperheen (tehtaan) vaihto.

   Bonus: Hyödynnä Java Reflection APIa tehtaan luonnissa.
   Tästä yksi piste lisää, jos palautus ajoissa. Myöhästyneitä palautuksia ei huomioida.

3. **Composite:** Suunnittele Composite-mallin mukaisesti rajapinnat ja toteutukset systeemille, jolla voit esittää pöytätietokoneen kokoonpanon ja laskea sen hinnan, kun kukin komponentti tietää oman hintansa.

   Idea on että tietokoneen kaikki osat ovat vaikkapa Laiteosa-rajapinnan toteuttajia. Laiteosalla on hinta.

   Laiteosia on erilaisia, esim:
   - muistipiiri
   - emolevy
   - prosessori
   - verkkokortti

- näytönohjainf
- kotelo

Näistä ainakin emolevy ja kotelo ovat koostekomponentteja. Näiden hinta muodostuu komponentin omasta hinnasta sekä mahdollisen sisällön eli liitettyjen komponenttien hinnasta.

Esitä luokkahierarkia ja rakenna pöytätietokonekokoonpano haluamistasi osista ja laske lopuksi kokoonpanon hinta ja tulosta se. Hinnan ilmoittava metodi palauttaa hinnan metodin paluuarvona.

Hyvän mielen bonus:

Esitä, kuinka voit luoda kaikki tuoteosat abstraktin tehtaan avulla. Esitä myös, kuinka konkreettista tehdasta vaihtamalla saat helposti lasketuksi eri tehtaiden tuottamien samanlaisten kokoonpanojen hintoja.

4. **Observer:** Toteuta luentomateriaalissa (Observer.pdf) Observer-mallin sovelluksena esitetty hahmotelma digitaalisesta kellosta Javalla täydentäen koodia puuttuvin osin. Käytä ratkaisussasi Javan APIsta löytyviä Observer-rajapintaa ja Observable-luokkaa. Jos haluat käyttää jotain muuta Javan tekniikkaa, koska Observer/Observable on deprecated Java 9:stä alkaen, se on ok. Vaikka Javan Observer/Observable-mekanismi on deprecated, sitä ei poisteta Javasta, mutta sen kehitystä ei jatketa. Sen käytölle on esitetty vaihtoehtoisia tapoja.

5. **Singleton:** Mieti jokin konkreettinen sovellustilanne, jossa Singleton-mallista olisi hyötyä. Esitä sovelluksen toteutus Javalla. Ks. toteutusvihjeet wikipedasta. Voit esim. toteuttaa abstraktin tehtaan singletonina.

6. **Decorator (valitse joko a, b tai c):**

   a) Toteuta data-rakenne ja sen käsittely joko pääsynvalvonnalla tai ilman pääsynvalvontaa. Pääsynvalvonta dekoraattorilla.

   b) Toteuta datan kirjoittaminen/lukeminen levylle/levyltä joko salakirjoitettuna tai ilman. Salakirjoitus ja purku dekoraattorilla.

   c) Pizzaravintola laatii pizzahinnaston decorator-mallin avulla. Pizza koostuu pohjasta ja erilaisista täytteistä. Pohjalla ja täytteillä on kullakin omat hinnat. Tee ohjelma, jossa rakennat kolme mieleistäsi pizzaa ja tulostat pizzamenun hintoineen. Pizzapohja ja kaikki täytteet toteuttavat Pizza-rajapintaa. Kaikki täytteet ovat dekoraattoreita. Piirrä myös luokkakaavio.

7. **State:** Määrittele pelihahmo ja sille kolme kehitysastetta. Toteuta eri kehitysasteet State-mallin mukaisesti. Pelihahmo voisi olla esim. Pokemon-

hahmo Charmander, jonka kaksi muuta kehitysastetta ovat Charmeleon ja Charizard. Määrittele pelihahmolle vähintään kolme erilaista toimintoa (metodia) siten, että käyttäytyminen riippuu pelihahmon tilasta. Tilanvaihtojen pitää toteutua automaattisesti ”pinnan alla”. Pääohjelma (client) ei siis vaihda Pokemonin tilaa. Pääohjelmassa Pokemon tekee esim. luupissa temppuja, joiden seurauksena tila aina jossain vaiheessa vaihtuu. Kirjoita sopivia tulostuksia, jotta tilanvaihdot havaitaan.

8. **Template Method:** Alla oleva Template Method -malliin perustuva kehys määrittelee yleisen pelin kulun. Toteuta siihen perustuen jokin yksinkertainen toimiva peli.

```
/**
 * An abstract class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 */

abstract class Game {

    protected int playersCount;

    abstract void initializeGame();

    abstract void makePlay(int player);

    abstract boolean endOfGame();

    abstract void printWinner();

    /* A template method : */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()){
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
```

Now we can extend this class in order to implement actual games:

```
class Monopoly extends Game {

    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    boolean endOfGame() {
        // ...
    }
```

```
    void printWinner() {
        // ...
    }

    /* Specific declarations for the Monopoly game. */

    // ...

}

class Chess extends Game {

    /* Implementation of necessary concrete methods */

    void initializeGame() {
        // ...
    }

    void makePlay(int player) {
        // ...
    }

    boolean endOfGame() {
        // ...
    }

    void printWinner() {
        // ...
    }

    /* Specific declarations for the chess game. */

    // ...

}
```

9. **Strategy:** tee a) tai b).

   a) Määrittele Strategy-mallin mukaisesti ListConverter-rajapinta, jossa määritellään listToString-metodi, joka saa parametrikseen List-tietorakenteen ja palauttaa listan merkkijonona, jossa

   1. strategia kirjoittaa rivinvaihtomerkin jokaisen alkion jälkeen
   2. strategia tulostaa rivinvaihtomerkin joka toisen alkion jälkeen
   3. strategia tulostaa rivinvaihtomerkin joka kolmannen alkion jälkeen

   Toteuta eri strategioissa listan läpikäynti eri tavoin.
   Eri tapoja:
   - iteraattorin käyttö
   - lista taulukoksi ennen läpikäyntiä, joka toteutetaan for-silmukassa
   - listan läpikäynti for-silmukassa käyttäen List-rajapinnan get-metodia.

   Testiohjelmassa luodaan lista ja tulostetaan eri strategioilla aikaansaatu lopputulos.

   b) Toteuta vähintään kolme erilaista taulukon lajittelualgoritmia (lähdekoodeja esim. https://java2novice.com/java-sorting-algorithms/. Esitä

sovellus, jossa lajittelualgoritmin valinta toteutetaan Strategy-mallin mukaisesti. Generoi tarpeeksi suuri lajiteltava aineisto ja mittaa lajitteluun kuluvat ajat.

10. **Chain of Responsibility:** Yrityksessä työntekijän lähiesimies pystyy hyväksymään 2 % palkankorotuspyynnöt. Pyyntö pitää ohjata yksikön päällikölle, jos se ylittää 2 % mutta on alle 5 %. Tätä suuremmat palkankorotuspyynnöt ohjataan yrityksen toimitusjohtajalle. Esitä, kuinka ratkaiset työntekijän palkankorotuspyynnön käsittelyn Chain of Responsibility –mallilla.

11. **Memento:** Arvuuttajalla on useita asiakkaita (säikeitä), joita kutakin varten se arpoo luvun, jota asiakas yrittää arvata toistuvasti. Peli käynnistyy siten, että asiakas ilmoittautuu peliin mukaan Arvuuttajan liityPeliin()-metodilla (asiakas välittää itsensä parametrina), Metodissa arvuuttaja arpoo luvun ja tallentaa sen.

    Perinteinen tapa toteuttaa Arvuuttaja, on määritellä siihen HashMap, jossa säilytetään asiakaskohtaiset tiedot (key = asiakas, value=luku). Arvatessaan lukua asiakas välittää arvaus()-kutsussa parametrina itsensä ja arvauksensa. Paluuarvona on true tai false.

    Mementoa käytettäessä HashMap-ratkaisua ei tarvita eikä asiakkaan viitteen välittämistä arvuuttajalle. Arvuuttaja palauttaa liityPeliin()-metodissa asiakkaalle Mementon, jossa on arvottu luku, joten Arvuuttajan ei sitä tarvitse muistaa. Huomaa, että asiakas ei saa päästä lukuun käsiksi! Jokaisen arvauksensa yhteydessä asiakas välittää Arvuuttajalle Mementon, jossa olevaa lukua Arvuuttaja vertaa uuteen arvaukseen ja kertoo, kuinka kävi. Peli päättyy, kun kaikki säikeet ovat arvanneet oikein. Toteuta ohjelma Javalla.

12. **Proxy:** Laajenna Wikipediassa olevan Proxy-esimerkkiä http://en.wikipedia.org/wiki/Proxy_pattern siten, että lisäät Image-rajapintaan showData()-metodin, joka näyttää kuvan nimen. Lisää pääohjelmaan valokuvakansio (esim. List-rakenne), johon lisäät valokuvia proxyinä. Tulosta pääohjelmassa tiedot valokuvakansion sisällöstä (kuvia ei ladata). Esitä myös, kuinka valokuvakansiota voidaan selata (kuvat ladataan, ellei vielä ole ladattu).

13. **Visitor:** tee a) tai b)

    a) Pohdi voitaisiinko Visitor-mallia käyttää State-mallin kontekstiolioiden käsittelyyn. Kontekstioliot olisivat eri pelihahmoja, jotka ovat eri tiloissa. Pelissä haluttaisiin käydä kaikki pelihahmot läpi esim. siten että jokaiselle hahmolle annetaan bonuspisteitä. Annettavien bonuspisteiden määrä voi riippua kuitenkin hahmosta ja sen tilasta. Bonus-visitorissa on metodi

kunkin hahmon kutakin tilaa kohden. Esitä tällainen bonuspisteiden jakaminen Java-koodina.

Toteutuksesta: Kontekstioliolle sanotaan accept(visitor), joka delegoi pyynnön tilaoliolleen.

b) Voitaisiinko päätös seuraavasta tilasta ulkoistaa Visitoriin. Tilaolio tai konteksti kutsuisi kunkin metodin lopuksi Visitoria (accept-kutsua ei tarvittaisi), joka tekee päätöksen mahdollisesta tilan vaihdosta. Esitä Java-koodina.

14. **Builder:** Kirjoita ohjelma, jossa rakennat kerroshampurilaisen **Builder**-mallin mukaisesti. Toteuta vähintään kaksi konkreettista builderia (Hesburger ja McDonalds). Toteuta builderit siten, että kummallakin on eri tietorakenne hampurilaisen koostamiseksi. Toisella esim. List, johon tallennetaan hampurilaisen osat olioina (määrittele osille omat luokat, esim. salaattia voisi edustaa konkreettisen builderin päässä class JäävuoriSalaatti, josta luodaan ilmentymä listaan) ja toisella StringBuilder, jossa osia edustavat merkkijonot. Konkreettisilla Buildereilla on oikeaa tyyppiä palauttavat getBurger()-metodit eli ne palauttavat hampurilaisen sellaisena tietorakenteena, jona se on luotu.

15. **Adapter:** Adapter-ratkaisu mahdollistaa olemassa olevan ohjelmamoduulin (esim. luokka) käyttämisen vaikka sen rajapinta olisi erilainen, kuin mitä asiakas haluaa käyttää. Mieti itse tätä havainnollistava esimerkki ja toteuta ohjelma, jossa esität, kuinka adapterin avulla olemassa oleva luokka palveluineen voidaan ottaa käyttöön muuttamatta tätä millään tavalla.

16. **Bridge:** Bridge-mallin idea on, että jokin yhdellä ohjelmistokerroksella toimiva abstraktio voidaan toteuttaa eri alustoilla ja että alusta voidaan vaikkapa vaihtaa kääntämättä asiakasohjelmaa. Valmistaudu kertomaan jokin esimerkki, jossa yhden kerroksen palvelu voidaan toteuttaa kahdella eri alustalla. Palauta kertomuksesi kirjallisessa muodossa OMAaan.

17. **Flyweight:** Lue Flyweight-mallia käsittelevä artikkeli:

    https://www.infoworld.com/article/2073632/make-your-apps-fly.html

    Testaa esimerkit 1 ja 2: String ja Border. Testaa myös OMAn Dokumentit/Tehtäväpohjat-kansiossa oleva JavaFX-esimerkki, jossa on samanlainen asetelma kuin Java Swing Border -esimerkissä. Valmistaudu esittelemään testiajot ja kertomaan, mitä testien perusteella voidaan todeta. Palauta raportti OMAan.

18. **Prototype:** Kirjoita kello+viisarit -ohjelma, jossa kello koostuu viisariolioista ja selvitä sen avulla, kuinka Javassa toteutetaan Prototype-mallia (Object.clone()). Esitä sellainen esimerkki, jossa toteutuu

syväkopiointi. Eli: tee kellosta klooni ja katso kuinka se käyttäytyy suhteessa alkuperäiseen, jos muutat alkuperäisen tai kloonin kellonaikaa.

19. **Mediator:** Suunnittele mäkihyppykisan luokat ja niiden välinen toiminta. Luokkia: Hyppääjä, Hyppy (jokaisella hyppääjällä on kaksi hyppyä, hyppääjä ylläpitää omat hyppynsä pituus- ja tyylipistetietoineen 2-alkioisessa Hyppy-taulukossaan), Mittamies, Arvostelutuomarit (5 kpl), Tulostaulu, Tulosrivi, Kisasihteeri (laskee hypyn jälkeen sille pisteet eli on käytännössä pelkkä laskukaavan tunteva laskukone, Mediator välittää sille kaikki lähtötiedot). Mediator on vastuussa edellä esitettyjen luokkien välisen kommunikointiprotokollan toteuttamisesta. Suunnittele se. Esitä ratkaisu sekvenssikaavion avulla.

20. **Iterator:** a) Tutki kuinka Javan iteraattori käyttäytyy, jos yritetään iteroida kokoelmaa kahdella säikeellä yhtä aikaa, kun molemmilla on oma iterattori.
    b) entä, jos säikeet käyttävät samaa iteraattoria vuorotellen?
    c) Kuinka käy, jos kokoelmaan tehdään muutoksia iteroinnin läpikäynnin aikana.
    d) Keksi jotain muuta testattavaa (esim. iteraattorin remove).

21. **Facade:**

    Muuta Javaksi ja täydennä Wikipediassa olevaa C++-ohjelman runkoa (https://en.wikipedia.org/wiki/Facade_pattern) tietokoneen ”boottaus-simulaattorille”. Liitä ohjelmaan välivaiheiden tulostuksia. Ohjelman suorittaminen: tulosta muistiin ladatut datat (vaikkapa merkit, char) "muistiosoite" kerrallaan. (Korvaa C++:n structit Javan luokilla ja lisää niihin tarvittavat muuttujat/tietorakenteet (harkitse taulukkorakenteita) .)
    Lue: https://en.wikipedia.org/wiki/Boot_sector

22. **Command:** Toteuta Command.pdf:n esimerkkiä mukaillen sovellus, jossa mallinnetaan kaksi seinäpainiketta: ”valkokangas ylös”, ”valkokangas alas”.