

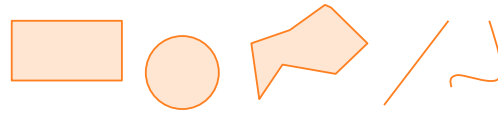
6. OBJEKTORIENTIERTE PROGRAMMIERUNG

180

Vererbung – Motivation

- Beispielhaftes Ziel: erweiterter Fundus an geometrischen Figuren

- Rechteck, Kreis, Polygon, Linie, Kurve etc.



- Gemeinsame Eigenschaften
 - Randfarbe, Randdicke, Füllfarbe, Muster etc.
- Gemeinsame Operationen
 - Zeichnen, Füllen, PointInX, Schneiden mit Geraden, etc.

181

Vererbung – Motivation

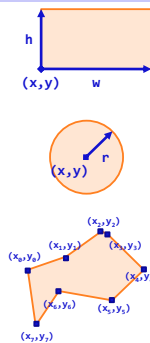
- Unterschiede

- Repräsentation

- Rechteck: x, y, w, h ,
- Kreis: x, y, r ,
- Polygon: Liste von Eckpunkten

- Operationen

- Zeichnen / Füllen verschieden aufwendig
- Schnitt mit Gerade oder Erkennung eines innenliegenden Punktes anders zu implementieren
- etc.



182

Vererbung – Motivation

Annahme: wir wollen ein Zeichenprogramm schreiben, welches über eine (verkettete) Liste auf seine geometrischen Objekte zugreift.

Wie lässt sich das realisieren?

Eine Möglichkeit (ohne OOP):

- Packe alle nötige Information für ein geometrisches Objekt in dieselbe Datenstruktur
- Füge eine «Schaltvariable» hinzu, mit der über die Art («den Typ») des Objektes entschieden werden kann.

183

Vererbung – Motivation

Realisierung ohne Vererbung:

```
public class Figure {
    enum Type {Rectangle, Circle};
    // «Umschalter» nach Art des Objektes
    Type typ;
    // Parameter für jede mögliche Figur (Kreis / Rechteck)
    Color col;
    int x, y, w, h, r;

    // Konstruktor für Rechteck
    Figure (Color rcol, int rx, int ry, int rw, int rh) {
        col = rcol; x = rx; y = ry; w = rw; h = rh;
        typ = Type.Rectangle;
    }
    // Konstruktor für Kreis
    Figure (Color ccol, int cx, int cy, int cr) {
        col = ccol; x = cx; y = cy; r = cr;
        typ = Type.Circle;
    }
    ...
}
```

Aufzählung (Enumeration) von benannten Konstanten. Zuweisung und Vergleich des gleichen Typs möglich. Wegen der Typprüfung besser als die Verwendung von einfachen Konstanten wie 0 oder 1.

184

Vererbung – Motivation

Realisierung ohne Vererbung:

```
public class Figure {
    ...
    public Color GetColor() { ... }
    public void SetColor(Color col) { ... }

    void DrawCircle(BufferedImage img) { ... }
    void DrawRectangle(BufferedImage img) { ... }

    public void draw(BufferedImage img) {
        switch (typ) {
            case Rectangle:
                DrawRectangle(img);
                break;
            case Circle:
                DrawCircle(img);
                break;
        }
    }
}
```

Je nach Wert von Variable typ wird DrawCircle oder DrawRectangle aufgerufen

Switch-Statement funktioniert wie eine If-Kaskade. Hier äquivalent:

```
if (typ = Type.Rectangle)
    DrawRectangle(img)
else if (typ = Type.Circle)
    DrawCircle(img)
```

185

Vererbung – Motivation

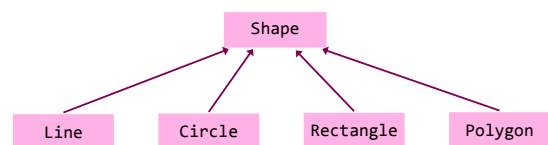
Nachteile obigen Vorgehens?

- Abundante Information
 - z.B. w, h nicht genutzt für den Kreis
 - z.B. `DrawCircle` nicht verwendet für Rechteck
- Administrativer Overhead
 - Der Programmierer muss Fallunterscheidung programmieren
- Nicht-invasives Erweitern des Codes unmöglich.
 - Möchte man einen neuen Typ von Figur hinzufügen, so muss man den ursprünglichen Code verändern.

186

Vererbung – Motivation

- Idee zur Vermeidung obiger Nachteile: Darstellung jeder geometrischen Figur als Erweiterung eines zugrunde liegenden Grundtyps **Shape**



- Gemeinsame Eigenschaften verbleiben (nur) in der Basisklasse **Shape**. Erklärung folgt.

187

Vererbung

Klassen können Eigenschaften (ver)erben:

```
public class Figure {
    Color col;

    // Default-Konstruktor jeder Figur
    Figure() { col = Color.black; }

    public void SetColor(Color c){ col = c; }
    public Color GetColor() { return col; }
}

public class Circle extends Figure {
    int x,y,r;

    // Konstruktor eines Kreises
    Circle (int cx, int cy, int cr) {
        x = cx; y = cy; r = cr;
    }
    ...
}
```

Datenfelder, welche für alle Figuren
deklariert sind

Methoden, welche für alle Figuren
deklariert sind

Datenfelder, welche nur für Circle
deklariert sind

188

Vererbung

Klassen können Eigenschaften (ver)erben:

```
public class Figure {
    Color col;

    // Default-Konstruktor jeder Figur
    Figure() { col = Color.black; }

    public void SetColor(Color c){ col = c; }
    public Color GetColor() { return col; }
}

public class Rectangle extends Figure {
    int x,y,w,h;

    // Konstruktor eines Rechtecks
    Rectangle (int rx, int ry, int rw, rh) {
        x = rx; y = ry; w = rw; h = rh;
    }
    ...
}
```

Datenfelder, welche nur für Rectangle
deklariert sind

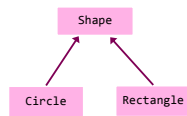
189

Vererbung

Klassen können Eigenschaften (ver)erben:

```
public class Figure { ... }
public class Circle extends Figure { ... }
public class Rectangle extends Figure { ... }

public class TestFigures {
    public static void main(String[] args) {
        ...
        Rectangle rectangle=new Rectangle(100,100,200,100);
        Circle circle= new Circle(100,100,50);
        circle.SetColor(Color.blue);
        rectangle.SetColor(Color.red);
        ...
    }
}
```



Aufruf der Methoden der
Basisklasse Figure

→ **Wiederverwendbarkeit** gemeinsam nutzbaren Codes
durch **Generalisierung**

190

Vererbung – Nomenklatur

```
class A {
    ...
}

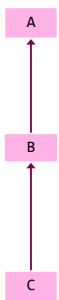
class B extends A {
    ...
}

class C extends B {
    ...
}
```

Basisklasse
(Superklasse)

Abgeleitete Klasse
(Subklasse)

«B und C erben von A»
«C erbt von B»



191

Vererbung – Kompatibilität

Ein abgeleitetes Objekt kann überall dort
verwendet werden, wo ein Basisobjekt gefordert
ist, aber nicht umgekehrt.

```
Rectangle rectangle = new Rectangle(0,0,10,10);
Figure figure = rectangle; // ok: Rectangle extends Figure
figure.SetColor(Color.red);
Rectangle r = new Circle(0,0,10); // type mismatch: cannot convert from Circle to Rectangle
Figure c = new Circle(0,0,10); // ok: Circle extends Figure
Circle circle = c; // type mismatch: cannot convert from Figure to Circle
```

192

Typprüfung

Wenn von einem Objekt bekannt ist, dass es zur Laufzeit von
abgeleitetem Typ ist, so kann man mit einem cast mit dynamischer
Typprüfung zuweisen:

```
Figure figure = new Rectangle(0,0,10,10);
Rectangle r = (Rectangle)figure; // ok: dynamisch geprüft
figure = new Circle(0,0,100);
Rectangle r2 = (Rectangle)figure; // Fehler zur Laufzeit, Compiler akzeptiert!
```

Um Fehler zur Laufzeit zu verhindern, oder allgemein einen Typ selbst
prüfen zu können, kann man **instanceof** verwenden:

```
if (figure instanceof Rectangle)
    System.out.println("figure is rectangle");
else
    System.out.println("figure is not rectangle");
```

193

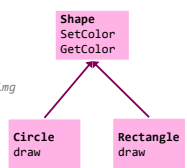
Polymorphie – Motivation

Ziel: Zeichnen der Figur entsprechend Ihrer Eigenschaften.

Vorgehen (noch ohne Polymorphie): Anbringen einer
entsprechenden Methode, z.B. bei Kreis und Rechteck:

```
public class Circle extends Figure {
    ...
    public void draw(BufferedImage img) {
        ... // Zeichne Kreis an x,y mit Radius r im Bild img
    }
}

public class Rectangle extends Figure {
    ...
    public void draw(BufferedImage img) {
        ... // Zeichne Rechteck an x,y mit Massen w,h im Bild img
    }
}
```



194

Polymorphie – Motivation

Ziel: Zeichnen der Figur entsprechend Ihrer Eigenschaften.

Vorgehen (noch ohne Polymorphie): Nun kann die jeweilige
Methode **draw** aufgerufen werden

```
public static void main(String[] args) {
    int width = 400;
    int height = 300;
    BufferedImage img = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
    ImageViewer panel = new ImageViewer(img);

    Rectangle rectangle=new Rectangle(100,100,200,100);
    Circle circle= new Circle(100,100,50);

    circle.SetColor(Color.blue);
    rectangle.draw(img);
    circle.draw(img);
    panel.repaint();
}
```

Ruft die jeweilige draw Methode auf,
gemäss statischem Typ.

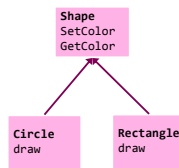
195

Polymorphie – Motivation

Ziel: Zeichnen der Figur entsprechend Ihrer Eigenschaften.

Problem dieses Vorgehens: so wie bisher deklariert, muss der statische Typ beim Aufruf stimmen.

```
public static void main(String[] args) {
    ...
    circle.draw(img);
    Figure figure = circle;
    figure.draw(img); // Fehler: Methode draw bei
                      // figure nicht definiert
    ...
}
```



Das ist insbesondere unpraktisch, wenn man z.B. eine verkettete Liste von Figuren führen will und die jeweilig passende **draw** Methode aufrufen möchte. Das war ja unser erklärtes Ziel!

196

Polymorphie

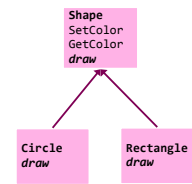
Polymorphie (in Java): Deklariert man eine (nicht statische) Methode mit gleichem Namen und gleicher Signatur in der Basisklasse und in abgeleiteten Klassen, so wird *zur Laufzeit* automatisch über die auszuführende Variante entschieden.

Man sagt, die Methode wird in der abgeleiteten Klasse *überschrieben*.

```
public class Figure {
    ...
    public void draw(BufferedImage img) { }
}

public class Circle extends Figure {
    ...
    public void draw(BufferedImage img) { ... // draw circle }
}

public class Rectangle extends Figure {
    ...
    public void draw(BufferedImage img) { ... // draw rect }
}
```



197

Polymorphie

Bei überschriebenen Methoden wird der dynamische Typ gewählt («dynamic dispatching»)

```
public class TestDynamicDispatch {
    public static void main(String[] args) {
        int width = 400;
        int height = 300;
        BufferedImage img = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
        ImageViewer panel = new ImageViewer(img);

        Figure f1 = new Rectangle(100,100,200,100);
        Figure f2 = new Circle(100,100,50);
        f1.draw(img); // wählt draw von Rectangle
        f2.draw(img); // wählt draw von Circle

        panel.repaint();
    }
}
```

Für C++-Kenner: in Java sind Methoden per Default virtuell. Es gibt kein «virtual» Schlüsselwort.

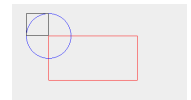
198

Polymorphie

Ziel erreicht!

```
public class TestFigureList {
    public static void DrawFigures(BufferedImage img, LinkedList<Figure> figures){
        ListIterator<Figure> it = figures.listIterator(0);
        while (it.hasNext())
        {
            Figure figure = it.next();
            figure.draw(img);
        }
    }

    public static void main(String[] args) {
        ...
        LinkedList<Figure> figures = new LinkedList<Figure>();
        figures.add(new Rectangle(Color.red, 100,100,200,100));
        figures.add(new Circle(Color.blue, 100,100,50));
        figures.add(new Rectangle(50,50,50,50));
        DrawFigures(img, figures);
        panel.repaint();
    }
}
```



199

Erweiterbarkeit

Mit den Konzepten der Vererbung und Polymorphie kann nun «nichtinvasiv» erweitert werden.

Neue Figuren können nun ohne Veränderung des anderen Quellcodes hinzugenommen und verwendet werden:

```
public class Line extends Figure{
    int x1,y1,x2,y2;

    Line (int lx1,int ly1,int lx2, int ly2)
    {
        x1 = lx1; x2 = lx2; y1 = ly1; y2 = ly2;
    }

    public void draw(BufferedImage img)
    {
        ... // Bresenham Algorithmus
    }
}
```

200

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Kapselung, Information Hiding

- Verbergen des Zustands und der Implementierungsdetails von Objekten
- Definition einer Schnittstelle zum Zugriff auf interne Datenstruktur → Abstraktion
- Ermöglicht das Sicherstellen von Invarianten

201

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Vererbung

- Objekte können Eigenschaften von Objekten erben
- Abgeleitete Objekte können neue Eigenschaften besitzen oder vorhandene überschreiben
- Macht Code- und Datenwiederverwendung möglich

202

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Polymorphie

- Ein Bezeichner kann abhängig von seiner Verwendung unterschiedliche Datentypen annehmen.
- Unterschiedliche Datentypen können bei gleichem Zugriff auf ihr gemeinsames Interface verschieden reagieren.
- Macht „nicht invasive“ Erweiterung von Bibliotheken möglich.

203

Fallstudie: Numerische Integration

Ziel: Erstellung eines Softwareframeworks zur numerischen Integration («Quadratur») einer gegebenen Funktion $f: \mathbb{R} \rightarrow \mathbb{R}$.

Problem aus Sicht der Softwareentwicklung: in Java gibt es keine Variablen von Funktionstyp. Wie können wir trotzdem ein generisches Tool bauen?

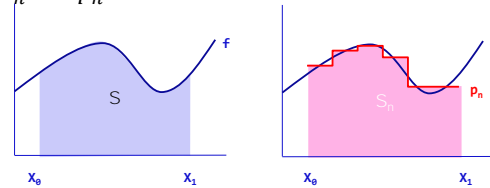
Antwort: wir verwenden Vererbung und Polymorphie für die generische Darstellung von reellwertigen Funktionen.

204

Numerische Integration

Einfachster Ansatz: Approximiere die Funktion f im gewünschten Integrationsintervall $[x_0, x_1]$ durch eine stückweise konstante Funktion p_n mit n Stücken.

Approximiere das Integral I von f durch das Integral I_n von p_n



205

Generischer Integrator

Abstrakte Funktionenklasse:

```
public abstract class Function {
    public abstract double Evaluate(double x);
}
```

Abstrakter Integrierer:

```
public abstract class Integrator {
    int n;

    public void SetNumberPieces(int pieces) {
        n = pieces;
    }

    public abstract double Integrate(Function f, double x0, double x1);
}
```

abstract :

- Abstrakte Klassen können nicht instanziiert werden
- Abstrakte Funktionen benötigen keinen Body, müssen jedoch von ererbenden Klassen, welche nicht abstract sind, implementiert werden.

206

Konkretisierung

Integrierer mit Rechteckregel

```
public class RectangleIntegrator extends Integrator {

    RectangleIntegrator(int pieces){
        n = pieces;
    }

    public double Integrate(Function f, double x0, double x1)
    {
        double sum = 0;
        double width=(x1-x0)/n; // Intervallbreite
        for (int i = 0; i<n; ++i)
        {
            double x = x0 + (i+0.5)*width; // Mitte des Intervalls
            double y = f.Evaluate(x); // Abgreifen des Funktionswertes
            sum += y*width; // Rechteckinhalt addieren
        }
        return sum;
    }
}
```

207

Zwei Funktionen

Parabel

```
class Square extends Function
{
    public double Evaluate (double x){return x*x;}
}
```

$$f(x) = x^2$$

Dichte der Normalverteilung

```
class Normal extends Function
{
    double mu;
    double sigma;

    Normal(double m, double s) {
        mu = m; sigma = s;
    }

    public double Evaluate(double x) {
        return 1/Math.sqrt(2*Math.PI)/sigma
            * Math.exp(-(x-mu)*(x-mu)/(2*sigma*sigma));
    }
}
```

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

208

Testbeispiel

```
public class TestIntegration {

    public static void main(String[] args) {
        Integrator integrator = new RectangleIntegrator(100);
        Function sq = new Square();
        Function N = new Normal(0,1);

        System.out.println("I(sq,0,2)= " + integrator.Integrate(sq, 0, 2) );

        for (int i=1; i<=3;++i)
            System.out.println("I(N,"+(-i) + "," + i + ")= "
                               + integrator.Integrate(N,-i,i));
    }
}
```

Ausgabe:

```
I(sq,0,2)= 2.6666000000000003
I(N,-1,1)= 0.6826975580161088
I(N,-2,2)= 0.9545141330225693
I(N,-3,3)= 0.9973041900876916
```

Deutung für normalverteilte Zufallsvariablen:
Innerhalb von 1 Standardabweichung liegen 68% der Daten
Innerhalb von 2 Standardabweichungen liegen 95% der Daten
Innerhalb von 3 Standardabweichungen liegen 99.7% der Daten

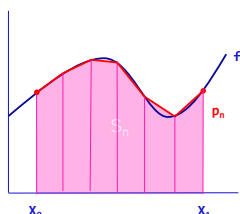
209

Andere Integrationsregeln

Trapezregel: Stückweise affine Approximation

```
public class TrapezoidalIntegrator extends Integrator{
    TrapezoidalIntegrator(int pieces){
        n = pieces;
    }

    public double Integrate(Function f, double x0, double x1){
        double sum = 0;
        double width=(x1-x0)/n;
        for (int i = 0; i<n; ++i)
        {
            double x = x0 + i*width;
            double y = f.Evaluate(x)
                + f.Evaluate(x+width);
            sum += y*width/2;
        }
        return sum;
    }
}
```



210

Beobachtung

Trapezregel ist für viele Funktionen kaum besser als die Rechteckregel.

Beispiel: Numerische Integration von $\sin(x)$ im Intervall $[0, \pi]$

- Eine Inspektion des Algorithmus zeigt, dass Rechteckregel und Trapezregel auch nahezu dasselbe tun.
- Für eine lineare Funktion stimmen die Regeln sogar überein.
- Die Rechteckregel überschätzt in der Regel das Integral, die Trapezregel unterschätzt.

n	Rechteck	Trapez
1	3.14159	0
2	2.22144	1.57079
4	2.05234	1.89611
8	2.01290	1.97423
16	2.00321	1.99357
32	2.00080	1.99839
64	2.00020	1.99960
128	2.00005	1.99990

211

Präzisierung*

Betrachtung eines Einzelstückes der numerischen Integration am Intervall $[l, r]$.

Annahme: Funktion f genügend oft differenzierbar

Taylor-Entwicklung um $\tilde{x} = \frac{l+r}{2}$

$$f(x) = f(\tilde{x}) + (x - \tilde{x})f'(\tilde{x}) + \frac{(x - \tilde{x})^2}{2}f''(\tilde{x}) + \frac{(x - \tilde{x})^3}{6}f^{(3)}(\tilde{x}) + \frac{(x - \tilde{x})^4}{24}f^{(4)}(\tilde{x}) + \dots$$

Integration von f ergibt (mit $\Delta = r - l$)

$$\int_l^r f(x)dx = \Delta \cdot f(\tilde{x}) + \frac{\Delta^3}{24} \cdot f''(\tilde{x}) + O(\Delta^5)$$

Dieser Term stellt genau die Rechteckregel dar!

212

Präzisierung*

Seien $I(f)$ das exakte Integral, $I^R(f)$ und $I^T(f)$ die Approximationen mit Rechteck- / Trapezregel.

Dann gelten

$$I^R(f) = I(f) - \frac{\Delta^3}{24}f''(\tilde{x}) + O(\Delta^5)$$

$$I^T(f) = I(f) + \frac{\Delta^3}{12}f''(\tilde{x}) + O(\Delta^5)$$

Das verhilft zu folgendem Trick

$$2 \cdot I^R(f) + I^T(f) = 3 \cdot I(f) + O(\Delta^5)$$

Daraus folgt die Simpson Regel

$$I(f) \approx I^S(f) := \frac{r-l}{6} \left(f(x_l) + 4f\left(\frac{r+l}{2}\right) + f(x_r) \right)$$

Der Fehler ist immerhin mit der dritten Potenz der Länge der Stücke kontrollierbar

Noch besser: Der Fehler fällt mit der 5ten Potenz!!!

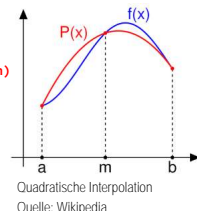
213

Simpson Regel*

Die Simpson-Integration korrespondiert mit quadratischer Interpolation von f

```
public class SimpsonIntegrator extends Integrator{
    SimpsonIntegrator(int pieces){
        n = pieces;
    }

    public double Integrate(Function f, double x0, double x1) {
        double sum = 0;
        double width = (x1-x0)/n;
        for (int i = 0; i < n; ++i)
        {
            double x = x0 + i*width;
            double y = f.Evaluate(x)+f.Evaluate(x+width)
                + 4*f.Evaluate(x+width/2);
            sum += y*width/6;
        }
        return sum;
    }
}
```



214

Newton Cotes Formeln*

- Die Newton-Cotes Formeln sind numerische Quadraturformeln zur approximativen Berechnung von Integralen durch Interpolation von Funktionen mit Lagrange Polynomen.
- Rechteck-, Trapez- und Simpson-Regeln sind Beispiele dieser Formeln.
- Eine andere berühmte numerische Integrationsmethode ist die Gauss-Quadratur
- Darüber hinaus sind Verfahren interessant, die das Abtastgitter in x-Richtung adaptiv wählen. Solche Verfahren sind wichtig bei Funktionen, die nicht überall gleich «glatt» sind.

215

Monte Carlo Integration

Nach dem Gesetz der grossen Zahlen konvergiert das empirische Mittel von identischen, unabhängig gezogenen Zufallsvariablen gegen den Erwartungswert der Zufallsvariablen*.

Es gilt also

$$f_n = \frac{1}{n} \sum_{i=1}^n f(X_i) \xrightarrow{f.s.} \mathbb{E}(f(X)) = \int_{-\infty}^{\infty} f(x)d\mu(x)$$

Wenn man die Zufallszahlen X_i gleichverteilt im Intervall $[x_0, x_1]$ zieht, so kann man das Integral von f also wie folgt approximieren

$$\int_{x_0}^{x_1} f(x)dx \approx (x_1 - x_0) \cdot \frac{1}{n} \sum_{i=1}^n f(X_i)$$

*in Wahrscheinlichkeit, sogar fast sicher 216

Monte Carlo Integration

Berechnet Mittelwert der Funktionswerte über dem Intervall durch Sampling und gibt dessen Produkt mit der Intervallbreite zurück.

```
public class MonteCarloIntegrator extends Integrator{

    MonteCarloIntegrator(int pieces){
        n = pieces;
    }

    public double Integrate(Function f, double x0, double x1) {
        double sum = 0;
        for (int i = 0; i < n; ++i)
        {
            double x = x0 + Math.random()*(x1-x0);
            sum += f.Evaluate(x);
        }
        return (x1-x0)*sum/n;
    }
}
```

217

Experiment: Vergleich der Verfahren

- Approximation des Integrals $\int_0^\pi \sin(x) dx = 2$

n	Rechteck	Trapez	Simpson	MonteCarlo
1	3.14159265	0	2.0943951	1.87890144
2	2.22144147	1.57079633	2.00455975	1.51525667
4	2.05234431	1.8961189	2.00026917	2.0769775
8	2.01290909	1.9742316	2.00001659	1.8314429
16	2.00321638	1.99357034	2.00000103	1.76335613
32	2.00080342	1.99839336	2.00000006	1.96607781
64	2.00020081	1.99959839	2	2.06425794
128	2.0000502	1.9998996	2	1.94378659
256	2.00001255	1.9999749	2	2.01618607
512	2.00000314	1.99999373	2	2.03129742
1024	2.00000078	1.99999843	2	1.9790763

218