Network And System Engineering
2017 - 2018

Theorie Opgaven & Practicumhandleiding Design Patterns

Henk van den Bosch
april 2018

Inhoud

1. Planning

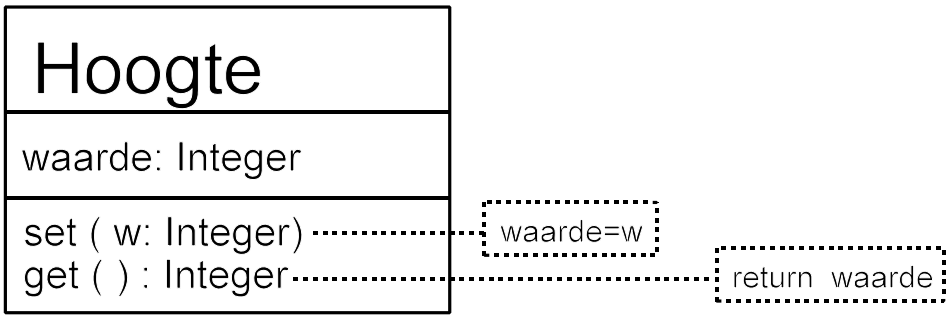
Wk	Theorie	Bron	Theorie opgaven	Practicum
1	- Layer - Model View Controller - Observer - Abstract Server - Factory Method - Template Method - Abstract Factory	Bosch: blz. 98 t/m 103 Bosch: blz. 104 t/m 113 Gamma: 5.7, Martin: H24 Gamma: 3.3 Gamma: 5.10, Martin: H14 Gamma: 3.1, Martin: H21	1. Observer / MVC 2. Lagenstructuur 3. Lagenstructuur & Abstract Server 4. Observer in Layers 5. Decorator	**1: Layers**
2	- Singleton - Facade - Composite - Decorator	Gamma: 3.5, Martin: H16 Gamma: 4.5, Martin: H15 Gamma: 4.3, Martin: H23 Gamma: 4.4	6. Template	**2: Composite & Factory Method**
3	- State - Strategy	Gamma: 5.8, Martin: H29 Gamma: 5.9, Martin: H14	7. Abstract Factory	**3: Facade**
4	- Proxy - Adapter	Gamma: 4.7, Martin: H26 Gamma: 4.1, Martin: H25	8. State	**4: Decorator**
5	- Bridge - Command	Gamma: 4.2, Martin: H25 Gamma: 5.2, Martin: H13	9. Proxy	**5: Robot**
6	- Type Object - Knowledge level	Johnson: 2.5 én 2.6 Johnson: 2.5 én 2.6	10. Adapter	
7			11. Bridge 12. Knowledge level	

Gamma: Design Patterns van Gamma, Helm, Johnson en Vlissides,
Martin: Agile Software Development van Robert Martin,
Johnson: Reader van Type Object Pattern
Bosch: Studiewijzer van SOH2SI van van den Bosch

2. Theorie opgaven

1 Observer / Model View Controller

Gegeven is de onderstaande klasse “Hoogte” die voorkomt in het ontwerp van een softwaresysteem in een vliegtuig.



De hoogte moet d.m.v. verschillende soorten meters (analoge meters, digitale meters) op het beeldscherm worden weergegeven.

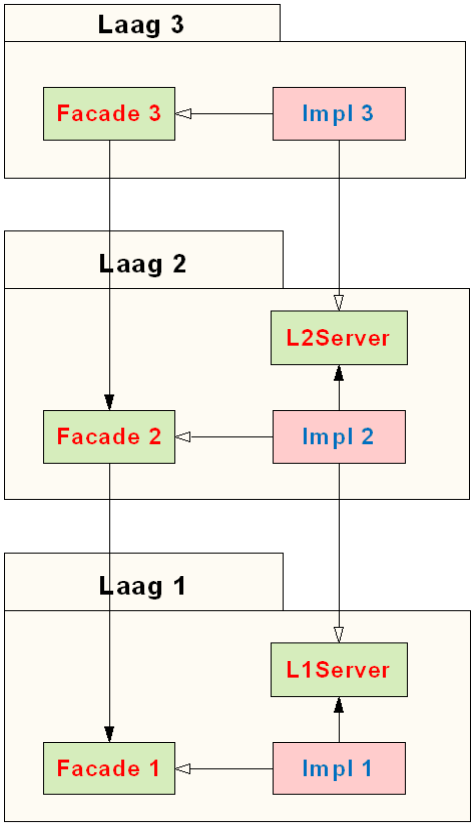
a) Maak een ontwerp op basis van de Model View Controller architectuur. Combineer hierbij de view en de controller tot één klasse zoals in een Document View architectuur. Ga hierbij van uit dat er geen framework voorhande is om deze architectuur te realiseren.

   Geef ook een sequentiediagram waarin de werking van het ontwerp duidelijk wordt. Gebruik minimaal 2 meter-objecten.

b) Dezelfde vraag als onderdeel a).
   Gebruik nu het Observer Framework op de laatste bladzijde.

   Geef ook weer een sequentiediagram waarin de werking van het ontwerp duidelijk wordt. Gebruik ook nu minimaal 2 meter-objecten.

2 Lagenstructuur



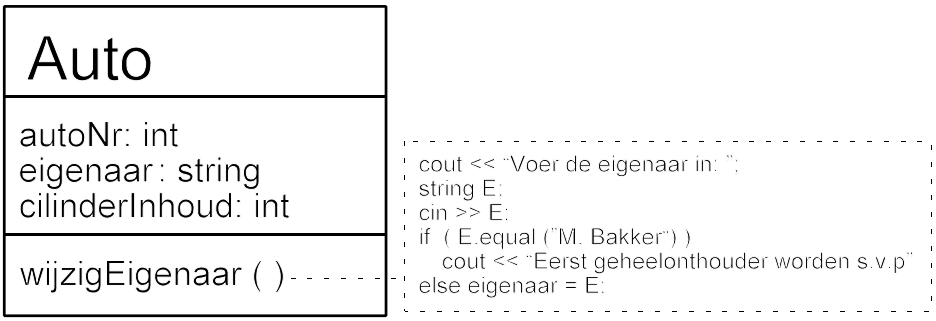
Gegeven de lagenstructuur in nevenstaande figuur.

Geef een implementie in C++ van de gehele lagenstructuur. Gebruik de onderstaande main-functie:

```
int main( )
{ Laag1Impl L1;
  Laag2Impl L2;
  Laag3Impl L3;

  L1.connectHigh(&L2);
  L2.connectLow (&L1);
  L2.connectHigh(&L3);
  L3.connectLow (&L2);

  L3.service();

  return 0;
}
```

3 Lagenstructuur & Abstract Server

Gegeven is de nevenstaande klasse.



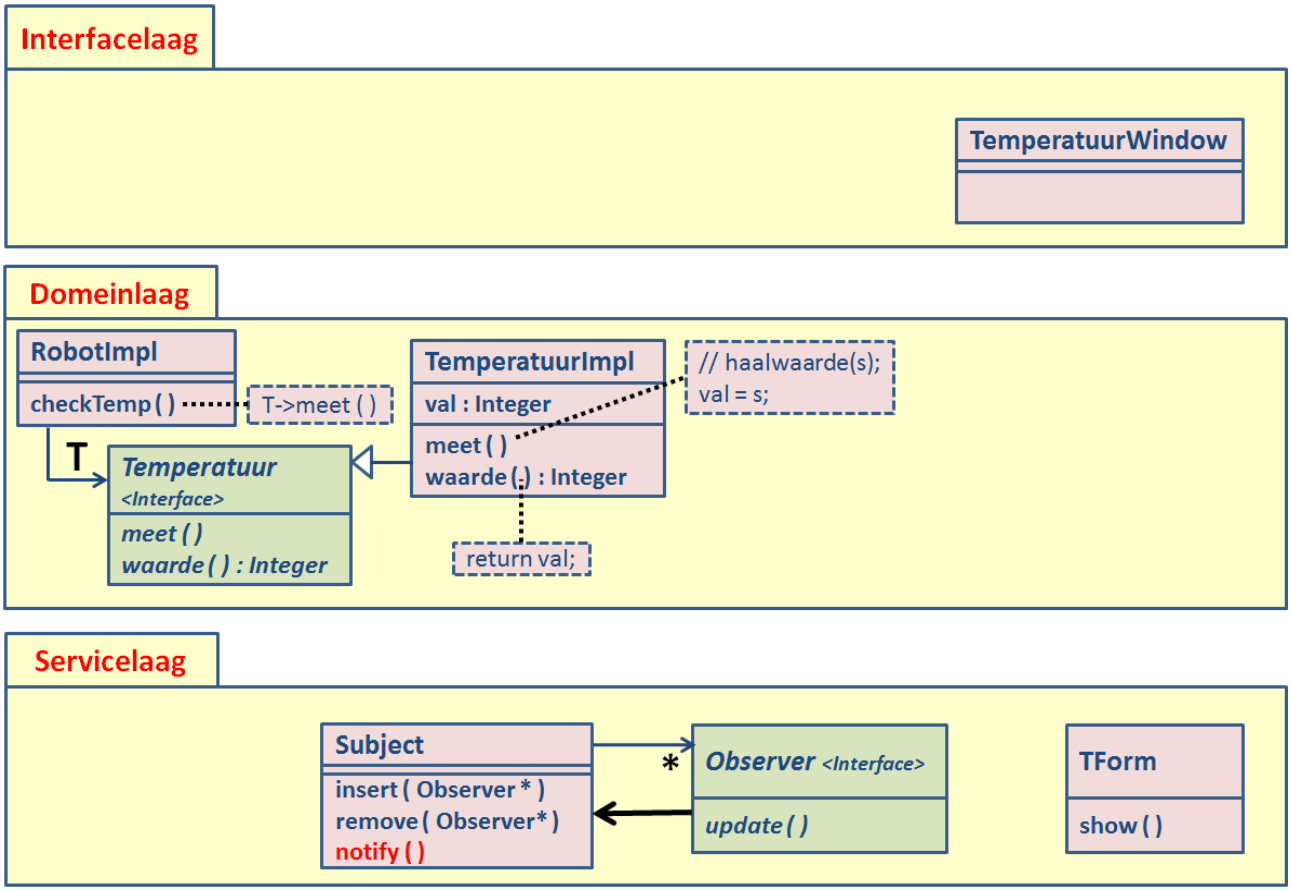
In de operatie “wijzigEigenaar( )” wordt blijkbaar gebruik gemaakt van de standaard in- en uitvoerklassen uit <iostream.h>.
De klasse “Auto” moet, door het toepassen van een ***Abstract Server***, onafhankelijk worden gemaakt voor de manier van invoeren (met cin) en uitvoeren (met cout) van de gegevens. Daartoe moet onder meer de klasse “InvoerScherm” worden gedefinieerd in de interfacelaag. De klasse “Auto” is en blijft deel van de semantische laag. De besturingslaag kan in deze opgave buiten beschouwing worden laten.

a) Geef het (volledige) klassediagram, dus inclusief attributen, operaties en algoritmen.

b) Teken een sequentiediagram behorend bij het diagram van vraag a waarin van **links naar rechts** de volgende 2 objecten voorkomen: ‘:Auto’ en ‘:InvoerScherm'. Een niet nader aangegeven object roept als eerste de operatie ‘wijzigEigenaar( )’ aan van het object ‘:Auto’.

4 Observer in Layers

Gegeven is het onderstaande, onvolledige ontwerp.
De klasse TemperatuurWindow moet, d.m.v. het Observer Framework, de temperatuur weergeven in de Interfacelaag. Gebruik gemaakt wordt van een grafische ontwikkelomgeving waarin het de bedoeling is dat gebruik gemaakt wordt van een TForm. Een nieuw window in de interfacelaag moet dan afgeleid worden van deze klasse TForm. Ter vereenvoudiging nemen we aan dat toch gebruik gemaakt wordt van cout voor het weergeven van informatie in zo'n window, alleen moet steeds de operatie show() worden aangeroepen als informatie is getoond d.m.v. cout.



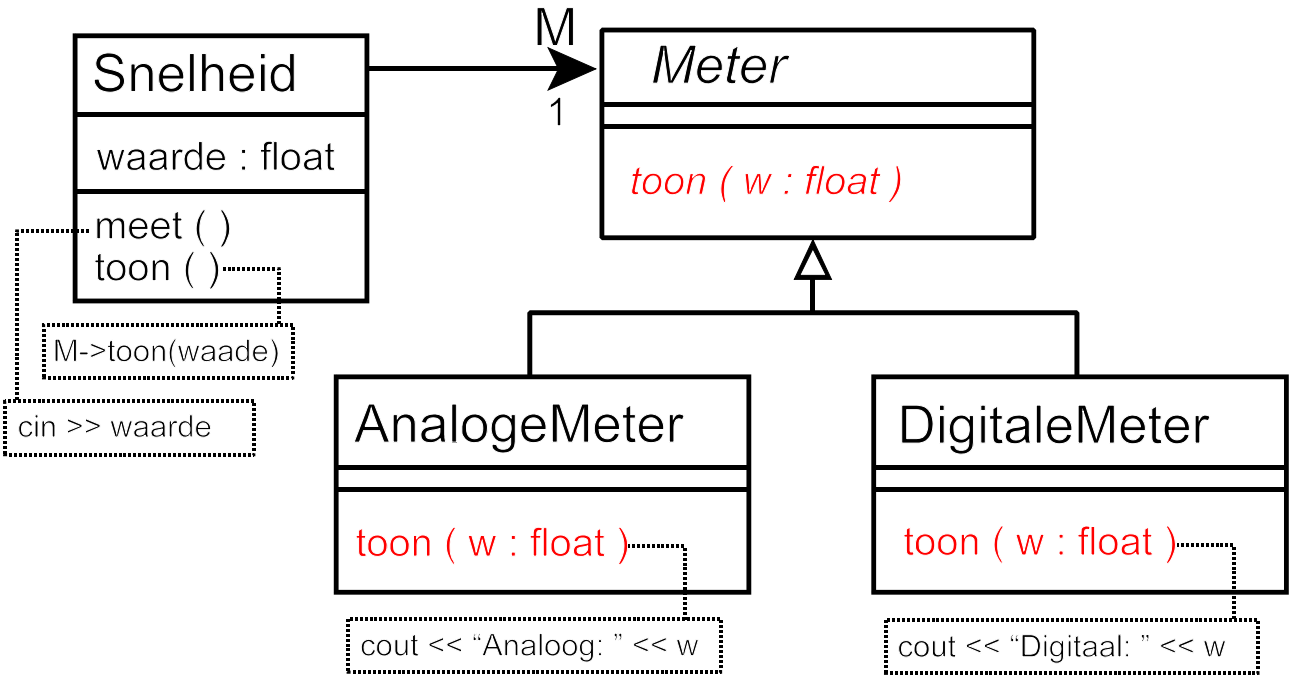
a) Maak het ontwerp af.

b) Realiseer het ontwerp in C++.
Gebruik het Observer-Framework op de laatste bladzijde.
De realisatie moet de onderstaande main-functie hebben:

```
int  main ( )
{
   TemperatuurImpl   T;
   TemperatuurWindow   W ( &T );
   RobotImpl   R ( &T );
   R.checkTemp ( );
   return  0;
}
```

5 Decorator

Gegeven is het onderstaande klassemodel.



De volgende implementatie is gegeven:

```
#include "iostream.h"

class  Meter
{
    public:   virtual ~Meter ( ) { }

              virtual  void  toon ( float  w ) = 0;
};

class  AnalogeMeter : public  Meter
{
    public:   virtual  void  toon ( float  w )  { cout << "Analoog=>> " << w << endl; }
};

class  DigitaleMeter : public  Meter
{
    public:   virtual  void  toon ( float  w )  { cout << "Digitaal=>> " << w << endl;}
};

class  Snelheid
{
    private:  float      waarde;
              Meter*     M;

    public:   Snelheid  ( Meter*  m ) : waarde ( 0 ),  M ( m ) { }
              virtual  ~Snelheid ( ) { }

              virtual  void  meet ( )     { cin>>waarde;          }
              virtual  void  toon  ( )    { M->toon ( waarde );   }
};
```

Er wordt nu gevraagd om de volgende nieuwe meters te definieren die extra eigenschappen hebben:

1. analoge en digitale meters met een omlijsting (een “border”),
2. analoge en digitale meters die vóóraf een zelf op te geven text afdrukken,
3. analoge en digitale meters die achteraf een zelf op te geven text afdrukken.

Het moet ook mogelijk zijn om analoge en digitale meters te gebruiken die combinaties van extra eigenschappen bezitten, zoals analoge en digitale meters die zowel vóóraf een zelf op te geven text afdrukken als ook een omlijsting hebben. Gebruik een Decorator voor een ontwerp dat flexibel is en gemakkelijk uitgebreid kan worden met meters met nog andere extra eigenschappen.
Gebruik de volgende main-functie:

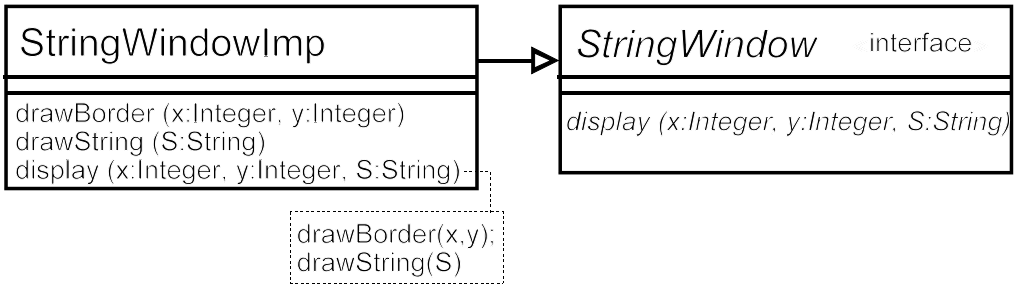
```
int  main ( )
{
   Meter*  M =  new  BorderDecorator
                    ( new  VoorTextDecorator
                          (   new  NaTextDecorator
                                 (   new DigitaleMeter,
                                     "Datum: 12 mei 2005\n"
                                 ),
                              "De snelheid is:\n"
                          )
                    );
   Snelheid  S ( M );
   S.meet ( );
   S.toon ( );
   return  0;
}
```

a) Geef het volledige klassemodel.

b) Geef de code in C++

6 Template Method

a) Gegeven is het onderstaand (niet volledige) klassemodel:



Opm: de werking van de operaties drawBorder() en drawString() is niet van belang

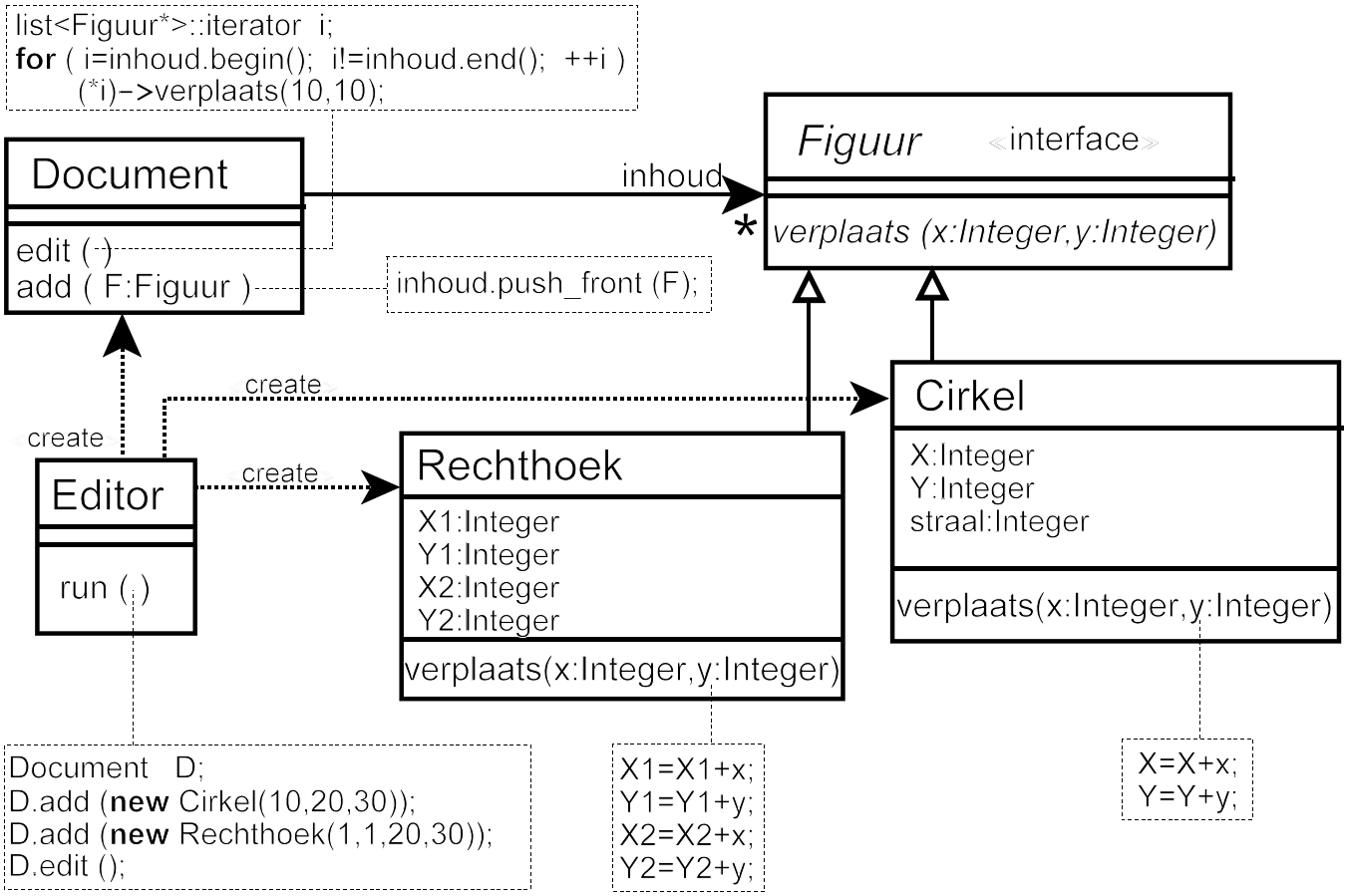
Maak van de operatie StringWindow::display(**int** x,**int** y,string S) een Template Method z.d.d. de operaties drawBorder en drawString primitieve operaties hiervan zijn.
Geef ***alleen*** van de ***klasse(n)*** waarin de Template Method ***en*** de primitieve operaties hiervan voorkomen de ***zo volledig mogelijke*** code in C++.

b) Gegeven de volgende 2 klassen van sorteermethoden: SortUp en SortDown. De methoden wijken maar op een enkel puntje van elkaar af. Gebruik een Template Method om deze 2 klassen met elkaar te combineren.

```
class  SortUp
{   public:  void sort ( int v [ ], int n )
                {   for ( int g = n / 2; g > 0; g /= 2 )
                      for ( int i = g; i < n; i++ )
                          for ( int j = i - g; j >= 0; j -= g )
                              if ( v [ j ] > v [ j + g ] )  swap ( v [ j ], v [ j + g ] );
                }
    private: void swap ( int &a, int &b )
                {   int t = a;
                    a = b;
                    b = t;
                }
};

class  SortDown
{   public:  void sort ( int v [ ], int n )
                {   for ( int g = n / 2; g > 0; g /= 2 )
                      for ( int i = g; i < n; i++ )
                          for ( int j = i - g; j >= 0; j -= g )
                              if ( v [ j ] < v [ j + g ] )  swap ( v [ j ], v [ j + g ] );
}
    private: void swap ( int &a, int &b )
                {   int t = a;
                    a = b;
                    b = t;
                }
};
```

7 Abstract Factory

Gegeven is het onderstaand (niet volledige) klassemodel:



Verder zijn nog de volgende constructoren gegeven in C++:

```
Rechthoek::Rechthoek(int x1, int y1, int x2, int y2) : X1(x1), Y1(y1), X2(x2), Y2(y2) {
}
Cirkel::Cirkel ( int x, int y, int r ) : X(x), Y(y), straal(r) { }
```

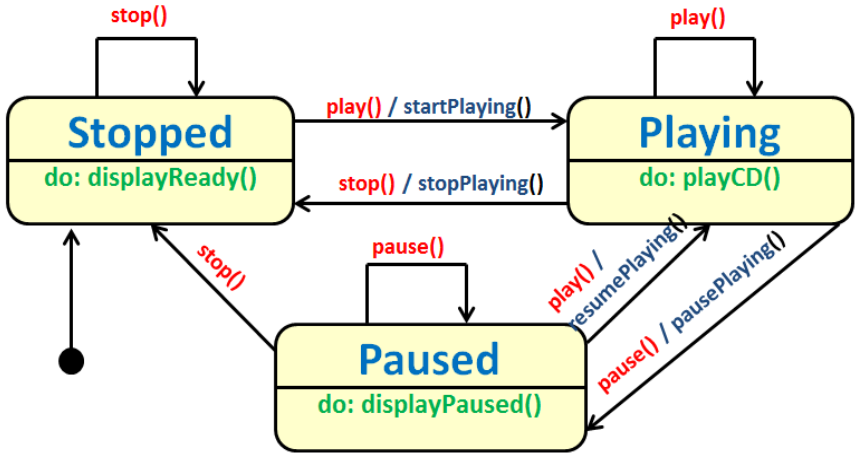
Gebruik het Abstract Factory design pattern (met Factory Methods) voor het onafhankelijk maken van de klasse Editor van de manier van het creëren van objecten.
Geef alleen een ***zo volledig mogelijk*** klassemodel.

De volgende main-functie moet kunnen worden gecompileerd:

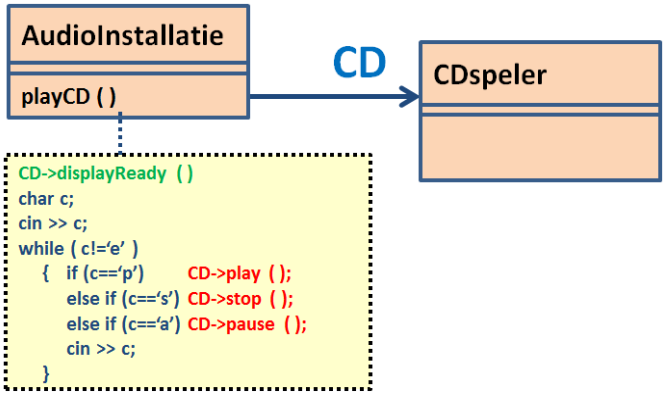
```
int main ( )
{   Factory1    F;
    Editor      E;
    E.run ( &F );
    return 0;
}
```

8 State

Gegeven is de onderstaande (sterk vereenvoudigde) statemachine van een CD speler:



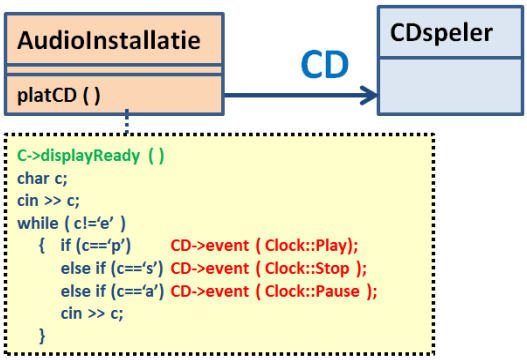
a) Maak een ontwerp (klassemodel) op basis van het State Design Pattern.
De volgende klassen moeten deel uitmaken van het ontwerp:



b) Codeer het ontwerp in C++.
Gebruik de volgende main functie:

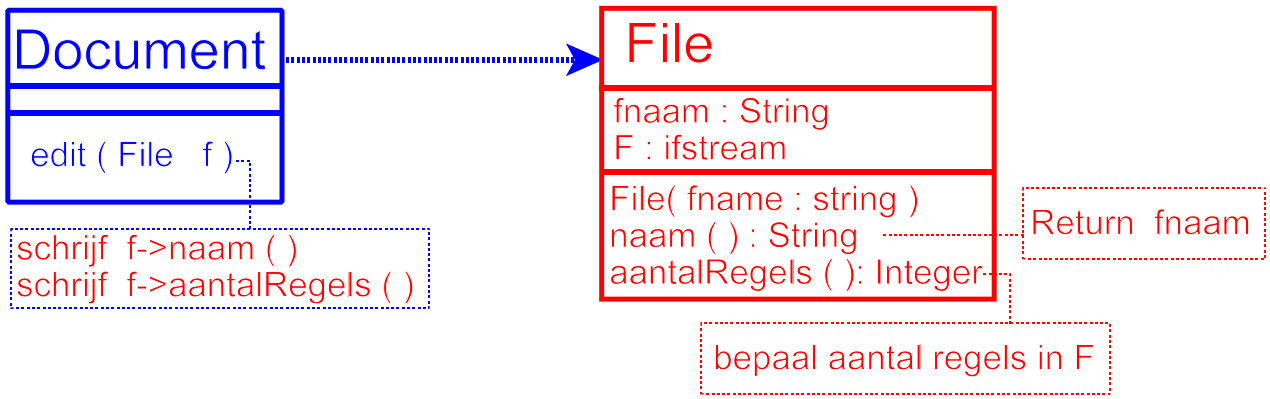
```
int   main( )
{  AudioInstallatie   M;
   M.playCD ( );
   return 0;
}
```

c) Maak nu een ontwerp d.m.v. een Transition Table.
Gebruik dezelfde main functie als in b maar nu met de volgende klassen:



9 Proxy

Gegeven is het onderstaande klassemodel:



Voor dit model is de onderstaande implementatie gegeven:

```
#include  <iostream.h>
#include  <fstream.h>
#include  <string>

using   namespace   std;

class   File
{
    private:  string       fnaam;
              ifstream     F;

    public:   File ( string   fname ) : fnaam ( fname ),  F ( fname.c_str( ) )  {  }

              virtual string  naam ( ) const { return   fnaam; }
              virtual  long  aantalRegels ( )
                    {   long   aantal=0;
                        char   c;
                        while  ( F.get ( c ) )     if ( c=='\n' )   ++aantal;
                        F.clear ( );
                        F.seekg ( 0 );
                        return    aantal;
                    }
};

class Document
{
    public:   virtual  void  edit ( File*  f ) { cout   <<   f->naam ( )   <<   f->aantalRegels ( ); }
};

void   main ( )
{
    File*  F = new   File ( "Proxy.cpp" );
    Document   D;
    D.edit ( F );
}
```

Het blijkt dat de klasse “File” een “goedkope” operatie heeft: “File::naam ( ):String” en een (soms) “dure” operatie: “File::aantalRegels ( ):Integer”.
Gebruik het Proxy design pattern voor het controleren van de toegang tot objecten van de

klasse “File”, zodanig dat een dure operatie alleen dan wordt aangeroepen als het ook echt nodig is.

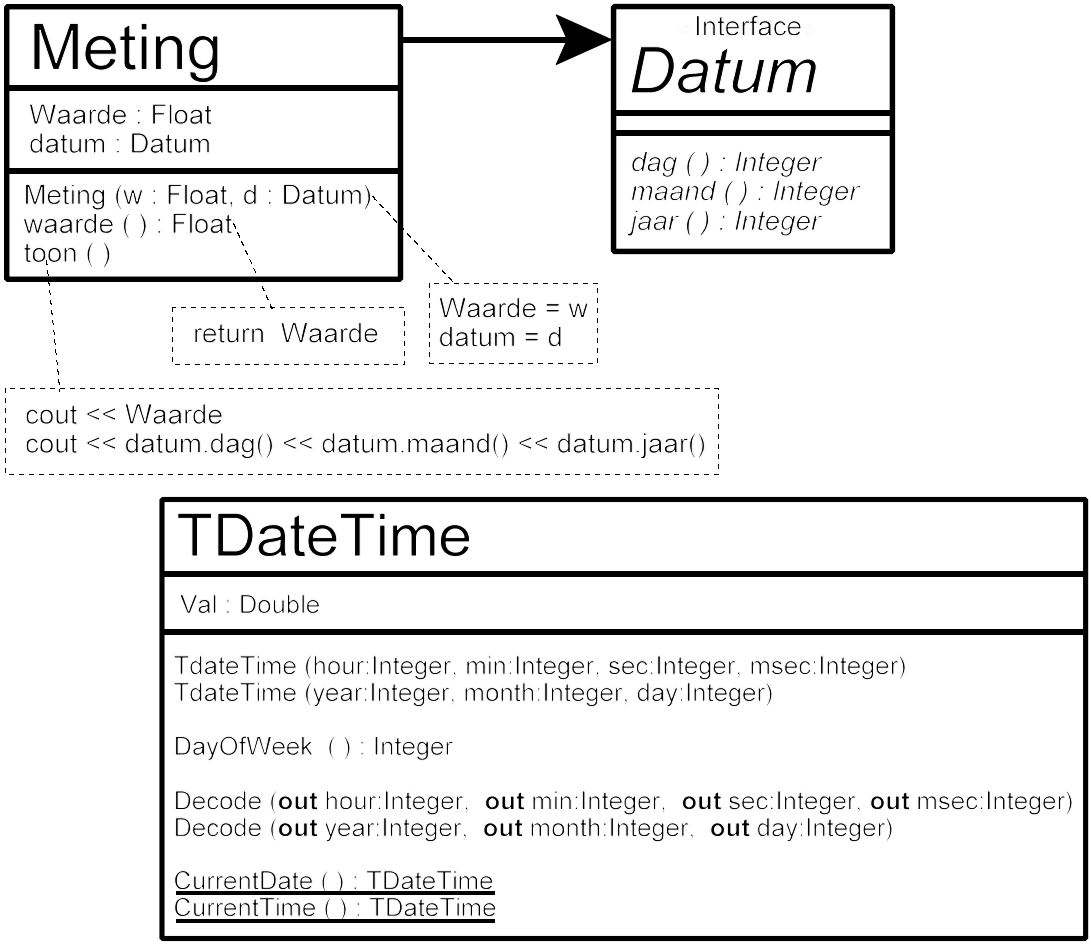
a) Geef het volledige klassemodel.

b) Geef de code in C++.
De volgende main-functie moet kunnen worden gecompileerd:

```
void   main ( )
{
    File*   F  =  new  FileProxy ( "Proxy.cpp" );
    Document   D;
    D.edit ( F );
}
```

10 Adapter

Gegeven is de onderstaande klasse “Meting” die voorkomt in een bestaand softwaresysteem waarvan de code ontbreekt. Zoals onderstaand klassemodel laat zien is deze klasse afhankelijk van de interface “Datum”. Om dit softwaresysteem nu te laten werken moet een concrete klasse ontworpen worden die daarvoor gebruikt kan worden. Het is de bedoeling dat daarvoor de klasse “TDateTime” wordt gebruikt omdat deze de benodigde functionaliteit levert en direct beschikbaar is in de bibliotheek van de Borland ontwikkelomgeving. Gebruik het Adapter design pattern voor dit ontwerp.



Geef een zo volledige mogelijk klassemodel in de volgende twee gevallen:

- de Adapter is een klassen-adapter
- de Adapter is een objecten-adapter

11 Bridge

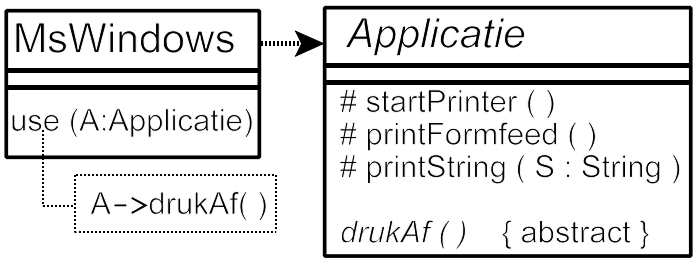
Gevraagd wordt een programma te ontwerpen waarmee met MsWindows verschillende soorten applicaties kunnen worden gebruikt (WordPerfect en Notepad) waarmee text moet kunnen worden afgedrukt met minstens 2 verschillende soorten printers: HP printers en Epson printers.

De in de bibliotheek beschikbare klasse ***HpPrinter*** biedt daartoe de volgende operaties:
1. **HpPrinter::init ( )** Hiermee moet de printer voor het printen worden geïnitialiseerd.
2. **HpPrinter::print (string S)** Hiermee kan een string S worden afgedrukt op het papier.

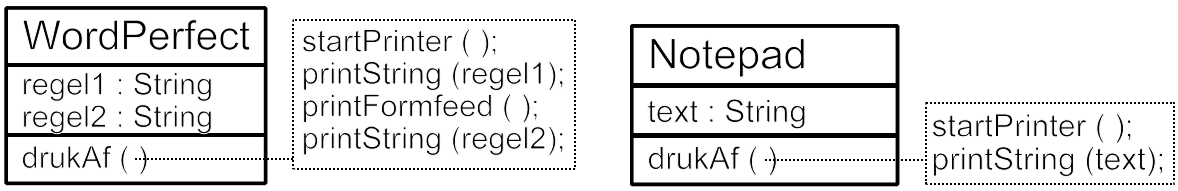
Let op: voor het printen van een formfeed op deze printer moet de string "**\f**" worden afgedrukt.

De in de bibliotheek beschikbare klasse ***EpsonPrinter*** biedt daartoe de operaties:
1. **EpsonPrinter::startUp ( )** Hiermee moet de printer voor het printen worden gestart.
2. **EpsonPrinter::formfeed ( )** Hiermee moet een eventuele formfeed worden afgedrukt.
3. **EpsonPrinter::printStr (string S)** Hiermee kan een string S worden afgedrukt op het papier.

Gegeven is nog het nevenstaande klassemodel:



De applicatie WordPerfect bevat de tekst in twee regels (2 strings), deze text moet met een formfeed worden afgedrukt. De applicatie Notepad bevat de text in één string en wordt zonder extra formfeed afgedrukt. Zie de onderstaande ***(niet volledige)*** figuuur.



WordPerfect en Notepad zijn dus beiden ***speciale*** applicaties.
De volgende main-functie moet kunnen worden gecompileerd:

```
void  main ( )
{   HpAdapter       Hp;
    EpsonAdapter    Epson;
    WordPerfect     Wp ( &Hp, "Dit is de beste", "tekstverwerker\n" );
    Notepad         Np  ( &Epson, "Dit is een eenvoudige tekstverwerker\n" );
    MsWindows       W;
    W.use ( Wp );
    W.use ( Np );
}
```

a) Gebruik een Bridge voor het realiseren van dit ontwerp. Gebruik in de implementaties van de Bridge ook Adapters (klassen-adapters) voor het (her)gebruiken van de klassen HpPrinter en EpsonPrinter.

   Geef alleen een ***zo volledig mogelijk*** klassemodel (met 9 klassen), dus ook de algoritmen in C++ van ***alle*** operaties (*inclusief de constructoren*) nauwkeurig beschrijven.

b) Noem 2 specificieke ***voordelen*** van een klassen-adapter t.o.v. een objecten-adapter.

12 Knowledge Level

Het afdelingshoofd van de afdeling Medische Fysica van het ziekenhuis “Lijpenberg” heeft opdracht gegeven tot de ontwikkeling van een softwaresysteem t.b.v. de registratie van het onderhoud aan de medische apparatuur in het ziekenhuis.

Ieder soort apparaat (een apparaattype) krijgt daartoe een uniek apparaattypenummer (atnr) en vastgelegd wordt de beschrijving, de leverancier, de producent en het type. Van een apparaattype kunnen meerdere exemplaren (apparaten) aanwezig zijn in het ziekenhuis. Ieder apparaat krijgt een uniek apparaatnummer (anr) en vastgelegd wordt van welke apparaattype het is, de aanschafdatum, de prijs en het serienummer.

Om het onderhoud aan de apparatuur vast te leggen in het systeem worden er verschillende typen onderhoudsbeurt gedefinieerd (“onderhoudsbeurttypen”). Ieder type onderhoudsbeurt krijgt een unieke code en de beschrijving ervan wordt vastgelegd, de periode (in dagen) en de tijdsduur van de onderhoudsbeurt (in kwartieren). Een onderhoudstype behoort altijd tot precies één apparaattype. Voor iedere type onderhoudsbeurt wordt ook bepaald uit welk type taken deze bestaat (visuele inspectie, calibratie, electrische veiligheidstest, stralingsveiligheids test, etc). Een taaktype kan voorkomen bij meerdere onderhoudsbeurttypen en wordt gedefinieerd door een code en een beschrijving van de taak met het daarbij benodigde materiaal.

Een (uitgevoerde!) onderhoudsbeurt wordt uitgevoerd op één apparaat en behoort bij precies 1 onderhoudsbeurttype. Van een uitgevoerde onderhoudsbeurt wordt vastgelegd op welke datum die heeft plaatsgevonden en wat het resultaat ervan is. Een uitgevoerde onderhoudsbeurt bestaat uit een aantal taken. Zo’n taak behoort altijd bij precies 1 (uitgevoerde) onderhoudsbeurt en precies 1 gedefinieerd taaktype. Bij een taaktype kunnen dus meerdere (uitgevoerde) taken behoren.

Iedere taak krijgt een uniek taaknummer en er wordt vastgelegd bij welke (uitgevoerde) onderhoudsbeurt deze behoort, van welk taaktype deze is en wat het resultaat is van de verrichting.

Geef een klassemodel waarin duidelijk onderscheid wordt gemaakt in een Knowledge Level en een Operational Level.

Let op: Vermeld ook de benodigde constraints.

Observer Framework:

Observer.h

```
#ifndef   __Observer_H
#define   __Observer_H

#include  <list>

using  namespace  std;

class    Subject;

class    Observer
{
    private:  Subject*     S;

    protected:   Subject*     getSubject ( )   const     { return S; }

    public:   Observer (Subject* s);
              virtual    ~Observer ( );
              virtual  void  update ( ) = 0;
};

class    Subject
{
    private:  list<Observer*>    L;

    protected:   virtual  void  notify ( );

    public:   Subject ( ) { }
              virtual  ~Subject ( ) { }
              virtual  void  insert       (Observer* s)  { L.push_front(s); }
              virtual  void  remove     (Observer* s)  { L.remove(s);       }
};

#endif
```

Observer.cpp

```
#include  "observer.h"

void   Subject::notify ( )
{
    for  ( list<Observer*>::iterator  i=L.begin( );  i!=L.end( );  ++i )     (*i)->update( );
}

Observer::Observer ( Subject* s ) : S(s)
{
    getSubject( )->insert(this);
}

Observer::~Observer ( )
{
    getSubject( )->remove ( this );
}
```

3. Practicumhandleiding

3.1 Inleiding

Het practicum is niet een op zichzelf staande oefening maar dient als directe ondersteuning van de theorielessen. Het practicum bestaat uit 5 opdrachten. De volgende tabel geeft aan wanneer de opdrachten moeten zijn afgerond.

Wk	Theorie	Bron	Theorie opgaven	Practicum
1	- Layer - Model View Controller - Observer - Abstract Server - Factory Method - Template Method - Abstract Factory	Bosch: blz. 98 t/m 103 Bosch: blz. 104 t/m 113 Gamma: 5.7, Martin: H24 Gamma: 3.3 Gamma: 5.10, Martin: H14 Gamma: 3.1, Martin: H21	1. Observer / MVC 2. Lagenstructuur 3. Lagenstructuur & Abstract Server 4. Observer in Layers 5. Decorator	**1: Layers**
2	- Singleton - Facade - Composite - Decorator	Gamma: 3.5, Martin: H16 Gamma: 4.5, Martin: H15 Gamma: 4.3, Martin: H23 Gamma: 4.4	6. Template	**2: Composite & Factory Method**
3	- State - Strategy	Gamma: 5.8, Martin: H29 Gamma: 5.9, Martin: H14	7. Abstract Factory	**3: Facade**
4	- Proxy - Adapter	Gamma: 4.7, Martin: H26 Gamma: 4.1, Martin: H25	8. State	**4: Decorator**
5	- Bridge - Command	Gamma: 4.2, Martin: H25 Gamma: 5.2, Martin: H13	9. Proxy	**5: Robot**
6	- Type Object - Knowledge level	Johnson: 2.5 én 2.6 Johnson: 2.5 én 2.6	10. Adapter	
7			11. Bridge 12. Knowledge level	

3.2 De opdrachten

Opdracht 1: Layers

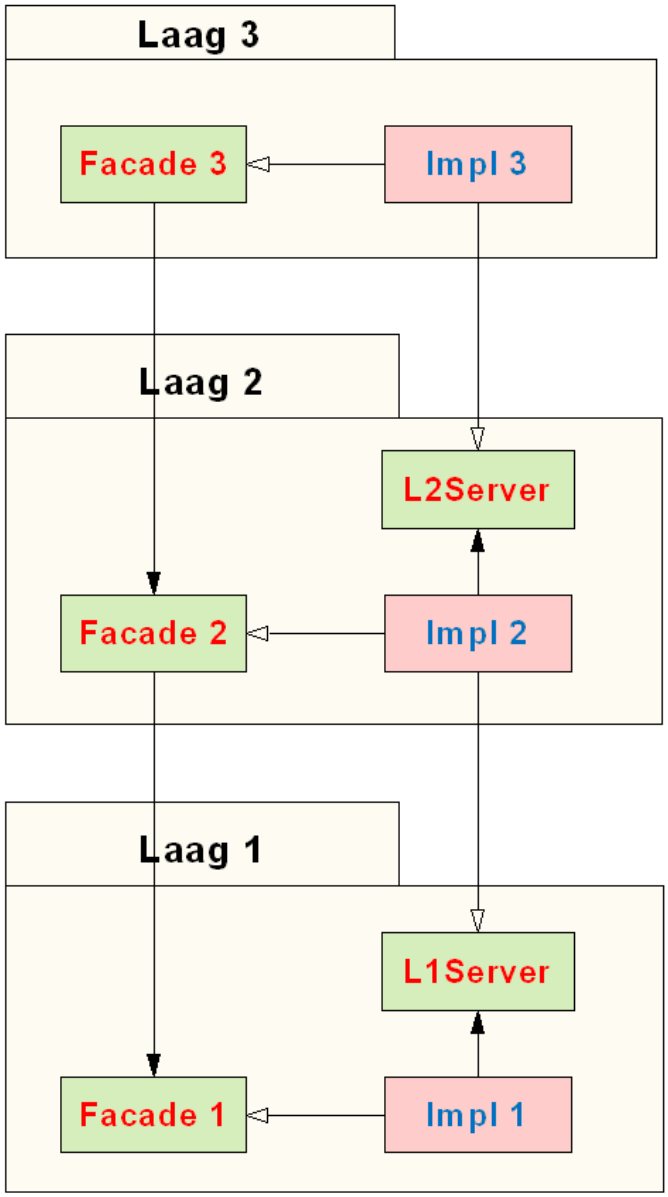
Gegeven de lagenstructuur in nevenstaande figuur.

Gevraagd wordt in C++ een implementatie te geven van de gehele lagenstructuur. Gebruik de onderstaande main-functie:

```
int main ( )
{   Laag1Impl  L1;
    Laag2Impl  L2;
    Laag3Impl  L3;

    L1.connectHigh ( &L2 );
    L2.connectLow ( &L1 );
    L2.connectHigh ( &L3 );
    L3.connectLow ( &L2 );

    L3.service ( );

    return 0;
}
```

De output van dit programma moet zijn:

```
Laag 3 start...
Laag 2 start...
Laag 1 start...
Laag 2 start...
Laag 3 start...
Laag 3 eindigt...
Laag 2 eindigt...
Laag 1 eindigt...
Laag 2 eindigt...
Laag 3 eindigt...
```

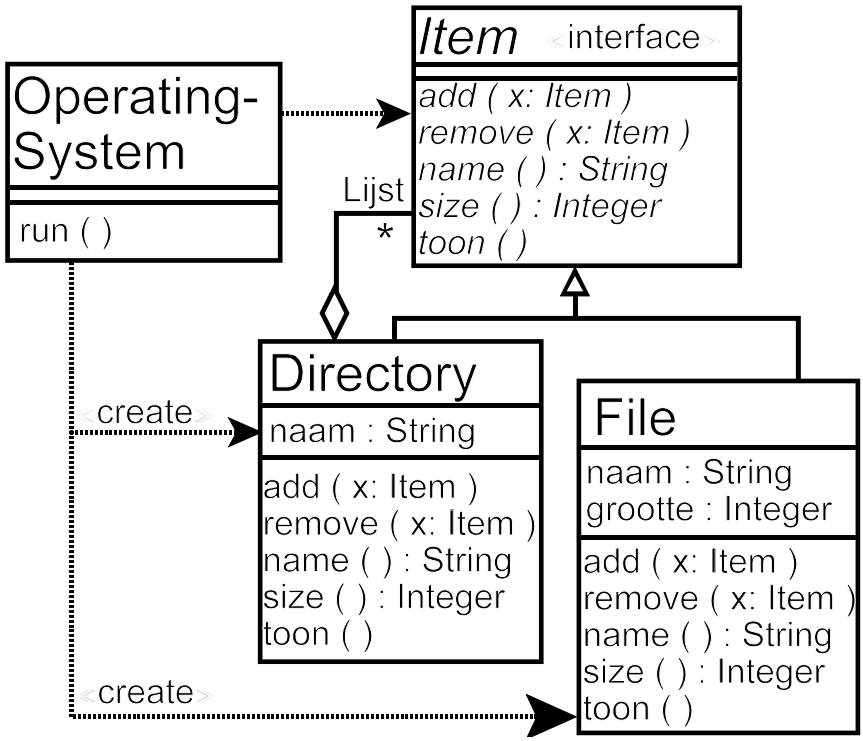


Opdracht 1

Geef een implementie in C++ van de gehele lagenstructuur.

Opdracht 2: Factory Method & Composite

Gegeven het volgende klassemodel:



Verder zijn nog de volgende implementaties van de operaties in C++ gegeven:

```
File::File ( string n, int g ) : naam ( n ), grootte ( g ) { }
string   File::name ( ) const   { return naam;                          }
int      File::size ( )         { return grootte;                       }
void     File::toon ( )         { cout<<naam<< " "<<grootte<<"\n";  }
void     File::add ( Item* x )  {                                       }
void     File::rem ( Item* x )  {                                       }

Directory::Directory ( string n ) : naam ( n ) { }
void              Directory::add ( Item* x )  { Lijst.push_front(x);    }
void              Directory::rem ( Item* x )  { Lijst.remove(x);     }
list<Item*> &     Directory::lijst ( )        { return Lijst;           }
void              Directory::toon ( )         { /*.....................*/  }
string            Directory::name ( ) const   { return naam;            }
int               Directory::size ( )         { /*.....................*/  }

void  OperatingSystem::run ( )
{
    Item* f = new File ("1.txt", 1000);
    Item* D = new Directory ("dir A");
    D->add ( f );
    D->toon ( );
}
```

Opdracht 2

a) Geef een volledige implementie in C++ van het klassemodel.

De operatie “Directory::size ( )” moet de totale grootte van een directory retourneren (inclusief zijn eigen grootte, die altijd 1024 byte is).

De operatie “Directory::toon ( )” moet zowel de naam als ook de totale grootte van een directory weergeven (inclusief zijn eigen grootte, die altijd 1024 byte is).

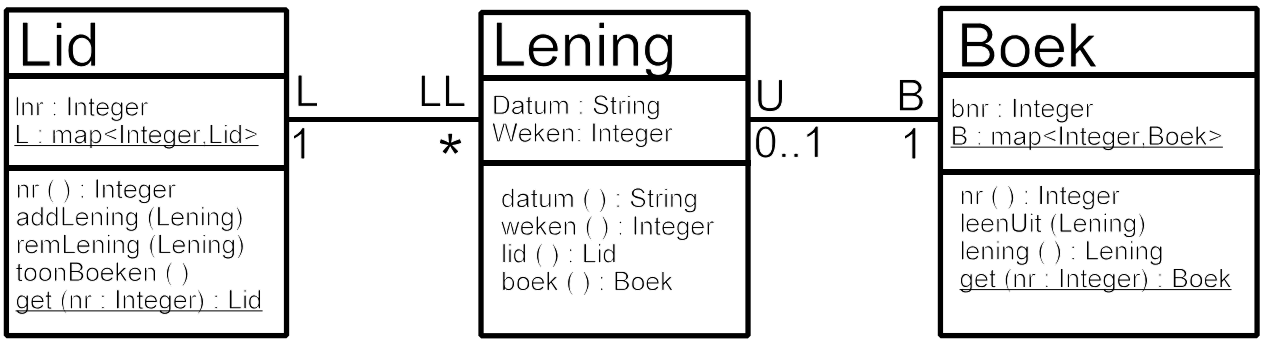
b) Maak van de operatie “OperatingSystem::run ( )” een Template Method door het gebruiken van Factory Methods in deze operatie voor het aanmaken van de objecten van de klasse “File” en de klasse “Directory”.
Definieer daartoe de nieuwe klasse “SimpleOperatingSystem”.

De volgende main-functie moet kunnen worden gecompileerd:

```
int  main ( )
{ SimpleOperatingSystem   OS;
  OS.run ( );
  return 0;
}
```

Opdracht 3: Facade als Singleton

Gegeven is het onderstaande gedeelte van een klassemodel dat opgevat wordt als een subsysteem:



Het klasse-attribuut “Lid::L” dient voor het bijhouden van alle aanwezige leden. Deze map heeft als key-waarden het nummer van een lid. De klasse-operatie “Lid::get (nr : Integer) : Lid” retourneert het lid met nummer “nr” uit de map “L”. Dezelfde constructie heeft de klasse Boek met het klasse-attribuut “B” en de klasse-operatie “Boek::get (nr : Integer) : Boek”.

De onderstaande implementatie van het subsysteem is gegeven:

Bibliotheek.h

```
#ifndef __Bibliotheek_H
#define __Bibliotheek_H

#include  <map>
#include  <list>
#include  <string>

using  namespace  std;

class   Lid;
class   Boek;

class   Lening
{
   private:  string    Datum;
             int       Weken;
             Lid*      L;
             Boek*     B;

   public:   Lening ( Lid*  l,  Boek*  b,  string  d,  int  w ) : L ( l ), B ( b ), Datum ( d ), Weken ( w ) { }
             virtual  ~Lening  ( ) { }

             virtual   string    datum   ( ) const  { return  Datum;   }
             virtual   int       weken   ( ) const  { return  Weken;   }
             virtual   Lid*      lid     ( ) const  { return  L;       }
             virtual   Boek*     boek    ( ) const  { return  B;       }
};
```

```
class   Lid
{
   private:  int     lnr;
             list<Lening*>   LL;

             static  map< int, Lid* >   L;

   protected: Lid ( int  n ) : lnr ( n )  { L [ lnr ] = this; }

   public:   virtual  ~Lid ( ) { }

             virtual    int     nr   ( ) const  { return  lnr; }
             virtual    void    addLening  ( Lening*  u )  {  LL.push_front ( u );  }
             virtual    void    remLening  ( Lening*  u )  {  LL.remove ( u );      }
             virtual    void    toonBoeken ( );

             static     Lid*    get ( int  nr );
             static     void    dump ( Lid*  l );
};

class   Boek
{
   private:  int     bnr;
             Lening*  U;

             static  map< int, Boek* >  B;

   protected:   Boek ( int  n ) : bnr ( n ),  U ( 0 )  { B [ bnr ] = this; }

   public:   virtual ~Boek ( ) { }

             virtual    int         nr             ( ) const { return  bnr;   }
             virtual    Lening*     lening         ( ) const { return  U;     }
             virtual    void        leenUit        ( Lening*  u )  {  U = u;  }
             virtual    void        retourneer     ( )         {   U=0;       }

             static     Boek*    get ( int  nr );
             static     void     dump ( Boek*  b );
};

#endif
```

Bibliotheek.cpp

```
#include  "Bibliotheek.h"
#include  <iostream.h>

Lid*  Lid::get ( int  nr )
{
    if  ( L.count ( nr ) )   return L [ nr ];
    else  return L [ nr ] = new  Lid ( nr );
}

void  Lid::dump ( Lid*  l )
{
    L.erase ( l->nr ( ) );  delete  l;
}

Boek*  Boek::get ( int  nr )
{
    if  ( B.count ( nr ) )   return B [ nr ];
    else  return B [ nr ] = new  Boek ( nr );
}

void  Boek::dump ( Boek*  b )
{
    B.erase ( b->nr ( ) );  delete  b;
}

void  Lid::toonBoeken ( )
{
    list<Lening*>::iterator i;
    for ( i = LL.begin ( );  i != LL.end ( );  ++i )
        if ( (*i) -> lid ( ) == this )
                cout << (*i) -> boek ( ) -> nr ( ) << (*i) -> datum ( ) << (*i) -> weken ( );
}

map< int, Lid* >        Lid::L;
map< int, Boek* >       Boek::B;
```

Opdracht 3

Ontwikkel een Facade (als Singleton) voor dit subsysteem met de volgende 3 functies:

1. het uitlenen van een boek, met achtereenvolgens de volgende parameters:
   - het nummer van het lid
   - het nummer van het boek
   - de datum van uitlenen
   - het aantal weken van de uitlening
2. het retourneren van een boek, met uitsluitend het boeknummer als parameter
3. het tonen van de gegevens van alle leningen van een lid, met uitsluitend het lidnummer als parameter

a) Maak een ontwerp in de vorm van een klassemodel.
Let op! Geef nauwkeurig aan:
- wat de attributen van de klassen zijn
- wat de operaties van de klassen zijn, inclusief parameters en typen
- van iedere operatie het algoritme

b) Geef een sequentiediagram waarin de werking van het ontwerp van vraag a wordt weergegeven. Hierin moet van iedere (niet abstracte) klasse 1 object voorkomen.

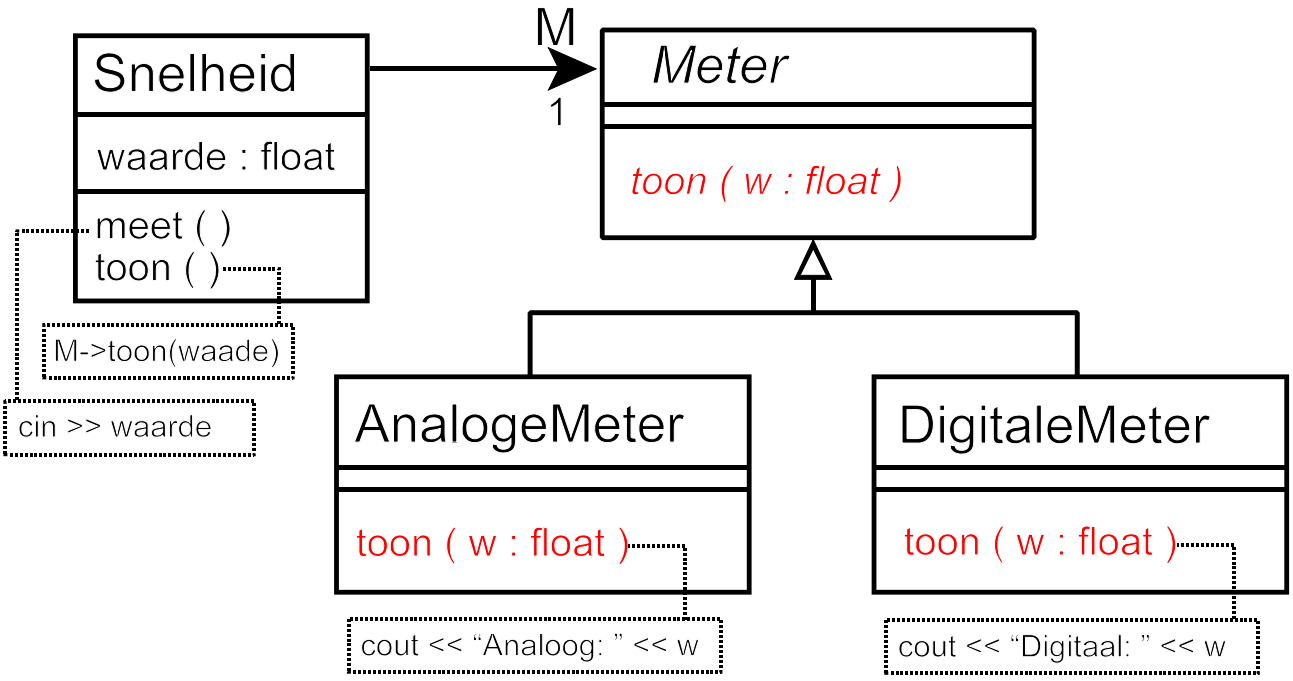
c) Realiseer het ontwerp in C++. Deze realisatie moet de onderstaande main-functie hebben:

```
#include "LeningFacadeImpl.h"

class  Toepassing
{  public:    virtual void gebruik ( LeningFacade*  F )
                    {  F->leenUit ( 7, 13, "10 mei 2003", 2 );
                       F->leenUit ( 7, 17, "15 mei 2003", 3 );
                       F->retourneer ( 17 );
                       F->toonBoeken ( 7 );
                    }
};

void  main ( )
{  Toepassing   T;
   T.gebruik ( LeningFacadeImpl::get ( ) );
}
```

De Facade moet een interface zijn zonder enige detail van een implementatie.
Maak gebruik van de hierboven gegeven implementatie van dit subsysteem.

Opdracht 4: Decorator

Gegeven is het onderstaande klassemodel.



De volgende implementatie is gegeven:

```
#include "iostream.h"

class  Meter
{
    public:    virtual ~Meter ( ) { }

               virtual  void  toon ( float  w ) = 0;
};

class  AnalogeMeter : public  Meter
{
    public:    virtual  void  toon ( float  w )  { cout << "Analoog=>> " << w << endl; }
};

class  DigitaleMeter : public  Meter
{
    public:    virtual  void  toon ( float  w )  { cout << "Digitaal=>> " << w << endl;}
};

class  Snelheid
{
    private:   float       waarde;
               Meter*     M;

    public:    Snelheid  ( Meter*  m ) : waarde ( 0 ),  M ( m ) { }
               virtual  ~Snelheid ( ) { }

               virtual  void  meet ( )     { cin>>waarde;            }
               virtual  void  toon  ( )    { M->toon ( waarde );     }
};
```

Opdracht 4

Er wordt nu gevraagd om de volgende nieuwe meters te definieren die extra eigenschappen hebben:

1. analoge en digitale meters met een omlijsting (een “border”),
2. analoge en digitale meters die vóóraf een zelf op te geven text afdrukken,
3. analoge en digitale meters die achteraf een zelf op te geven text afdrukken.

Het moet ook mogelijk zijn om analoge en digitale meters te gebruiken die combinaties van extra eigenschappen bezitten, zoals analoge en digitale meters die zowel vóóraf een zelf op te geven text afdrukken als ook een omlijsting hebben. Gebruik een Decorator voor een ontwerp dat flexibel is en gemakkelijk uitgebreid kan worden met meters met nog andere extra eigenschappen.

De volgende code moet goed werkend worden gerealiseerd:

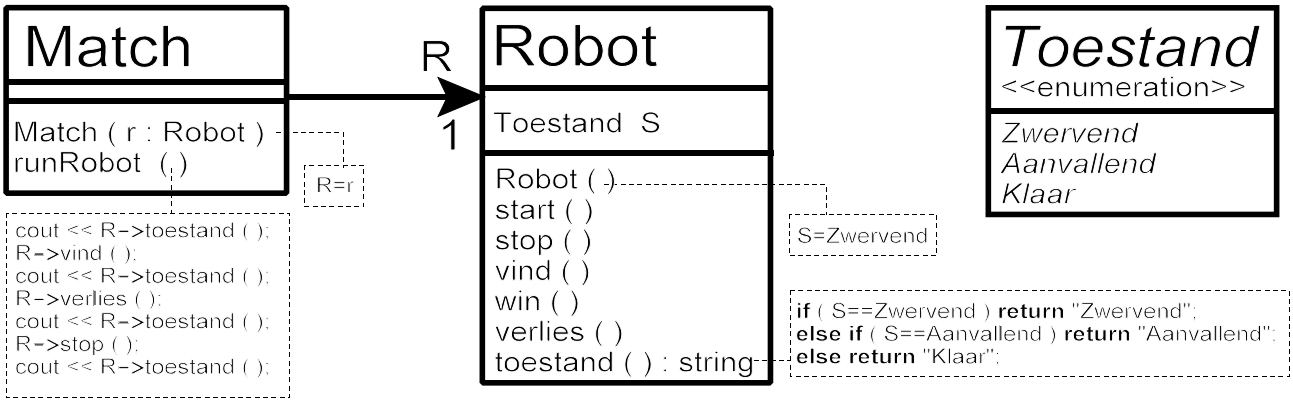
```
void  main ( )
{
   Meter*  M =  new  BorderDecorator
                    ( new  VoorTextDecorator
                          (   new  NaTextDecorator
                                (   new DigitaleMeter,
                                    "Datum: 12 mei 2005\n"
                                ),
                              "De snelheid is:\n"
                          )
                    );
   Snelheid  S ( M );
   S.meet ( );
   S.toon ( );
}
```

a) Geef het volledige klassemodel.

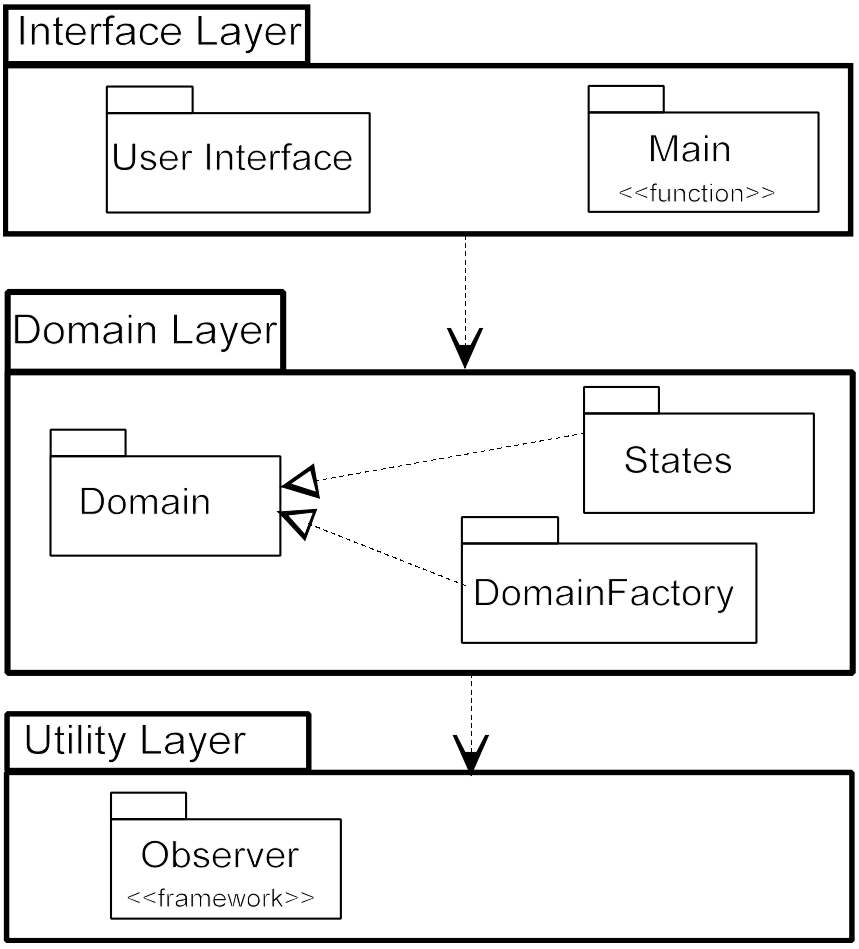
b) Geef de code in C++

Opdracht 5: Robot

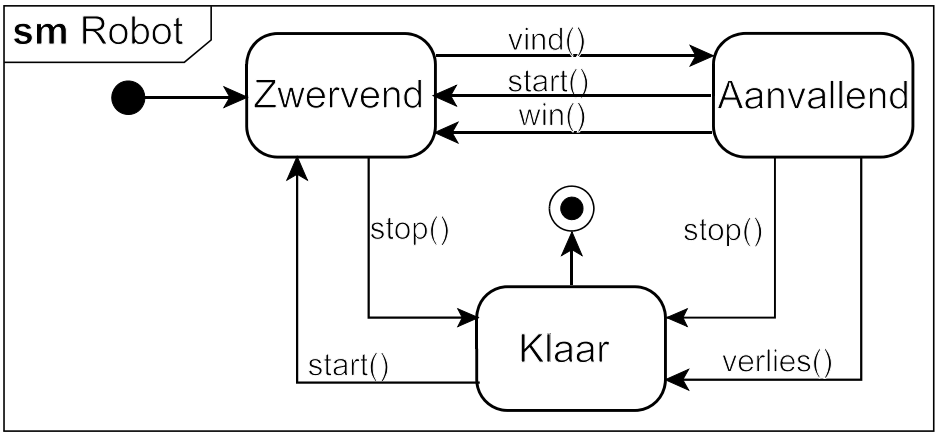
Het doel van deze opdracht is het toepassen van diverse patronen bij het ontwerp van een lagenstructuur. Gegeven is het onderstaande klassemodel van een robotwedstrijd.



Dit model vormt de basis voor de domeinlaag van de onderstaande lagenstructuur:



Er wordt gebruik gemaakt van een Observer voor het in de interfacelaag weergeven van de toestand van de robot, hierbij moet het framework op bladzijde 30 worden gebruikt. Verder wordt er een State gebruikt voor het realiseren van de toestanden van de robot. Het moet eenvoudig zijn, door het wijzigen van een DLL, de toestanden van de robot te veranderen. Een Abstract Factory moet de objecten van Match en Robot creëren.
Van de Robot is onderstaand (onvolledig) State Transition Diagram gegeven.



De volgende transition table geeft alle transities weer.

	Zwervend	**Aanvallend**	**Klaar**
start ( )	Zwervend	Zwervend	Zwervend
stop ( )	Klaar	Klaar	Klaar
win ( )	Zwervend	Zwervend	Klaar
vind ( )	Aanvallend	Aanvallend	Klaar
verlies ( )	Zwervend	Klaar	Klaar

Opdracht 5

a) Geef een volledig klassemodel van het ontwerp met daarin ***alle*** details.

b) Geef een volledig packagediagram waarin alle klassen in de packages worden genoemd en waarin alle afhankelijkheden tussen de packages zijn aangegeven.

c) Bereken alle ontwerpparameters en geef in een grafiek de abstractie als functie van de instabiliteit van de packages weer.

d) Realiseer het ontwerp in C++.
Deze realisatie moet bestaan uit 5 DLL's en 1 executable.

e) Maak een nieuwe DLL voor de package State waarin één nieuwe State wordt gedefinieerd: "Verloren". Als de robot nu in de toestand "Aanvallend" is dan verandert de toestand van de robot als gevolg van de event "verlies()" in "Verloren". Bedenk ook zinnige transities voor de andere toestanden en events.

De bedoeling is dat door de introductie van deze nieuwe DLL de andere DLL's niet wijzigen.

Observer framework:

Observer.h

```
#ifndef   __Observer_H
#define   __Observer_H

#include  <list>

using  namespace  std;

class    Subject;

class    Observer
{
   private:  Subject*     S;

   protected:   Subject*     getSubject ( )   const     { return S; }

   public:   Observer (Subject* s);
             virtual    ~Observer ( );
             virtual  void  update ( ) = 0;
};

class    Subject
{
   private:  list<Observer*>    L;

   protected:   virtual  void  notify ( );

   public:   Subject ( ) { }
             virtual  ~Subject ( ) { }
             virtual  void  insert       (Observer* s)  { L.push_front(s); }
             virtual  void  remove     (Observer* s)  { L.remove(s);      }
};

#endif
```

Observer.cpp

```
#include  "observer.h"

void   Subject::notify ( )
{
   for  ( list<Observer*>::iterator  i=L.begin( );  i!=L.end( );  ++i )     (*i)->update( );
}

Observer::Observer ( Subject* s ) : S(s)
{
   getSubject( )->insert(this);
}

Observer::~Observer ( )
{
   getSubject( )->remove ( this );
}
```