Inlämningsuppgift  
Mjukvarukomponenter i C++  
(HT 2013)

Adam Näslund

a13adana@student.his.se

0209

Institutionen för kommunikation och information

Högskolan i Skövde

Innehållsförteckning

[1 Introduktion 4](#_Toc404621547)

[2 Tekniker 5](#_Toc404621548)

[2.1 Designmönster 5](#_Toc404621549)

[2.2 Isolering 14](#_Toc404621550)

[2.3 Kontrakt 18](#_Toc404621551)

[2.4 Testning 25](#_Toc404621552)

[2.5 Beroenden 29](#_Toc404621553)

[3 Applikation 37](#_Toc404621554)

[3.1 Problem 37](#_Toc404621555)

[3.2 Design 37](#_Toc404621556)

[3.3 Kod 37](#_Toc404621557)

[3.4 Körexempel 37](#_Toc404621558)

[3.5 Analys 38](#_Toc404621559)

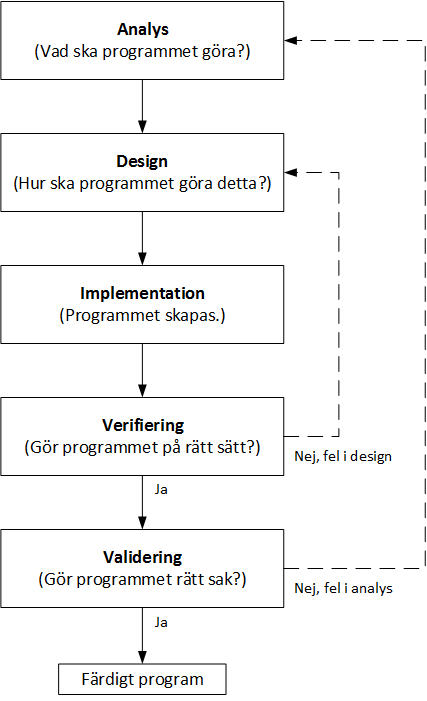
[4 Slutsats 39](#_Toc404621560)

**//TODO: Uppdatera**

# Introduktion

Denna rapport presenterar resultatet från de olika delmoment som utgjorde laborationsdelen av kursen Mjukvarukomponenter i C++ HT 13.

Delmomenten bestod av att hitta ett lämpligt exempelproblem, utforma en lösning på detta problem som använde relevanta tekniker, implementera lösningen i C++, skriva exempelkod för att visa att problemet hade lösts och slutligen analysera lösningens styrkor och svagheter. Detta förfarande speglar till viss del de fem stegen i programutvecklingsprocessen, vilka illustreras i figur 1 nedan.



Figur 1, programutvecklingsprocessen.

Analysdelen av delmomenten innehåller viss bedömning av lösningarnas kvalitet. Programkvalitet består av många olika delar som exempelvis utbyggbarhet, effektivitet, testbarhet, tillförlitlighet, användbarhet och förståelighet. Alla delar är dock inte lika relevanta för varje program och de kan i vissa fall stå mot varandra, varför varje programs kvalitet alltid måste utvärderas baserat på dess syfte och sammanhang.

Rapporten är strukturerad så att det för varje delmoment finns fem sektioner, Problem, Design, Kod, Körexempel och Analys; som respektive innehåller det exempelproblem som löstes, lösningens utformning, lösningens implementation i C++, resultatet från körning av exempelkoden samt en analys av lösningens kvalitet.

# Tekniker

## Designmönster

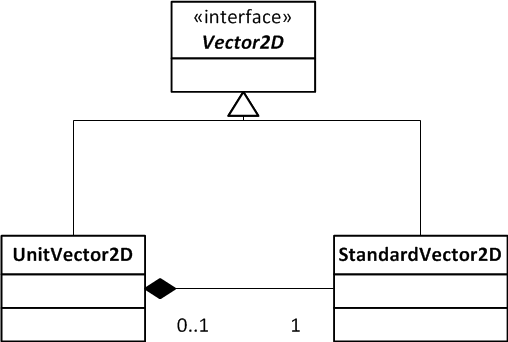
### Problem

Ett tvådimensionellt spel behöver representera riktning och position för entiteter i spelvärlden. Lösningen ska göra det enkelt att flytta entiteter i sin riktning och tillhandahålla grundläggande funktionalitet som kan vara användbar för spelet.

### Design

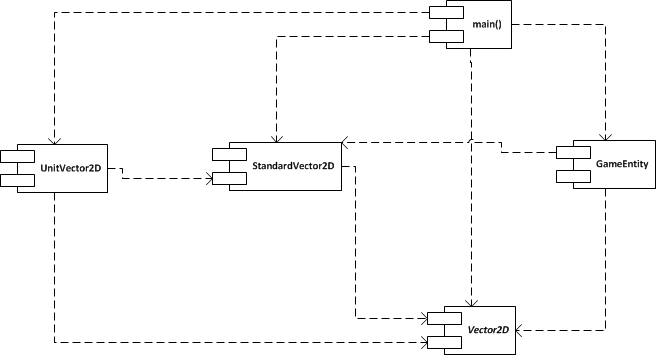
Jag valde att representera både riktning och position som vektorer på grund av att detta gör det mycket enkelt att använda dem tillsammans för förflyttning och för att detta enkelt möjliggör annan funktionalitet som kan vara användbar för ett spel, till exempel att ta reda på riktning eller avstånd från en entitet till en annan.

Då riktningens längd inte ska vara av betydelse valde jag att representera riktningen som en enhetsvektor. Enhetsvektorns klass UnitVector2D implementeras som en modifierad version av en vanlig vektor (StandardVector2D) med hjälp av designmönstret Proxy. Båda klasserna UnitVector2D och StandardVector2D ärver från den abstrakta basklassen Vector2D vilken ger det gemensamma gränssnitt genom vilket de kan användas, se figur 2 nedan.



Figur 2, klassdiagram för Vector2D samt subklasser.

I lösningen finns, förutom själva vektorklasserna, även exempelkod som visar hur klassen kan användas för att flytta två stycken spelentiteter av typen gameEntity och ändra riktning på den ena av dem så att den alltid är vänd bort från den första. Komponentdiagram som visar hur komponenterna använder varandra tillsammans med exempelkoden finns i figur 3.



Figur 3, komponentdiagram för Vector2D med subklasser och kod som ger exempel på användning.

### Kod

#### main.cpp

#include "StandardVector2D.h"

#include "UnitVector2D.h"

#include <iostream>

/\* This is a simple mockup of how a game entity could use the classes for movement \*/

struct GameEntity

{

/\* Holding pointers to these vectors doesn't really make a lot of sense in this

context, but we are more or less forced to as the base class is abstract. \*/

Vector2D \*position, \*direction;

float speed;

GameEntity(float speed, Vector2D \*position, Vector2D \*direction)

: speed(speed), position(position), direction(direction)

{

}

void move(float deltaTime)

{

/\* Here we can see the simple usage of the vector classes for movement. \*/

StandardVector2D step(\*direction);

step.multiply(speed \* deltaTime);

position->add(step);

}

~GameEntity()

{

delete position;

delete direction;

}

};

int main()

{

GameEntity entity1(2, new StandardVector2D(0, 0), new UnitVector2D(2, 1));

GameEntity entity2(1, new StandardVector2D(1, 1), new UnitVector2D());

float deltaTime = 0;

for(int i = 0; i < 10; ++i)

{

deltaTime = 1.0f;

// Delta time would normally be set to the time since last update.

std::cout<<"Entity1 Position: "<<entity1.position->getX()<<", "<<entity1.position->getY()

<<" \tDirection: "<<entity1.direction->getX()<<", "<<entity2.direction->getY()<<std::endl;

std::cout<<"Entity2 Position: "<<entity2.position->getX()<<", "<<entity2.position->getY()

<<" \tDirection: "<<entity2.direction->getX()<<", "<<entity2.direction->getY()<<std::endl;

std::cout<<std::endl;

entity1.move(deltaTime);

entity2.move(deltaTime);

/\* To show off the power and simplicity of the solution

let's have entity2 change its direction away from entity1

each iteration. Effectively fleeing from it. \*/

StandardVector2D v(\*entity2.position);

v.subtract(\*entity1.position);

entity2.direction->set(v.getX(), v.getY());

}

char a;

std::cin>>a; // Wait for input as a hack so the console stays open

return 0;

}

#### Vector2D.h

#ifndef VECTOR2D\_H

#define VECTOR2D\_H

class Vector2D

{

public:

virtual ~Vector2D() {};

virtual float getX() const = 0;

virtual float getY() const = 0;

virtual void set(float x, float y) = 0;

virtual void setX(float x) = 0;

virtual void setY(float y) = 0;

virtual void add(const Vector2D &vector) = 0;

virtual void subtract(const Vector2D &vector) = 0;

virtual void multiply(const float scalar) = 0;

virtual void normalize() = 0;

};

#endif // VECTOR2D\_H

#### StandardVector2D.h

#ifndef STANDARDVECTOR2D\_H

#define STANDARDVECTOR2D\_H

#include "Vector2D.h"

class StandardVector2D : public Vector2D

{

public:

StandardVector2D();

StandardVector2D(float x, float y);

explicit StandardVector2D(const Vector2D &vector);

virtual float getX() const override;

virtual float getY() const override;

virtual void set(float x, float y) override;

virtual void setX(float x) override;

virtual void setY(float y) override;

virtual void add(const Vector2D &vector) override;

virtual void subtract(const Vector2D &vector) override;

virtual void multiply(const float scalar) override;

virtual void normalize() override;

private:

float x, y;

};

#endif // STANDARDVECTOR2D\_H

#### StandardVector2D.cpp

#include "StandardVector2D.h"

#include <math.h>

StandardVector2D::StandardVector2D(float x, float y)

: x(x), y(y)

{

}

StandardVector2D::StandardVector2D()

: x(0), y(0)

{

}

StandardVector2D::StandardVector2D(const Vector2D &vector)

: x(vector.getX()), y(vector.getY())

{

}

float StandardVector2D::getX() const

{

return x;

}

float StandardVector2D::getY() const

{

return y;

}

void StandardVector2D::set(float x, float y)

{

this->x = x;

this->y = y;

}

void StandardVector2D::setX(float x)

{

this->x = x;

}

void StandardVector2D::setY(float y)

{

this->y = y;

}

void StandardVector2D::add(const Vector2D &vector)

{

x+=vector.getX();

y+=vector.getY();

}

void StandardVector2D::subtract(const Vector2D &vector)

{

x-=vector.getX();

y-=vector.getY();

}

void StandardVector2D::multiply(float scalar)

{

x\*=scalar;

y\*=scalar;

}

void StandardVector2D::normalize()

{

if(!(x == 0 && y == 0))

{ // We can't normalise the zero vector

float length = sqrt((x \* x) + (y \* y));

x/=length;

y/=length;

}

}

#### UnitVector2D.h

#ifndef UNITVECTOR2D\_H

#define UNITVECTOR2D\_H

#include "Vector2D.h"

#include "StandardVector2D.h"

class UnitVector2D : public Vector2D

{

public:

UnitVector2D();

UnitVector2D(float x, float y);

explicit UnitVector2D(const Vector2D &vector);

virtual float getX() const override;

virtual float getY() const override;

virtual void set(float x, float y) override;

virtual void setX(float x) override;

virtual void setY(float y) override;

virtual void add(const Vector2D &vector) override;

virtual void subtract(const Vector2D &vector) override;

virtual void multiply(const float scalar) override;

virtual void normalize() override;

private:

StandardVector2D vector;

};

#endif // UNITVECTOR2D\_H

#### UnitVector2D.cpp

#include "UnitVector2D.h"

#include <math.h>

UnitVector2D::UnitVector2D(float x, float y)

: vector(StandardVector2D(x, y))

{

vector.normalize();

}

UnitVector2D::UnitVector2D()

: vector(StandardVector2D())

{

// We allow 0,0 unit vectors even though they're not really unit vectors

// as they are useful to signify "no direction"

}

UnitVector2D::UnitVector2D(const Vector2D &vector)

: vector(vector.getX(), vector.getY())

{

this->vector.normalize();

}

float UnitVector2D::getX() const

{

return vector.getX();

}

float UnitVector2D::getY() const

{

return vector.getY();

}

void UnitVector2D::set(float x, float y)

{

vector.set(x,y);

vector.normalize();

}

void UnitVector2D::setX(float x)

{

vector.setX(x);

vector.normalize();

}

void UnitVector2D::setY(float y)

{

vector.setY(y);

vector.normalize();

}

void UnitVector2D::add(const Vector2D &vector)

{

StandardVector2D v(vector.getX(), vector.getY());

v.normalize();

v.add(this->vector);

v.normalize();

this->vector = v;

}

void UnitVector2D::subtract(const Vector2D &vector)

{

StandardVector2D v(vector.getX(), vector.getY());

v.normalize();

this->vector.subtract(v);

this->vector.normalize();

}

void UnitVector2D::multiply(const float scalar)

{

// There are only two cases in which scalar multiplication of a unit vector

// changes it and they are covered below.

if(scalar < 0)

{

vector.multiply(-1);

}

else if(scalar == 0)

{

vector.set(0, 0);

}

}

void UnitVector2D::normalize()

{

// Do nothing

}

### Körexempel

Nedan visas utdata från exempelkoden. Här visas position och riktning de två spelobjekten Entity1 och Entity2 i tio steg av ”spelet”. Entity2 ändrar i slutet av varje steg sin riktning så den pekar bort från Entity1 medan Entity1 behåller samma riktning.

Entity1 Position: 0, 0 Direction: 0.894427, 0

Entity2 Position: 1, 1 Direction: 0, 0

Entity1 Position: 1.78885, 0.894427 Direction: 0.894427, 0.132648

Entity2 Position: 1, 1 Direction: -0.991163, 0.132648

Entity1 Position: 3.57771, 1.78885 Direction: 0.894427, -0.180838

Entity2 Position: 0.00883675, 1.13265 Direction: -0.983513, -0.180838

Entity1 Position: 5.36656, 2.68328 Direction: 0.894427, -0.263407

Entity2 Position: -0.974676, 0.95181 Direction: -0.964685, -0.263407

Entity1 Position: 7.15542, 3.57771 Direction: 0.894427, -0.302777

Entity2 Position: -1.93936, 0.688403 Direction: -0.953062, -0.302777

Entity1 Position: 8.94427, 4.47214 Direction: 0.894427, -0.32634

Entity2 Position: -2.89242, 0.385627 Direction: -0.945253, -0.32634

Entity1 Position: 10.7331, 5.36656 Direction: 0.894427, -0.342244

Entity2 Position: -3.83768, 0.0592872 Direction: -0.939611, -0.342244

Entity1 Position: 12.522, 6.26099 Direction: 0.894427, -0.353811

Entity2 Position: -4.77729, -0.282957 Direction: -0.935317, -0.353811

Entity1 Position: 14.3108, 7.15542 Direction: 0.894427, -0.36266

Entity2 Position: -5.7126, -0.636768 Direction: -0.931921, -0.36266

Entity1 Position: 16.0997, 8.04984 Direction: 0.894427, -0.369685

Entity2 Position: -6.64453, -0.999428 Direction: -0.929157, -0.369685

Exemplet visar att lösningen kan användas för att representera både position och riktning för entiteter, samt att dessa kan användas tillsammans för att flytta entiteterna.

### Analys

En stark fördel med lösningen är att det är mycket enkelt att använda position och riktning tillsammans då de båda representeras som vektorer. Lösningen gör det även enkelt att utföra diverse position- och riktningsrelaterade operationer, som till exempel att ta reda på riktningen från en entitet till en annan (vilket görs i exempelkoden i 2.1.3 main.cpp).

Lösningen är även enkelt utbyggbar med nya vektorfunktioner som kan vara användbara. Till exempel skulle funktionalitet för att ge vektorlängd kunna läggas till och användas för att ta reda på avståndet mellan entiteter.

Användandet av en abstrakt basklass i lösningen kan dock ifrågasättas, då nyttan av att kunna hantera vektorer av olika typ via samma gränssnitt är tvivelaktig och måste vägas mot den ökade komplexitet, risk för minnesläckage och risk för logiska fel som uppstår vid användandet av pekare eller referenser.

Den representation av riktning som enhetsvektor som används medför också en något hög beräkningskomplexitet då relativt tunga operationer som att dra roten ur tal och att instansiera objekt utförs vid användning av nästan varje funktion klassen.

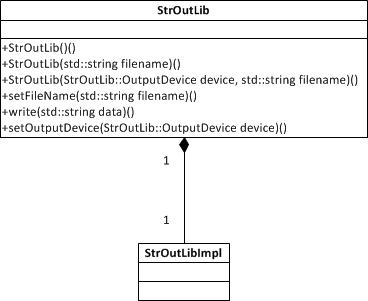
## Isolering

### Problem

Flera olika program behöver kunna skriva text till både filer och till konsol (standard out) på ett enkelt sätt.

### Design

Jag valde att lösa problemet genom att skapa ett programbibliotek för textutskrift. För att kunna ändra implementationen utan att de program som använder biblioteket behöver kompileras om är implementationen isolerad med hjälp av tekniken pimpl. Figur 4 nedan visar klassdiagram för lösningen samt det gränssnitt som exponeras av StrOutLib.



Figur 4, klassdiagram för lösningen med gränssnitt som exponeras av biblioteket.

I main.cpp i lösningen finns exempelkod som visar hur klassen används, figur 5 visar hur de olika delarna i exemplet använder varandra.

\\hsnas3\StudentHome\2013\a13adana\Documents\mjukkomp_da327g\1_2_StrOutLib\komponentdiagram.png

Figur 5, komponentdiagram för StrOutLib samt exempelkod i main.

### Kod

#### main.cpp

#include <string>

#include <iostream>

#include "StrOutLib.h"

int main()

{

StrOutLib strOut;

strOut.write("Console is the default output device");

// For file output a filename must be set and the output device must be set to FILE

strOut.setFileName("test1.txt");

strOut.setOutputDevice(StrOutLib::FILE);

strOut.write("Writing to a file.");

// We can also use the convenient constructor that takes a filename to set it to file output

StrOutLib fileOut("test2.txt");

fileOut.write("Writing to another file.");

std::cin.get();

return 0;

}

#### StrOutLib.h

#ifndef STROUTLIB\_H

#define STROUTLIB\_H

#include <string>

class StrOutLib

{

public:

enum OutputDevice { CONSOLE, FILE };

StrOutLib();

explicit StrOutLib(StrOutLib::OutputDevice device, std::string filename = "");

explicit StrOutLib(std::string filename);

~StrOutLib();

void setFileName(std::string filename);

void write(std::string data);

void setOutputDevice(StrOutLib::OutputDevice device);

private:

StrOutLib(const StrOutLib&) {}

// Copying this class does not really make sense

StrOutLib& operator=(StrOutLib) {}

struct StrOutLibImpl;

StrOutLibImpl \*mPImpl;

};

#endif // STROUTLIB\_H

#### StrOutLib.cpp

#include "StrOutLib.h"

#include <string>

#include <iostream>

#include <fstream>

#include <assert.h>

struct StrOutLib::StrOutLibImpl

{

std::string mBuffer;

std::string mFilename;

StrOutLib::OutputDevice mOutDev;

std::ofstream mFstream;

StrOutLibImpl(StrOutLib::OutputDevice outputDevice = StrOutLib::CONSOLE, std::string filename = "")

: mBuffer(), mFilename(filename), mOutDev(outputDevice), mFstream()

{

}

void write()

{

if(mOutDev == StrOutLib::CONSOLE)

{

std::cout << mBuffer;

mBuffer.clear();

}

else if(mOutDev == StrOutLib::FILE)

{

assert(!mFilename.empty());

// In lieu of throwing an exception we just don't write anything if a filename is missing

if(!mFilename.empty())

{

mFstream.open(mFilename);

mFstream << mBuffer;

mFstream.close();

}

}

}

};

StrOutLib::StrOutLib()

: mPImpl(new StrOutLibImpl())

{

}

StrOutLib::StrOutLib(StrOutLib::OutputDevice device, std::string filename)

: mPImpl(new StrOutLibImpl(device, filename))

{

}

StrOutLib::StrOutLib(std::string filename)

: mPImpl(new StrOutLibImpl(StrOutLib::FILE, filename))

{

}

StrOutLib::~StrOutLib()

{

delete mPImpl;

mPImpl = 0;

}

void StrOutLib::setFileName(const std::string filename)

{

mPImpl->mFilename = filename;

}

void StrOutLib::write(const std::string data)

{

mPImpl->mBuffer += data;

mPImpl->write();

}

void StrOutLib::setOutputDevice(StrOutLib::OutputDevice device)

{

mPImpl->mOutDev = device;

}

### Körexempel

När exempelkoden körs skrivs följande ut till konsolfönstret (om det är standard out):

Console is the default output device

Följande skrivs till filen test1.txt:

Writing to a file.

Följande skrivs till filen test2.txt:

Writing to another file.

Vilket visar att biblioteket går att använda både för utskrift till konsol och till olika filer.

### Analys

Biblioteket som utgör lösningen kan enkelt användas av olika program för att skriva text både till filer och till konsol. Isoleringen av implementationen gör att så länge gränssnittet förblir detsamma, kan biblioteket ändras utan att kod som beror på det behöver kompileras om, något som är mycket fördelaktigt för ett kodbibliotek.

En möjlig nackdel med lösningen att prestandan för filskrivning kan vara dålig då filströmmen öppnas och strängs för varje skrivning. Om detta är ett problem går det dock, på grund av den isolerade implementationen, att optimera biblioteket utan att bryta binärkompatibilitet med tidigare versioner.

## Kontrakt

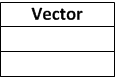
### Problem

Ett program behöver kunna representera och hantera vektorer av n längd. Vektorerna skall kunna adderas, subtraheras och jämnföras med varandra, samt multipliceras med en skalär. Det ska även gå att ta skalärprodukten av två vektorer.

### Design

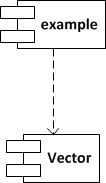
Jag valde att lösa problemet med en vektorklass Vector där de olika operationerna implementerats som operatorer på klassen. För att tillgodose att klassen fungerar och används som den ska använde jag i varje funktion assert-satser för att kontrollera previllkor, postvillkor samt invariant.

Vektorklassen beror enbart på ostream och cassert, den förstnämnda för stöd av << operatorn så att en vektor enkelt kan skrivas ut, och den senare för kontroll av kontrakt under debugging. Att klassen har minimala beroenden syns i klassdiagrammet i figur 6.



Figur 6, klassdiagram för Vector.

I example.cpp finns exempelkod som visar hur klassen kan användas. Klassdiagrammet i figur 7 visar hur exempelkoden använder Vector.



Figur 7, komponentdiagram för klassen Vector med exempelkod.

### Kod

#### example.cpp

#include "Vector.h"

#include <iostream>

int main()

{

double a[5] = {1,2,3,4,5};

double b[5] = {1,0,1,0,1};

double c[5] = {1,2,1,2,1};

Vector u(5, a);

Vector v(5, b);

Vector w(5, c);

std::cout << "u: " << u << "\nv: " << v << "\nw: " << w << "\n";

std::cout << "\n";

std::cout << "u + v = " << u + v << "\n";

std::cout << "\n";

std::cout << "u \* v = "<< u \* v << "\n";

std::cout << "\n";

std::cout << "u - 2\*v = "<< u - 2\*v << "\n";

std::cout << "\n";

u \*= 2;

std::cout << "u \*= 2" << "\nu: " << u <<"\n";

std::cout << "\n";

v += w;

std::cout << "v += w" << "\nv: " << v <<"\n";

std::cout << "\n";

w -= v;

std::cout << "w -= v" << "\nw: " << w <<"\n";

std::cout << "\n";

std::cin.get();

return 0;

}

#### Vector.h

#ifndef VECTOR\_H

#define VECTOR\_H

#include <iosfwd>

class Vector

{

public:

Vector(int length, const double values[]);

Vector(const Vector&);

~Vector();

Vector& operator=(const Vector &rhs);

Vector& operator+=(const Vector &rhs);

Vector& operator-=(const Vector &rhs);

Vector& operator\*=(const double &scalar);

const double & operator[](int index) const; // For getting value

double & operator[](int index); // For setting value

int length() const;

private:

bool invariant() const;

double \*mValues; // Pointer to array of mLength elements

const int mLength;

};

const Vector operator+(const Vector &lhs, const Vector &rhs);

const Vector operator-(const Vector &lhs, const Vector &rhs);

const double operator\*(const Vector &lhs, const Vector &rhs); // Dot product

const Vector operator\*(const Vector &vector, double scalar);

const Vector operator\*(double scalar, const Vector &vector);

bool operator==(const Vector &lhs, const Vector &rhs);

bool operator!=(const Vector &lhs, const Vector &rhs);

std::ostream& operator<<(std::ostream &stream, const Vector &vector); // Prints values to stream

#endif VECTOR\_H

#### Vector.cpp

#include "Vector.h"

#include <cassert>

#include <ostream>

namespace

{

// Helper function to copy all values of one Vector to another

void copyValues(const Vector & source, Vector & target)

{

assert(source.length() == target.length());

for (int i = 0; i < source.length(); ++i)

{

target[i] = source[i];

}

assert(source[0] == target[0]);

assert(source[target.length()-1] == target[source.length()-1]);

}

}

Vector::Vector(const int length, const double values[])

: mLength(length), mValues(new double[length])

{

assert(length > 0);

for(int i = 0; i < mLength; ++i)

{

mValues[i] = values[i];

}

assert(mValues[0] == values[0]);

assert(mValues[mLength-1] == values[length-1]);

assert(invariant());

}

Vector::Vector(const Vector& vector)

: mLength(vector.mLength), mValues(new double[vector.mLength])

{

assert(mLength > 0);

copyValues(vector, \*this);

assert(mValues[0] == vector.mValues[0]);

assert(mValues[vector.mLength-1] == vector.mValues[mLength-1]);

assert(invariant());

}

Vector::~Vector()

{

assert(invariant());

delete mValues;

mValues = 0;

}

Vector& Vector::operator=(const Vector &rhs)

{

assert(invariant());

assert(rhs.length() == mLength);

copyValues(rhs, \*this);

assert(mValues[0] == rhs[0]);

assert(mValues[rhs.length()-1] == rhs[mLength-1]);

assert(invariant());

return \*this;

}

Vector& Vector::operator+=(const Vector &rhs)

{

assert(invariant());

assert(rhs.length() == mLength);

#ifndef NDEBUG

double oldFirst = mValues[0];

double oldLast = mValues[mLength-1];

#endif

for (int i = 0; i < mLength; ++i)

{

mValues[i] += rhs.mValues[i];

}

assert(mValues[0] == oldFirst + rhs.mValues[0]);

assert(mValues[rhs.length()-1] == oldLast + rhs.mValues[mLength-1]);

assert(invariant());

return \*this;

}

Vector& Vector::operator-=(const Vector &rhs)

{

assert(invariant());

assert(rhs.length() == mLength);

#ifndef NDEBUG

double oldFirst = mValues[0];

double oldLast = mValues[mLength-1];

#endif

for (int i = 0; i < mLength; ++i)

{

mValues[i] -= rhs.mValues[i];

}

assert(mValues[0] == oldFirst - rhs.mValues[0]);

assert(mValues[rhs.length()-1] == oldLast - rhs.mValues[mLength-1]);

assert(invariant());

return \*this;

}

Vector& Vector::operator\*=(const double &scalar)

{

assert(invariant());

#ifndef NDEBUG

double oldFirst = mValues[0];

double oldLast = mValues[mLength-1];

#endif

for (int i = 0; i < mLength; ++i)

{

mValues[i] \*= scalar;

}

assert(mValues[0] == oldFirst \* scalar);

assert(mValues[mLength-1] == oldLast \* scalar);

assert(invariant());

return \*this;

}

// For getting value

const double & Vector::operator[](int index) const

{

assert(invariant());

assert(index < mLength);

assert(index >= 0);

return mValues[index];

}

// For setting value

double & Vector::operator[](int index)

{

assert(invariant());

assert(index < mLength);

assert(index >= 0);

return mValues[index];

}

int Vector::length() const

{

assert(invariant());

assert(mLength > 0);

return mLength;

}

bool Vector::invariant() const

{

// This is not a very useful invariant, but it's all we can really check.

return mLength > 0 && mValues != 0;

}

const Vector operator+(const Vector &lhs, const Vector &rhs)

{

assert(lhs.length() > 0 && lhs.length() == rhs.length());

Vector result(lhs);

result += rhs;

return result;

}

const Vector operator-(const Vector &lhs, const Vector &rhs)

{

assert(lhs.length() > 0 && lhs.length() == rhs.length());

Vector result(lhs);

result -= rhs;

return result;

}

const double operator\*(const Vector &lhs, const Vector &rhs)

{

assert(lhs.length() != 0 && lhs.length() == rhs.length());

double result = 0;

for(int i = 0; i < lhs.length(); ++i)

{

result += lhs[i]\*rhs[i];

}

return result;

}

const Vector operator\*(const Vector &vector, const double scalar)

{

assert(vector.length() != 0);

Vector result(vector);

result \*= scalar;

return result;

}

const Vector operator\*(const double scalar, const Vector &vector)

{

return vector\*scalar;

}

bool operator==(const Vector &lhs, const Vector &rhs)

{

assert(lhs.length() > 0 && lhs.length() == rhs.length());

for(int i = 0; i < lhs.length(); ++i)

{

if(lhs[i] != rhs[i])

{

return false;

}

}

return true;

}

bool operator!=(const Vector &lhs, const Vector &rhs)

{

return !(lhs == rhs);

}

std::ostream& operator<<(std::ostream &stream, const Vector &vector)

{

assert(vector.length() != 0);

stream << vector[0];

for(int i = 1; i < vector.length(); ++i)

{

stream << ", " << vector[i];

}

return stream;

}

### Körexempel

Exempelkoden skapar tre vektorer, skriver ut dem och utför sedan flera olika operationer på dem och skriver ut resultatet för att visa att de går att addera, subtrahera, multiplicera med skalärer och ta skalärprodukten av.

u: 1, 2, 3, 4, 5

v: 1, 0, 1, 0, 1

w: 1, 2, 1, 2, 1

u + v = 2, 2, 4, 4, 6

u \* v = 9

u - 2\*v = -1, 2, 1, 4, 3

u \*= 2

u: 2, 4, 6, 8, 10

v += w

v: 2, 2, 2, 2, 2

w -= v

w: -1, 0, -1, 0, -1

### Analys

Användningen av operatoröverlagring för de olika operationerna gör vektorklassen väldigt enkel att använda och bidrar till att göra kod som använder klassen läsbar. Kontrollen av in- och utdata samt invariant med hjälp av asserts gör att de flesta sorters felaktig användning av klassen, samt många fel som kan uppstå i klassen, genererar väldefinierat beteende. Lagring av värden som typen double innebär att klassen har hög precision på sina flyttal vilket minskar risken för avrundningsfel, men även att klassen kräver mer minne än om den skulle lagra sina tal som en mindre precis typ.

En annan nackdel med lösningen är att konstruktorn, som tar en array med doubles samt en int som visar hur stor arrayen är, är mycket lätt att använda på ett felaktigt sätt. Om arrayen har färre element än det inskickade värdet säger blir beteendet odefinierat. En annan allvarlig nackdel är att kontroll av indata enbart görs via asserts, vilket innebär att produktionskod helt kommer sakna denna kontroll.

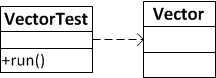
## Testning

### Problem

Vektorklassen i 2.3.3 behöver testas för att försäkra dess korrekta funktionalitet, både i nuläget och efter framtida ändringar.

### Design

Testningen implementeras med klassen VectorTest vilken med hjälp av assert-satser testar alla vektorklassens funktioner, samt dess funktion vid vissa gränsfall. Testklassen använder förutom standardbibliotek endast klassen Vector, vilket syns i klassdiagrammet i figur 8.



Figur 8, klassdiagram för VectorTest

Lösningen innehåller även en fil vid namn main som visar hur VectorTest används. Hur denna använder VectorTest kan ses i komponentdiagrammet i figur 9.

\\hsnas3\StudentHome\2013\a13adana\Documents\mjukkomp_da327g\1_4_VectorTest\komponentdiagram.png

Figur 9, komponentdiagram för VectorTest.

### Kod

#### main.cpp

#include "VectorTest.h"

int main()

{

VectorTest::run();

return 0;

}

#### VectorTest.h

#ifndef VECTORTEST\_H

#define VECTORTEST\_H

struct VectorTest

{

static void run();

};

#endif // VECTORTEST\_H

#### VectorTest.cpp

#include "VectorTest.h"

#include "Vector.h"

#include <cassert>

#include <sstream>

void VectorTest::run()

{

/\* Testing all functions \*/

{

double a[5] = {1,2,3,4,5};

double b[5] = {0,1,2,3,4};

const Vector v0(5, a);

const Vector v1(5, b);

Vector v2(v0);

// Constructor, length, and const []

assert(v0.length() == 5);

assert(v0[0] == 1);

assert(v0[4] == 5);

// Copy constructor

assert(v2.length() == 5);

assert(v2[0] == 1);

assert(v2[4] == 5);

// Comparison

assert(v0 != v1);

assert(v0 == v2);

// Assignment []

v2[0] = 42;

assert(v2[0] == 42);

v2[v2.length() - 1] = 13;

assert(v2[v2.length() - 1] == 13);

// Assignment and compount assignment

v2 = v0;

assert(v2 == v0);

{

v2 = v0;

v2 += v1;

double cTmp[5] = {1,3,5,7,9};

Vector vTmp(5, cTmp);

assert(v2 == vTmp);

}

{

v2 = v0;

v2 -= v1;

double cTmp[5] = {1,1,1,1,1};

Vector vTmp(5, cTmp);

assert(v2 == vTmp);

}

{

v2 = v0;

v2 \*= 2;

double cTmp[5] = {2,4,6,8,10};

Vector vTmp(5, cTmp);

assert(v2 == vTmp);

}

// Binary arithmetic

v2 = v0;

v2 += v1;

assert(v0 + v1 == v2);

v2 = v0;

v2 -= v1;

assert(v0 - v1 == v2);

assert(v0 \* v1 == 40); // Dot product

v2 = v0;

v2 \*= 2;

assert(v0 \* 2 == v2);

assert(2 \* v0 == v2);

// Stream

std::stringstream s;

s << v0;

assert(s.str() == "1, 2, 3, 4, 5");

}

/\* Edge cases \*/

{

{

// Single element large values

double a[1] = {123456789012.34567890};

double b[1] = {-123456789.01234567890};

const Vector v0(1, a);

const Vector v1(1, b);

assert(v0[0] == a[0]);

assert(v1[0] == b[0]);

assert((v0 + v1)[0] == a[0] + b[0]);

}

{

// Large number of elements

const int NUM\_ELEMS = 9000;

double a[NUM\_ELEMS] = {};

double b[NUM\_ELEMS] = {};

int sum1 = 0;

int sum2 = 0;

for(int i = 0; i < NUM\_ELEMS; ++i)

{

a[i] = i;

b[i] = i;

// sum1 will be the sum of all values in a and b together

sum1 += 2\*i;

}

Vector v0(NUM\_ELEMS, a);

const Vector v1(NUM\_ELEMS, b);

// Add the vectors

v0 += v1;

for(int i = 0; i < NUM\_ELEMS; ++i)

{

// Sum up the values of v0

sum2 += v0[i];

}

// And check that the sums match

assert(sum1 == sum2);

}

}

}

### Körexempel

Eftersom programmet använder assert-satser, innebär ett test som inte hittar några fel att körningen inte avbryts. Nedan ges exempel på hur det kan se ut när klasses testas och inga fel hittas:

The program '[3776] 1\_4\_VectorTest.exe' has exited with code 0 (0x0).

### Analys

Testet har utformats så att det testar varje funktion klassen har, detta betyder dock inte att all kod nödvändigtvis körs, även om en ytlig analys antyder det. Två extremfall med enkla operationer på vektorer som enbart har ett, väldigt stort element och vektorer som har väldigt många element testas även.

Att testerna implementerats som assert-satser innebär att testprogrammet avbryts om ett fel identifierats. Detta har fördelen att programmet då kan debuggas för att se vilka anrop som har gjorts samt vilka värden variablerna hade då felet upptäcktes, vilket kan hjälpa utvecklaren identifiera vad som gick fel. Det innebär dock också att om något enskilt test misslyckas så kommer inte de efterföljande testen att köras, samt att det kan vara komplicerat att automatisera testningen.

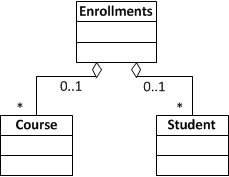
## Beroenden

### Problem

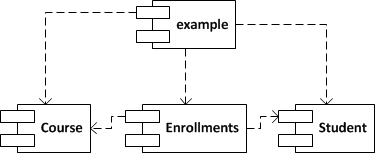
Relationen mellan studenter och kurser behöver representeras. Det skall gå att registrera och avregistrerade studenter på kurser. En student ska kunna vara registrerad på flera kurser och en kurs ska givetvis kunna ha flera studenter registrerade. Det ska även gå att på ett enkelt sätt ta fram alla studenter registrerade på en given kurs, samt att ta fram alla kurser på vilka en given student är registrerad.

### Design

Studenter representeras av klassen Student, kurser av klassen Course och relationen mellan dem representeras av klassen Enrollments. Då relationen hanteras av en separat klass finns inga beroenden mellan Student och Course, vilket syns både i klassdiagrammet i figur 10 och i komponentdiagrammet i figur 11.



Figur 10, klassdiagram för lösningen.



Figur 11, komponentdiagram för lösningen.

### Kod

#### example.cpp

#include "Student.h"

#include "Course.h"

#include "Enrollments.h"

#include <set>

#include <iostream>

void printEnrollments(const Enrollments &enrollments)

{

std::set<Course\*> courses = enrollments.getAllCourses();

for(Course\* course : courses)

{

std::cout << "----- " << course->getId() << ": " << course->getName() << " -----\n";

std::set<Student\*> students = enrollments.getStudents(course);

for(Student\* student : students)

{

std::cout << student->getName() << "\n";

}

std::cout << "\n";

}

}

int main()

{

Student \*adam = new Student("Adam Naslund", "Kurortsvagen 2");

Student \*erik = new Student("Erik Andersson", "Norra trangallén 6");

Student \*anders = new Student("Anders Eriksson", "Norra trangallén 6");

Student \*peter = new Student("Peter Andersson", "Kurortsvagen 4");

Course \*mjukkomp = new Course("DA327G", "Mjukvarukomponenter i C++");

Course \*linalg = new Course("MA142G", "Linjar algebra");

Enrollments ht2013;

ht2013.enroll(adam, mjukkomp);

ht2013.enroll(erik, mjukkomp);

ht2013.enroll(anders, mjukkomp);

ht2013.enroll(peter, mjukkomp);

ht2013.enroll(adam, linalg);

ht2013.enroll(erik, linalg);

ht2013.enroll(anders, linalg);

ht2013.enroll(peter, linalg);

printEnrollments(ht2013);

std::cout << "\n### Removing Anders from DA327G. ###\n";

ht2013.disenroll(anders, mjukkomp);

printEnrollments(ht2013);

std::cout << "\n### Removing Peter from all courses. ###\n";

ht2013.disenroll(peter);

printEnrollments(ht2013);

std::cout << "\n### Removing all students from MA142G. ###\n";

ht2013.disenroll(linalg);

printEnrollments(ht2013);

std::cin.get();

return 0;

}

#### Enrollments.h

#ifndef Enrollments\_H

#define Enrollments\_H

#include <set>

#include <map>

#include <vector>

class Student;

class Course;

class Enrollments

{

public:

void enroll(Student \*student, Course \*course);

void disenroll(Student \*student, Course \*course);

void disenroll(Student \*student);

void disenroll(Course \*course);

std::set<Student \*> getStudents(Course \*course) const;

std::set<Course \*> getCourses(Student \*student) const;

std::set<Student \*> getAllStudents() const;

std::set<Course \*> getAllCourses() const;

private:

std::map<Course \*, std::set<Student \*>> mCourseToStudents;

std::map<Student \*, std::set<Course \*>> mStudentToCourses;

};

#endif // Enrollments\_H

#### Enrollments.cpp

#include "Enrollments.h"

void Enrollments::enroll(Student \*student, Course \*course)

{

mCourseToStudents[course].insert(student);

mStudentToCourses[student].insert(course);

}

void Enrollments::disenroll(Student \*student, Course \*course)

{

mCourseToStudents[course].erase(student);

mStudentToCourses[student].erase(course);

}

void Enrollments::disenroll(Student \*student)

{

mStudentToCourses.erase(student);

for(auto i = mCourseToStudents.begin(); i != mCourseToStudents.end(); ++i)

{

i->second.erase(student);

}

}

void Enrollments::disenroll(Course \*course)

{

mCourseToStudents.erase(course);

for(auto i = mStudentToCourses.begin(); i != mStudentToCourses.end(); ++i)

{

i->second.erase(course);

}

}

std::set<Student \*> Enrollments::getStudents(Course \*course) const

{

auto students = mCourseToStudents.find(course);

if(students == mCourseToStudents.end())

{

std::set<Student \*> empty;

return empty;

}

else

{

return students->second;

}

}

std::set<Course \*> Enrollments::getCourses(Student \*student) const

{

auto courses = mStudentToCourses.find(student);

if(courses == mStudentToCourses.end())

{

std::set<Course \*> empty;

return empty;

}

else

{

return courses->second;

}

}

std::set<Student \*> Enrollments::getAllStudents() const

{

std::set<Student \*> students;

for(auto i = mStudentToCourses.begin(); i != mStudentToCourses.end(); ++i)

{

students.insert(i->first);

}

return students;

}

std::set<Course \*> Enrollments::getAllCourses() const

{

std::set<Course \*> courses;

for(auto i = mCourseToStudents.begin(); i != mCourseToStudents.end(); ++i)

{

courses.insert(i->first);

}

return courses;

}

#### Course.h

#ifndef COURSE\_H

#define COURSE\_H

#include <string>

class Course

{

public:

Course(std::string id, std::string name);

std::string getId() const;

void setId(std::string id);

std::string getName() const;

void setName(std::string name);

private:

std::string mId;

std::string mName;

};

#endif // COURSE\_H

#### Course.cpp

#include "Course.h"

Course::Course(std::string id, std::string name) :

mId(id), mName(name)

{

}

std::string Course::getId() const

{

return mId;

}

void Course::setId(std::string id)

{

mId = id;

}

std::string Course::getName() const

{

return mName;

}

void Course::setName(std::string name)

{

mName = name;

}

#### Student.h

#ifndef STUDENT\_H

#define STUDENT\_H

#include <string>

class Student

{

public:

Student(std::string name, std::string address);

std::string getName() const;

void setName(std::string name);

std::string getAddress() const;

void setAddress(std::string address);

private:

std::string mName;

std::string mAddress;

};

#endif // STUDENT\_H

#### Student.cpp

#include "Student.h"

Student::Student(std::string name, std::string address) :

mName(name), mAddress(address)

{

}

std::string Student::getName() const

{

return mName;

}

void Student::setName(std::string name)

{

mName = name;

}

std::string Student::getAddress() const

{

return mAddress;

}

void Student::setAddress(std::string address)

{

mAddress = address;

}

### Körexempel

Nedan visas utdata från exempelkoden som demonstrerar att studenter kan registreras och avregistreras från kurser.

----- DA327G: Mjukvarukomponenter i C++ -----

Adam Naslund

Erik Andersson

Anders Eriksson

Peter Andersson

----- MA142G: Linjar algebra -----

Adam Naslund

Erik Andersson

Anders Eriksson

Peter Andersson

### Removing Anders from DA327G. ###

----- DA327G: Mjukvarukomponenter i C++ -----

Adam Naslund

Erik Andersson

Peter Andersson

----- MA142G: Linjar algebra -----

Adam Naslund

Erik Andersson

Anders Eriksson

Peter Andersson

### Removing Peter from all courses. ###

----- DA327G: Mjukvarukomponenter i C++ -----

Adam Naslund

Erik Andersson

----- MA142G: Linjar algebra -----

Adam Naslund

Erik Andersson

Anders Eriksson

### Removing all students from MA142G. ###

----- DA327G: Mjukvarukomponenter i C++ -----

Adam Naslund

Erik Andersson

### Analys

I lösningen befordras relationen mellan studenter och kurser till en separat klass vilket effektivt undviker cirkulära beroenden. Då klassen som hanterar relationerna inte använder några medlemsfunktioner från de objekt den hanterar och enbart hanterar pekare är de enda beroenden som finns i lösningen namnberoenden från relationshanteringsklassen till de två andra klasserna. Klasserna är därmed mycket löst kopplade vilket förenklar testning, underhåll och förståelse av koden.

En nackdel med lösningen är att en separat klass behövs samt att denna klass behöver instansieras för att fungera som en hållare för relationerna. Detta kan dock tänkas vara användbart i de fall där olika samlingar av relationer mellan studenter och kurser kan finnas, exempelvis för olika terminer.

En brist med lösningen är att klassen som hanterar relationerna tar emot och håller pekare till studenter och kurser. Därmed finns risken att null-pekare skickas in och delas ut, vilket kan resultera i krascher under körning, samt risken för att objekt involverade i relationer tas bort utan att relationen tas bort, vilket skulle resultera i så kallade ”dangling pointers” vilka ger odefinierat beteende, med risk för allvarliga fel, om de används.

En annan brist med lösningen är att varje relation internt representeras som två relationer, en från student till kurs och en från kurs till student. Detta är görs för att snabbt och enkelt kunna slå upp åt båda hållen, men har nackdelen att båda dessa relationer måste hållas uppdaterade, vilket ökar komplexiteten i relationsklassen samt ökar tidskomplexiteten för vissa av dess operationer.

# Applikation

## Problem

Ett enkelt spel ska skapas där spelaren ska ta sig uppåt i spelvärlden genom att hoppa mellan plattformar som genereras slumpmässigt. Vyn ska följa spelaren uppåt i världen och spelet är över då spelaren missar en plattform och faller utanför nederkanten av skärmen. Målet med spelet är att spelaren ska ta sig så högt upp som möjligt och spelarens poäng ska visas när spelet är över.

Lösningen ska utformas så att den är lätt att bygga vidare på och enkel att underhålla.

## Design

För att lösningen skulle vara lätt att bygga ut och underhålla designades den för att minimera kopplingar mellan komponenter och för att varje komponent skulle ha en väldefinierad funktion. Vidare utnyttjades moderna programmeringstekniker, både för att göra programkoden lättare att förstå och för att undvika problem relaterade till minneshantering och pekare.

I samtliga fall där det var möjligt gavs instanser en ägare till vilken deras livstid knöts. I de fall där ägandeskap behövde överföras användes unique\_ptr, vilket är en typ av smart pekare som tvingar ägandeskap och knyter instansens livstid till dess ägare. För delade resurser användes shared\_ptr, vilket är en smart pekare som håller reda på hur många kopior det finns av den. Delade resurser allokeras av den statiska klassen ResourceManager vilken använder funktionaliteten i shared\_ptr för att kunna frigöra resurser som inte längre används.

Banan skapas av LevelBuilder som använder EntityCreator vilken implementerar designmönstret factory för att enkelt kunna skapa olika banobjekt och för att nya sorters banobjekt lätt ska kunna läggas till utan att andra klasser behöver ändras.

För grafik, fönsterhantering, tangentbordsinput och tidtagning används biblioteket SFML 2.1 under zlib/libpng-licensen. I lösningen används även grafiska resurser från Platform Tileset Nature släppt under CC0 och från Kit the Firefox mascot av creek23 släppt under CC-BY-SA 3.0 samt typsnittet Mockup släppt under Open Font License Version 1.1.

TODO: Lägg till klass- och komponentdiagram, samt ta upp dessa i text.

## Kod

TODO: lägg in kod

### Filnamn1.cpp

### Filnamn2.h

### Filnamn2.cpp

### ...

### FilnamnN.h

### FilnamnN.cpp

## Körexempel

TODO: screenshot

## Analys

Genomgående användning av smarta pekare har helt eliminerat behovet av manuell minneshantering i lösningen, vilket är bra då det drastiskt minskar risken för minnesläckage, ett annars vanligt problem.

Användningen av smarta pekare har dock medfört en något ökad koppling mellan komponenter då även klasser som enbart hanterar pekare till objekt av en viss typ behöver ha tillgång till den kompletta typen (för att kunna anropa destruktorn), vilket innebär att de måste inkludera klassen även om de inte använder några av dess funktioner. Dessa beroenden är markerade med WRITEME i klassdiagrammet ovan för att särskilja dem från beroenden på gränssnitt.

# Slutsats

Vad ska det stå här? Seriöst, har ingen aning…

”Kapitel 4: Slutsats. Sammanfatta dina resultat och dina erfarenheter” Well fuck me.