LÖSNINGSFÖRSLAG till TENTAMEN i
Datateknik B , Designmönster, 5 poäng, fredag den 2 juni 2006

1. Nedanstående program ger en beskrivning av spelare i ett fotbollslag. Programmet är katastrofalt dåligt skrivet och uppfyller knappast några krav man kan ställa på objektorienterad programmering. Skriv om programmet så att det verkligen blir objektorienterat! Beskriv också hur din lösning tillämpar
- ”Open/Closed principle”
- ”Once-and-only-once-rule”

Kommentera även skillnader i ’cohesion’ och ’coupling’ mellan din lösning och den ursprungliga koden.

```
   #include<iostream>
#include<string>
#include<vector>
using namespace std;

enum EFooted { LEFT, RIGHT, BOTH };  // Best foot
enum EReaction { SLOW, FAST, LIGHTNING };

class Defender {
public:
    Defender(string n, int num, EFooted f)
    :name(n), type("Defender"), number(num), foot(f)
    { }

    int getNumber() const { return number; }

    string getFoot() {
      string s;
      switch(foot) {
         case LEFT: s=" left-footed"; break;
         case RIGHT: s= " right-footed"; break;
         case BOTH: s= " both-footed"; break;
      }
      return s;
    }

    void display(ostream &os=cout) {
      os << number << ' ' << name << ' ' << type;
    }
private:
    EFooted foot;
    string name;
    string type;
    int number;
}; // Defender
class Forward {
```

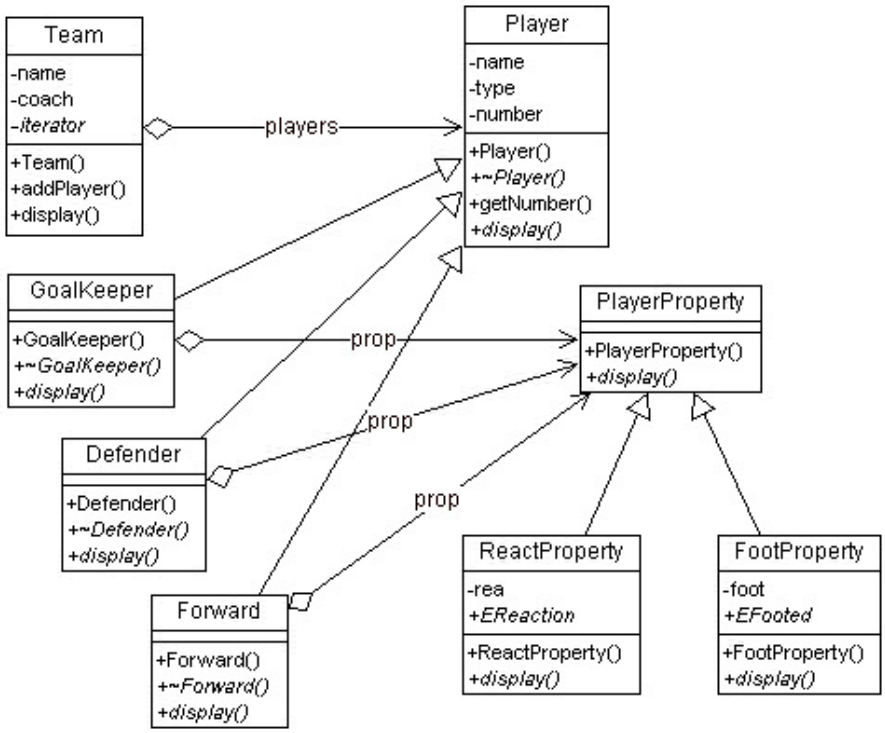
```
public:
    Forward(string n, int num, EFooted f)
    :name(n), type("Forward"), number(num), foot(f)
    { }

    string footToString() {
       switch(foot) {
         case LEFT: return " left-footed";
         case RIGHT: return " right-footed";
         case BOTH: return " both-footed";
       }
       return "";
    }

    int getNumber() const { return number; }

    void display(ostream &os=cout) {
      os << number << ' ' << name << ' ' << type;

    }
private:
    EFooted foot;
    string name;
    string type;
    int number;
}; // Forward

class GoalKeeper {
public:
    GoalKeeper(string n, int num, EReaction rea)
    :name(n), type("GoalKeeper"), number(num),
reaction(rea)
    { }

    string reactToString() {
       switch(reaction) {
         case SLOW: return " slow";
         case FAST: return " fast";
         case LIGHTNING: return " lightning-like";
       }
       return "";
    }

    int getNumber() const { return number; }

    void display(ostream &os=cout) {
      os << number << ' ' << name << ' ' << type;
    }
private:
    EReaction reaction;
    string name;
```

```
    string type;
    int number;
}; // Forward



class Team {
private:
   vector<Defender> defenders;
   vector<Forward> forwards;
   vector<GoalKeeper> goalKeepers;
   string name;
   string coach;

   typedef  vector<Defender>::iterator DefIt;
   typedef  vector<Forward>::iterator ForIt;
   typedef  vector<GoalKeeper>::iterator GolIt;

public:
   Team(string n, string c): name(n), coach(c) {}

   void addDefender(Defender d) {
         defenders.push_back(d);
   }

   void addForward(Forward f) {
        forwards.push_back(f);
   }

   void addGoalKeeper(GoalKeeper g) {
        goalKeepers.push_back(g);
   }

   void display(ostream &os=cout) {
     os << name << ":\n";
     GolIt firstG =
     goalKeepers.begin(),lastG=goalKeepers.end(), itG;
     for (itG=firstG; itG!=lastG; itG++ )  {
            itG->display(os);
            os << itG->reactToString() << endl;
     }
     DefIt firstD =
     defenders.begin(), lastD=defenders.end(),itD;
     for (itD=firstD; itD!=lastD; itD++ )  {
          itD->display(os);
          os << itD->getFoot() << endl;
     }
     ForIt firstF =
     forwards.begin(), lastF=forwards.end(),itF;
     for (itF=firstF; itF!=lastF; itF++ )  {
           itF->display(os);
```

```
            os << itF->footToString() << endl;
      }
      os<<"\ncoached by "<< coach <<'\n'<<endl;
   }
};

int main() {

   Team t1("Dream Team", "Dagge 'Demonen' Danielsson");
   GoalKeeper g1("Lennart 'Limmet' Larsson", 1, LIGHTNING);
   Forward f1("Kalle 'Klippan' Karlsson",4,BOTH);
   Forward f2("Sven 'Stinget' Svensson",8,RIGHT);
   Defender d1("Roffe 'Ruffis' Ragnarsson",12,LEFT);

   t1.addGoalKeeper(g1);
   t1.addForward(f1);
   t1.addForward(f2);
   t1.addDefender(d1);

   t1.display();
   cin.get();
   return 0;
}
```

(8 p)

Lösning



”Open/Closed principle”
Closed: Vi kapslar in det som ska vara stabila interface i två abstrakta basklasser, PlayerProperty och Player.
Open: Konkreta subklasser implementerar de olika variationerna. Vi kan programmera mot interfacen och enkelt lägga till nya konkreta subklasser.

”Once-and-only-once-rule”
Den ursprungliga koden bryter mot regeln i flera avseenden. Bl.a. genom att varje spelarklass implementerar funktionen

```
void display(ostream &os=cout) {
      os << number << ' ' << name << ' ' << type;
    }
```

I lösningen defineras denna en gång i basklassen Player.

”Coupling”
I den ursprungliga koden finns mycket starka kopplingar. Bl.a. är klassen Team beroende av samtliga klasser som representerar spelare. I avsaknad av ett gemensamt interface kommer ändring i *något* av dessa klassers interface av tvinga fram ändringar i Team. I lösningen är Team beroende endast av interfacet i basklassen Player.

”Cohesion”
I den ursprungliga koden ansvarar de enskilda spelarklasserna själva för skriva ut speciella egenskaper för spelarna. I lösningen görs detta i egna ’propertyklasser’. Spelarklasserna behöver då inte känna till detaljer om egenskaperna eller hur de skrivs ut. På detta sätt har vi fått en högre grad av ’cohesion’, ett ökat mått av inre sammanhang.

```
#include<iostream>
#include<string>
#include<vector>

using namespace std;

class PlayerProperty {
public:
  PlayerProperty () {}
  virtual void display (ostream&) const=0 ;
};

class FootProperty : public PlayerProperty {
public:
  enum EFooted { LEFT, RIGHT, BOTH };
  FootProperty(EFooted f)
  :foot(f)
  { }
```

```
  virtual void display (ostream &os) const  {
    string s;
    switch(foot) {
        case LEFT: s = " left-footed"; break;
        case RIGHT: s = " right-footed"; break;
        case BOTH: s = " both-footed"; break;
    }
    os << s;
  }
private:
  EFooted foot;
};

class ReactProperty : public PlayerProperty {
public:
  enum EReaction { SLOW, FAST, LIGHTNING };
  ReactProperty(EReaction r)
  :rea(r)
  { }

  virtual void display (ostream &os) const {
    string s;
    switch(rea) {
        case SLOW: s = " slow"; break;
        case FAST: s = " fast"; break;
        case LIGHTNING: s = " lightning-like"; break;
    }
    os << s;
  }
private:
  EReaction rea;
};

class Player {
private:
    string name;
    string type;
    int number;
public:
    Player(string n, string t, int num)
    :name(n), type(t), number(num)
    { }
    virtual ~Player() { }

    int getNumber() const { return number; }

    virtual void display(ostream &os) {
       os << '\n' << number << ' ' << name << ' ' << type;
    }
```

```
};

class Defender : public Player {
private:
   PlayerProperty *prop;
public:

   Defender(string name, int num, FootProperty *p=0)
   :Player(name, "Defender", num), prop(p)
   { }

   virtual ~Defender() {
      delete prop;
   }

   void display(ostream &os=cout) {
     Player::display(os);
     if(prop)
       prop->display(os);
   }
}; // Defender


class Forward : public Player {
private:
   PlayerProperty *prop;
public:

   Forward(string name, int num, FootProperty *p=0)
   :Player(name, "Forward", num), prop(p)
   { }

   virtual ~Forward() {
      delete prop;
   }

   void display(ostream &os=cout) {
     Player::display(os);
     if(prop)
       prop->display(os);
   }
}; // Forward



class GoalKeeper : public Player {
private:
   PlayerProperty *prop;
public:
```

```
    GoalKeeper(string name, int num, ReactProperty *p=0)
    :Player(name, "GoalKeeper", num), prop(p)
    { }

    virtual ~GoalKeeper() {
       delete prop;
    }

    void display(ostream &os=cout) {
      Player::display(os);
      if(prop)
        prop->display(os);
    }
};  // GoalKeeper



class Team {
private:
   vector<Player*> players;
   string name;
   string coach;

   typedef  vector<Player*>::iterator iterator;

public:

   Team(string n, string c): name(n), coach(c) {}

   void addPlayer(Player *p) {
     players.push_back(p);
   }


   void display(ostream &os=cout) {
            os << name << ":\n";
        iterator first=players.begin(), last=players.end(),it;
        for (it=first; it!=last; it++ )
            (*it)->display(os);

        os << "\n\ncoached by " << coach << '\n' << endl;
   }
};

int main() {

   Team t1("Dream Team", "Dagge 'Demonen' Danielsson");

   Player *p =
   new GoalKeeper("Lennart 'Limmet' Larsson", 1,
```

```
        new ReactProperty(ReactProperty::LIGHTNING));
      t1.addPlayer(p);
      p = new Forward("Kalle 'Klippan' Karlsson",4,
        new FootProperty(FootProperty::BOTH));
      t1.addPlayer(p);
      p = new Forward("Sven 'Stinget' Svensson",8,
        new FootProperty(FootProperty::RIGHT));
      t1.addPlayer(p);
      p= new Defender("Roffe 'Ruffis' Ragnarsson",12,
        new FootProperty(FootProperty::LEFT));
      t1.addPlayer(p);


      t1.display();
      // delete .....

      cin.get();
      return 0;
   }
```

2. I följande program finns (minst) två designmönster dolda. Vilka? Motivera ditt svar!

```
#include <iostream>
#include <vector>
using namespace std;

class Stuff {
public:
  virtual void info()=0;
  virtual void use()=0;
};

class StandardStuff : public Stuff {
public:
  void info() { cout << " with a laser boosted water
gun..." << endl; }
  void use() { cout << "Splash splosh..." << endl; }
};

class OtherStuff {
public:
  void display() { cout << "  with a new shining
Light Saber..." << endl; }
  void utilize() { cout << "Swish swosh bzzz..." <<
endl; }
};
```

```
class  HeavyStuff : public Stuff, private OtherStuff
{
  public:
  void info() { OtherStuff::display();  }
  void use() { OtherStuff::utilize(); }
};

class ClassicStuff : public Stuff {
public:
  void info() { cout << " with an old-fashioned
wooden club..." << endl; }
  void use() { cout << "Tjonk tjonk..." << endl; }
};

/*-----------------------------------------------*/

class PlayGround {
public:
  virtual void info()=0;
};

class Space : public PlayGround {
public:
  void info() {
  cout << " inside a wreck of an old space
battleship";
  }
};

class Earth : public PlayGround {
public:
  void info() {
  cout << " in a great middle earth magma dome";
  }
};

/*-----------------------------------------------*/

class Performer {
public:
  virtual void fight() = 0;
  virtual void talk() = 0;
  virtual void info() = 0;
protected:
  Stuff *theStuff;
};

class JediKnight : public Performer {
public:
  JediKnight() {
     theStuff = new HeavyStuff;
```

```
  }

  void fight() {
    cout << "Yoda is really outperforming!!" <<
endl;
    theStuff->use();
  }

  void  talk() {
    cout << "Yoda: Not easy this fight is..." <<
endl;
  }

  void info() {
     cout << "- Yoda, the JediKnight master";
     theStuff->info();
  }

};

class Droid : public Performer {
public:
  Droid() {
    theStuff = new StandardStuff;
  }

  virtual void fight() {
    cout << "A droid will fight until the bitter end
..." << endl;
    theStuff->use();
  }

  void  talk() {
    cout << "Droid: I am programmed to shoot
everything in sight!" << endl;
  }

  void info() {
    cout << "- A standard fighting droid";
    theStuff->info();
  }
};

class CaveMan : public Performer {
public:
  CaveMan() {
    theStuff = new ClassicStuff;
  }

  virtual void fight() {
```

```
    cout << "Never underesitmate a caveman..." <<
endl;
    theStuff->use();
  }

  void  talk() {
    cout << "Caveman: Uggh ugggh!" << endl;
  }

  void info() {
     cout << "- A classic caveman";
     theStuff->info();
  }
};

/*-------------------------------------------------*/

class Fight {
public:
  vector<Performer*> pv;
  PlayGround *pg;

  Fight(PlayGround *p)
  : pg(p) {}

  void populate(vector<Performer*>&p) {
    pv=p;
  }

  void info() {
    cout << "We are"; pg->info(); cout << " together
with " << endl;
    vector<Performer*>::iterator first=pv.begin(),
last=pv.end(),it;
    for(it=first;it!=last;++it)
      (*it)->info();
    }

  void getComments() {
    vector<Performer*>::iterator first=pv.begin(),
last=pv.end(),it;
      for(it=first;it!=last;++it)
        (*it)->talk();
  }

  void start() {
    vector<Performer*>::iterator first=pv.begin(),
last=pv.end(),it;
      for(it=first;it!=last;++it)
        (*it)->fight();
  }
```

```
};

/*-----------------------------------------------*/

class Cosy {
protected:
  Fight *theFight;

public:
  virtual void getHuligans(vector<Performer*>& v) {
    v.push_back(new CaveMan);
    v.push_back(new Droid);
  }

  virtual PlayGround* grabAsetting() {
    return new Earth();
  }

  virtual void releaseThePowers() {
    theFight->info();
    theFight->start();
    theFight->getComments();
  }

  void fixAfight() {
    PlayGround *pg = grabAsetting();
    theFight = new Fight(pg);
    vector<Performer*> warriors;
    getHuligans(warriors);
    theFight->populate(warriors);
  }
};

class Nice: public Cosy {
public:
  virtual void getHuligans(vector<Performer*>& v) {
    v.push_back(new JediKnight);
    v.push_back(new Droid);
  }

  virtual PlayGround* grabAsetting() {
    return new Space();
  }

  virtual void releaseThePowers() {
    theFight->info();
    theFight->getComments();
    theFight->start();

  }
};
```

```
int main() {
   Cosy *cosy = new Cosy();
   cosy->fixAfight();
   cosy->releaseThePowers();
   cout << "----------------------------" << endl;
   cosy = new Nice();
   cosy->fixAfight();
   cosy->releaseThePowers();
   cin.get();
   return 0;
}
```
(4 p)

Lösning

1. **KlassAdapter**

```
class OtherStuff {
...
};

class  HeavyStuff : public Stuff, private OtherStuff
{ ...
};
```

`HeavyStuff` är en klassadapter som anpassar interfacet från `OtherStuff` till interfacet i `Stuff.`

2. **Factory Method**

I klassen Cosy behöver vissa objekt allokeras i `fixAfight().` Exakt vilka objekt som ska skapas i subklasser till `Cosy` överåts dock till respektive subklass genom att `fixAfight` utnyttjar factory-metoder för att skapa dessa objekt: `getHuligans och grabAsetting.` Genom att omdefiniera dessa funktioner kan alltså subklassen `Nice` ändra förutsättningarna enligt önskemål.

Ett tänkbart svar i detta sammanhang är även **Template Method** där `fixAfight()` definierar stegen som ska genomlöpas och där deriverade klasser kan omdefiniera vissa steg. Men eftersom exemplet handlar om allokering, att överlåta till subklasser att avgöra vilka objekt som ska skapas, så är detta svar inte lika bra och ger endast en poäng.

3. Ett **Strategy**-pattern finns gömt i klassen `Fight.` Beteendet för `Fight::info` betäms delvis av vilket `Playground`-objekt som Fight-objektet konfigureras med via konstruktorn.

Felaktiga svar:

- **Strategy** i sammanhanget `Performer` med pekaren `theStuff` och subklasser. `Performer` är en abstrakt basklass som saknar konstruktor för att konfigurera `theStuff` med en viss implementation av `Stuff`. De olika 'strategierna' är låsta i implementationen av de konkreta subklasserna `JediKnight`, `Droid` och `CaveMan` och man har ingen möjlighet att välja beteende hos, eller konfigurera dessa. Med andra ord: möjlighet till explicit konfigurering saknas.
- **Bridge** i samma sammanhang. Om det vore en Bridge skulle `Performer` implementera sina funktioner med hjälp av det objekt som `theStuff` pekar på och subklasserna skulle använda `Performer's` implementationer och inte använda `theStuff` direkt. Men, det viktiga är att i koden saknas explicit möjlighet att konfigurera `Performer` eller någon dess subklasser med ett `Stuff`-objekt! Vi är låsta till de implementationer av `Stuff` som allokeras i respektive subklass och då har hela Bridge-konceptet fallit och vi har fortfarande kvar ett potentiellt problem med 'kombinatorisk explosion'.

3. Vilket DP?

a. I ett givet sammanhang behöver en klass kunna varieras när det gäller ett visst beteende eller en viss algoritm.. Vilket DP passar bra? *Lösning:* Strategy (eller möjligtvis Template method)

b. Du ska implementera ett transaktionsliknande beteende där flera olika saker ska utföras för att en operation i sin helhet ska vara lyckad. Om något steg misslyckas ska status kunna återställas till startläget. Vilket DP? *Lösning:* Command

c. Vilket designmönster tillåter ett objekt att agera som mellanhand för ett annat objekt som kanske t.o.m. finns på en annan maskin? *Lösning:* Proxy

d. Antag att du har ett antal grundläggande klasser som implementerar ett interface från samma basklass. Ett antal sådanda objekt samlas i en sammansatt klass. Vilket DP ger möjligheten att addera även en sådan sammansättning till en annan sammansättning? Dvs en sammansättning ska kunna bestå både av av grundläggande objekt och andra sammansättningar i flera rekursiva led. *Lösning:* Composite (eller möjligtvis

Decorator)

(4 p)

4. Förklara begreppen *typ*, *klass* och *interface* och hur de är relaterade till varandra.

Lösning:

Ett objekts *klass* definierar hur objektet är implementerat. Klassen definierar objektets interna attribut/tillstånd och implementationen av dess funktioner.

Ett objekts *typ* är däremot endast relaterat till dess *interface* - dvs uppsättningen av meddelanden som det kan reagera på. *Ett objekt kan alltså ha flera typer och objekt av olika klasser kan ha samma typ.* Det råder naturligvis ett nära förhållande mellan objekts klass och dess typ. Eftersom klassen definierar vilka operationer ett objekt kan utföra finns så definerar den också objektet typ (eller typer). När vi säger att ett objekt är en instans av en viss klass, så menar vi att objektet stödjer det interface som definierats av klassen.

(se kapitlet Introduction, sid 16, i ”Design Patterns”)

(4 p)

5. Avgör om nedanstående 10 påståenden är korrekta eller ej. Varje rätt svar ger +1 p, varje felaktigt svar ger -1 p och varje uteblivet svar ger 0 p (Minsta poäng på uppgiften är 0 p !).

Lösning:

a) En switch-sats indikerar ofta problem med 'Tight coupling'. **SANT**

b) Begreppet *inkapsling* kan innefatta alla typer av accessrestriktioner för en klient. **SANT**

c) Den gemensamma kärnan (’commonality') i ett koncept motsvaras på konceptuell nivå av en abstrakt klass. **SANT**

d) En Proxy kan användas för att dölja allokering/deallokering av minne. **SANT**

e) En klassadapter konstrueras i C++ genom multipelt arv från två icke-publika basklasser. **FALSKT**

f) Designmönstret Bridge är ett exempel på principen "Välj komposition före arv". **SANT**

g) En funktion med 'low cohesion' är ineffektivt kodad, dvs den behöver onödigt mycket processorkraft för att lösa den aktuella uppgiften. **FALSKT**

h) MVC-mönstret använder sig av DP Template Method. **FALSKT**

i) En motsvarighet till den funktionsorienterade programmeringens callback-funktioner kan lätt åstadkommas genom att tillämpa DP Strategy. **FALSKT**

j) Arv från en ’protected’ basklass kan vara ett bra alternativ till komposition. **FALSKT.** (Ett **privat** arv däremot ger ett implementationsarv.)