```
// TestGenericVisitor.cpp

// DJD 2022-10-9 version 0.9 #6
// (C) Datasim Education BV 2022

/*
A really modern implementation of a generic Visitor pattern that is an improvement on
other approaches in C++:

        1. std::visit() and std::variant()
        2. Traditional Acyclic Visitor using subtype polymorphism, dynamic casting and
        possibly multiple inheritance.

The proposed solution is more maintainable than solutions 1 and 2 while at the same time
having better performance and reliability properties than solution 2. Our solution uses a
combination of
template methods and C++ Concepts.

We take the well-known example of 2d shapes (Point, Line, Circle) that can be visited in
different ways.
The two visitors represent new functionality for drawing and scaling these shapes.

We note that we do not have a class hierarchy nor virtual functions. All behaviour is
compile-time driven as it were.

C++ Concepts is a game changer.

I just want to say one word to you.
Just one word.
Yes, sir.
Are you listening?
Yes, I am.
Concepts.
Exactly how do you mean?
There's a great future in C++ Concepts.
Think about it.
Will you think about it?

This code contains:

        1. Visitor pattern based on C++20 Concepts (provide-requires contracts).
        2. Multimethods (the Holy Grail) .. what C++ tried for 30 years.
        3. Multimethods with variadic parameters, a) object with multiple visitors,
        b) a visitor with multiple objects.

        More later.
*/

#include <iostream>

// Visitor with templates,V2
struct Point;
struct Line;
struct Circle;

// Using C++20 Concepts to define contracts/protocols
// 1. The baseline contract between GOF Context and Visitor
```

```
template <typename Visitor, typename Context>
    concept IVisitor = requires (Visitor & v, Context & c)
{
    v.visit(c);
};

template <typename Visitor, typename Context>
    concept IAccept = requires (Visitor & v, Context & c)
 {
        c.accept(v);
 };

template <typename Visitor, typename Context>
    concept IAcceptVisitor = IVisitor<Visitor, Context> && IAccept<Visitor, Context>;

// Specialisations of Concept interfaces (from 2 to 1 template parameter)
template <typename Visitor>
    concept IPointVisitor = IVisitor<Visitor, Point>;

template <typename Visitor>
    concept ILineVisitor = IVisitor<Visitor, Line>;

template <typename Visitor>
    concept ICircleVisitor = IVisitor<Visitor, Circle>;

// An interface definition == set of abstract methods, essentially
// for later (maybe)
template <typename Visitor>
    concept ICombinedVisitor = IPointVisitor<Visitor> && ILineVisitor<Visitor> &&
ICircleVisitor<Visitor>;

/* End of Protocol definitions*/


struct Point
{
    double x;
    double y;

    Point() : x(0.0), y(0.0) {}
    Point(double X, double Y) : x(X), y(Y) {}

    template <typename T> requires IPointVisitor<T>
        void accept(T& t)
    {
        t.visit(*this);
    }
};

struct Line
{
    Point _p1;
    Point _p2;

    Line() : _p1(Point()), _p2(Point()) {}
    Line(const Point& p1, const Point& p2) : _p1(p1), _p2(p2) {}

    template <typename T> requires ILineVisitor<T>
```

```
        void accept(T& t)
        {
            t.visit(*this);
        }
};

struct Circle2
{ // NOT Using Concepts, take pot luck

    Point c;
    double r;

    Circle2() : c(Point()), r(1.0) {}
    Circle2(const Point& cen, double radius) : c(cen), r(radius) {}

    template <typename T>
        void accept(T& t)
        {
            t.visit(*this);
        }
};

struct Circle
{ // Using Concepts

    Point c;
    double r;

    Circle() : c(Point()), r(1.0) {}
    Circle(const Point& cen, double radius) : c(cen), r(radius) {}

    template <typename T>
        void accept(T& t) requires ICircleVisitor<T>
        {
            t.visit(*this);
        }
};

// Specific visitors (Draw and Scale)
struct Draw
{
    void visit(Point& p)
    {
        std::cout << "("<< p.x << ", " << p.y << ")\n";
    }

    void visit(Line& L)
    {
        Draw dr;
        std::cout << "[\n";
        dr.visit(L._p1);
        dr.visit(L._p2);
        std::cout << "]\n";
    }


  /*  void visit(Circle& cir)
    {
```

```
        Draw dr;
        dr.visit(cir.c);
    }
   */
};


struct Scale
{
    double fac;
    Scale(double factor) : fac(factor) {}
    void visit(Point& p)
    {
        p.x *= fac;
        p.y *= fac;
    }

    void visit(Line& L)
    {
        Scale mod(fac);
        mod.visit(L._p1);
        mod.visit(L._p2);
    }

};

// Multimethods, the Holy Grail of C++
template <typename Visitor, typename Context>
    void multimethod(Visitor& v, Context& c) requires IAcceptVisitor<Visitor, Context>
{
        // Just 2 different ways to do the same thing
        v.visit(c);
        c.accept(v);
}


// Command multipattern ... a list of objects on a single visitor
 template <typename T, typename Visitor>
    void command(Visitor v, T& c)
 {
       c.accept(v);
 }


template <typename T, typename Visitor, typename ... Types>
    void command(Visitor v , T& arg, Types& ... args)
{ // 1 visitor, multiple contexts

        command(v, arg);
        command(v, args...);
}

 // Command multipattern ... a list of visitors  on a single object
 template <typename T, typename Visitor>
    void command2(T& c, Visitor v)
  {
      //  c.accept(v);
        v.visit(c);
```

```
  }

template <typename T, typename Visitor, typename ... Types>
    void command2(T& arg, Visitor& v, Types& ... args)
{ // 1 context, multiple visitors

        command2(arg, v);
        command2(arg, args...);
 }


int main()
{

    // Contract with C++ 20 Concepts
    {
        std::cout << "Contracts, points and lines\n";
        Point p1(2.0, -3.0); Point p2(22.0, -23.0);
        Draw draw;
        Scale mod(0.5);

        Line myLine(p1, p2);
        myLine.accept(mod);
        myLine.accept(draw);
        mod.visit(myLine);
        draw.visit(myLine);
        myLine.accept(mod);
        draw.visit(myLine);
    }

    {
        // OOPs; I deliberately forgot to define Circle visitors
        Circle cir(Point(0.0, 0.0), 1.0);
        Draw draw;
       // cir.accept(draw); // 'Circle::accept' : the associated constraints are not
satisfied

        Circle2 cir2(Point(0.0, 0.0), 1.0);
        Draw draw2;
       // cir2.accept(draw2); // 'draw': unreferenced local variable

    }

    {
        // multimethods
        std::cout << "Multimethod\n";
        Point p(2.0, 4.0); Point p2(20.0, 41.0);
        Point p3(-2.0, -4.0); Point p4(21.0, 42.0);
        Line myLine(p, p4);
        Draw draw;
        Scale mod(0.5);

        // Magic
        multimethod(draw, p);
        multimethod(mod, p);
        multimethod(draw, p);

        std::cout << "variadics\n";
```

```
        command(draw, p);
        command(draw, p, p2, p3, p4, myLine);

        /*3. Multimethods with variadic parameters, a) object with multiple visitors,
        b) a visitor with multiple objects.*/

        command2(p, draw, mod, draw, mod, draw, mod, draw);
        command2(p2, draw, mod, draw);
    }
}
```