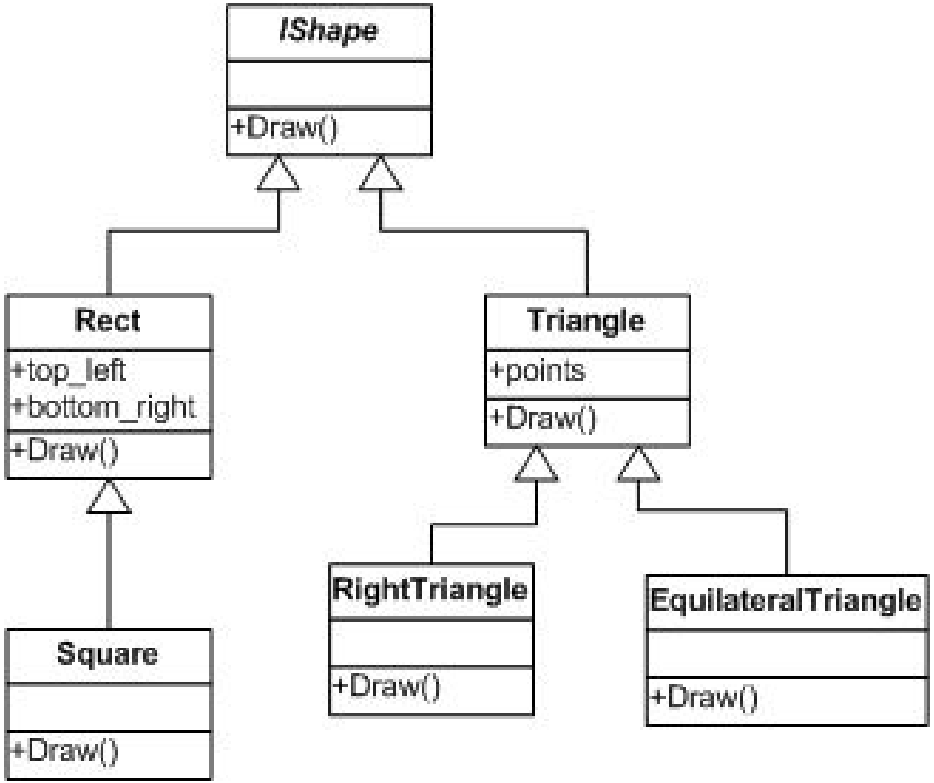
Multimethods.

Motivation

- no use of type casts of any kind (dynamic, static, reinterpret, const or C-style)
- no use of RTTI;
- no use of preprocessor;
- strong type safety;
- separate compilation;
- constant time of multimethod execution;
- no dynamic memory allocation (via `new` or `malloc`) during multimethod call;
- no use of nonstandard libraries;
- only standard C++ features is used.

Intersection

Hierarchy


IShape
+Draw()
Rect
+top_left
+bottom_right
+Draw()
Triangle
+points
+Draw()
Square
+Draw()
RightTriangle
+Draw()
EquilateralTriangle
+Draw()

The problem

```
CheckIntersection(IShape s0, IShape s1)
CheckIntersection(IShape s0, IShape s1, IShape s2)
CheckIntersection(IShape s0, … s4)
...
```

What we have

```
CheckIntersection(Rectangle s0, Triangle s1, Circle s2)
CheckIntersection(Rectangle s0, Rectangle s1, Rectangle s2)
CheckIntersection(Circle s0, Circle s1, Circle s2)
…
We have all the intersection we need
```

That would be easy

```
template<class T0, class T1, class T2>
struct Helper
{
  Do(IShape* s0, IShape* s1, IShape* s2)
  {
     CheckIntersection((T0) s0, (T1)s1, (T2) s2);
  }
}
```

Find types one by one

```
CheckIntersection(IShape* s0, IShape* s1, IShape* s2)
```

FirstStep

```
struct FirstStep {
    vector<IShape*> unknownArgs;
    ...
    }
```

SecondStep

```
template<typename T0>
struct SecondStep {
    T0* arg0;
    vector<IShape*> unknownArgs;
    ...
    }
```

ThirdStep

```
template<typename T0, typename T1>
struct ThirdStep {
    T0* arg0;
    T1* arg1;
    vector<IShape*> unknownArgs;
    ...
    }
```

Nth step

```
template<typename T0, typename T1, …>
struct NthStep {
    T0* arg0;
    T1* arg1;
    ...
    bool Intersect() {
        return CheckIntersection<T0, T1, …>( arg0, arg1, ...);
    }
}
```

How to find a type?

Double dispatch

```
struct Dispatcher {
  virtual void Dispatch( Triangle* shape ) {
    cout << "This is a triangle." << endl;
  }
  virtual void Dispatch( Rect* shape ) {
    cout << "This is a rect." << endl;
  }
};

void PrintType( IShape* shape ) {
  Dispatcher dsp;
  shape.Visit( dsp );
}
```

Basic idea

```
bool CheckIntersection(IShape* s0, IShape* s1)
{
  FirstStep<2> step;
  step.AddUnknownArg( s1 );
  Dispatcher<FirstStep<2>> dsp(step);
  s0.Visit( dsp );
  return dsp.GetResult();
}
```

Basic idea for 3

```
bool CheckIntersection(IShape* s0, IShape* s1, IShape* s2)
{
  FirstStep<3> step;
  step.AddUnknownArg( s1 );
  step.AddUnknownArg( s2 );
  Dispatcher<FirstStep<3>> dsp(step);
  s0.Visit( dsp );
  return dsp.GetResult();
}
```

Arity

```
template<int Arity>
struct FirstStep {
  template< typename TShape0 >
  void Intersect( TShape0* arg0 ) {
    SecondStep<TShape0, Arity> step;
    step.Arg0( arg0 );
    step.AddUnknownArgs( unknownArgs.begin()+1,
                         unknownArgs.end() );
    Dispatcher<SecondStep<TShape0, Arity>> dsp;
    unknown[0]->Visit( dsp );
    SetResult( dsp.GetResult() );
```

SecondStep for 2 arguments

```
template<>
struct SecondStep<2> {
  template< typename TShape1 >
  void Intersect( TShape1* _arg1 ) {
    bool result = RealIntersect( arg0, _arg1 );
    SetResult(result);
  }
```

SecondStep for n arguments

```
template<int Arity>
struct SecondStep {
template< typename TShape1 >
  void Intersect( TShape1* _arg1 )
    ThirdStep<TShape0, TShape1, Arity> step;
    step.Arg0( myArg0 );
    step.Arg1( arg1 );
    step.AddUnknownArgs(myUnknownArgs[1..]);
    Dispatcher<ThirdStep<TShape0, TShape1, Arity>> dsp;
    myUnknown[0]->Visit( dsp );
    SetResult( dsp.GetResult() );
```

Implementation

VisitableShapes

```
struct IShape_ {
  virtual void Visit( IDispatcher& disp ) {
    disp.Dispatch(this);
  }
};
```

Dispatch methods

```
struct Dispatcher {
  virtual void Dispatch( Triangle& _shape );
  virtual void Dispatch( Rectangle& _shape );
  virtual void Dispatch( Circle& _shape );
  ...
};
```

Basic idea. Reminder

```
bool CheckIntersection(IShape* s0, IShape* s1, IShape* s2)
{
  Dispatcher<FirstStep<3>> dsp;
  dsp.AddUnknownArg( s1 );
  dsp.AddUnknownArg( s2 );
  s0.Visit( dsp );
  return dsp.GetResult();
}
```

How to generate dispatch methods?

typedef

```
typedef int my_type;

void main()
{
    my_type v = 5;
}
```

typedef

```
struct my_struct {
    typedef int my_type;
};

void main()
{
    my_struct::my_type i = 5;
    vector<my_struct::my_type> v;
}
```

typedef

```
template<class T>
struct my_struct {
    typedef T my_type;
};

void main()
{
    my_struct<double>::my_type d = 0.5; // double
    my_struct<int>::my_type i = 5;      // int

}
```

typedef

```
template<int a>
struct my_struct {
    typedef double my_type;
};

template<>
struct my_struct<0> {
    typedef int my_type;
};

    my_struct<5>::my_type d = 0.5; // double
    my_struct<0>::my_type i = 5;       // int
```

TypeList

```
struct ListEnd_ {};

template< typename THead, typename TTail = ListEnd >
struct TypeList
{
  typedef THead head;
  typedef TTail tail;
};
```

Shapes

```
class Rect; class Square;
class Triangle; class RightTriangle;

typedef TypeList< Rect,
        TypeList< Square,
        TypeList< Triangle,
        TypeList< RightTriangle
                    > > > > Shapes;
```

Print TypeList

```
template<int TL>
struct my_printer {
    void print() {
        cout << typeid(typename TL::head).name() << " ";
        my_printer<typename TL::tail>.print();
    }
};

template<>
struct my_printer<ListEnd> {
    void print() { /* do nothing */ }
};
```

Print Shapes

```
typedef TypeList< Rect,
        TypeList< Square,
        TypeList< Triangle,
        TypeList< RightTriangle,
                  ListEnd  > > > > Shapes;

void main() {
    my_printer<Shapes>.print();
}
```

typename

```
struct A {
    typedef double my_type;
    static const int my_var;
};

template<class T>
struct B {
    vector<typename T::my_type> v;
    vector<T::my_type> v; // ERROR! Type expected
    static const int i = T::my_var;
};

B<A> b;
```

Reverse and print TypeList

Dispatcher

What we want

```
Dispatcher<Triangle, Rectangle, Circle, ...>
```

for any number for types,

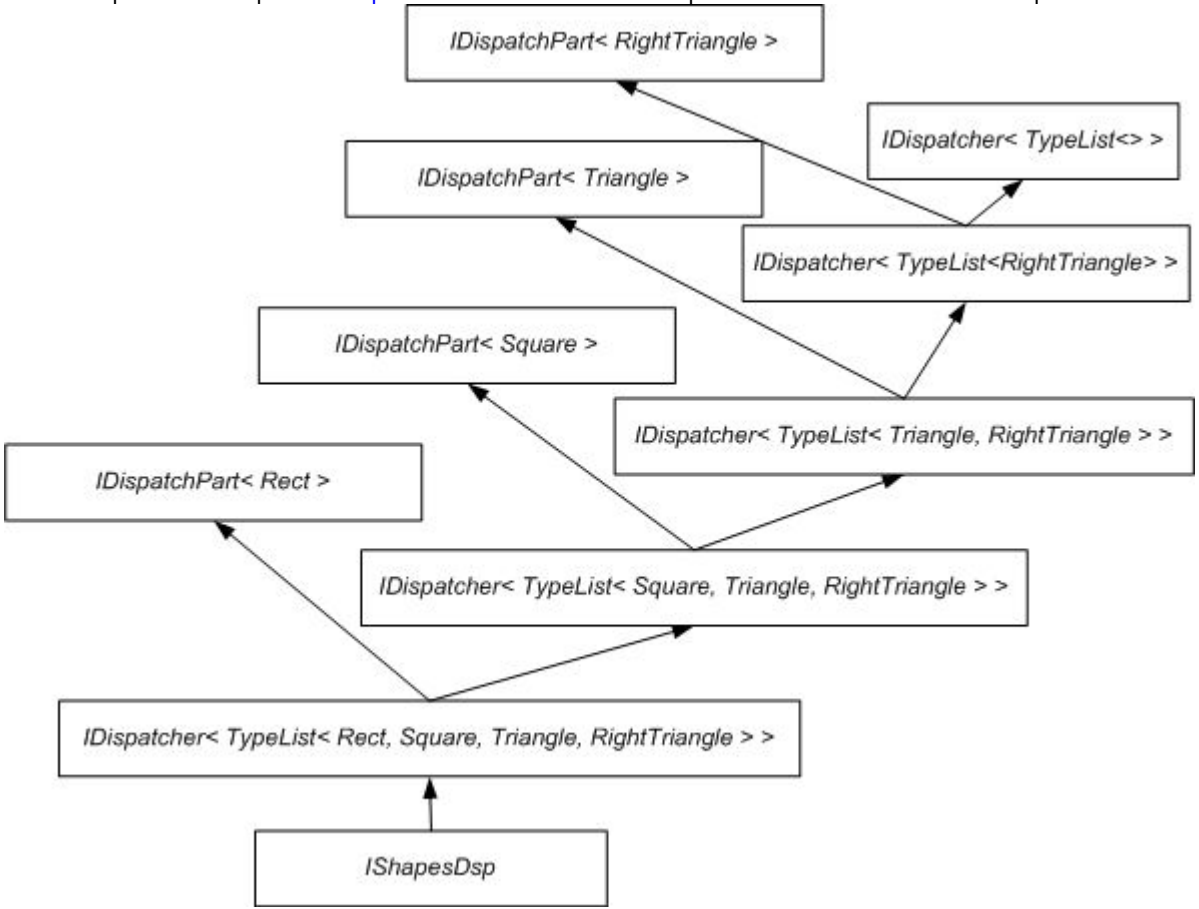
so is has:

```
virtual void Dispatch( Triangle& _shape );
virtual void Dispatch( Rectangle& _shape );
virtual void Dispatch( Circle& _shape );
```

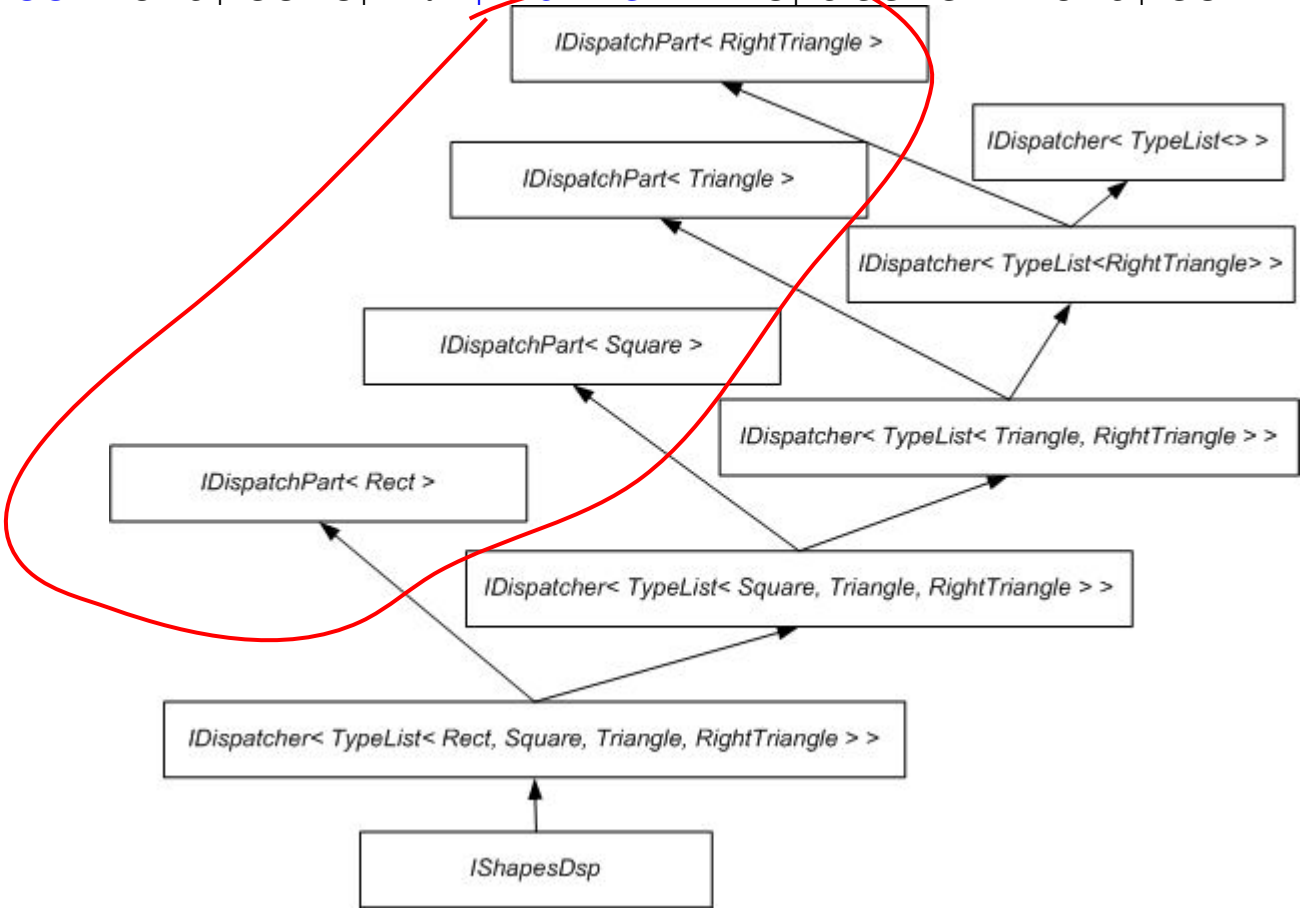
IDispatcher

```
template< typename TItem >
struct IDispatchPart {
  virtual void Dispatch( TItem& item ) = 0;
};

template< typename TItems >
struct IDispatcher
  : public IDispatchPart< typename TItems::head >
  , public IDispatcher< typename TItems::tail > {};

template<>
struct IDispatcher< ListEnd > {};
```

```
class IShapesDsp : public IDispatcher< Shapes > {};
```



```
class IShapesDsp : public IDispatcher< Shapes > {};
```



Basic idea. Reminder

```
bool CheckIntersection(IShape* s0, IShape* s1, IShape* s2)
{
  Dispatcher<FirstStep<3>> dsp;
  dsp.AddUnknownArg( s1 );
  dsp.AddUnknownArg( s2 );
  s0.Visit( dsp );
  return dsp.GetResult();
}
```

Analisys

- *2 * n* virtual methods are called;
- *n* auxiliary objects are constructed on stack;
- *n * (n + c)* non-virtual *get/set* methods are called (may be *inlined*), where *c* is a small constant.

Links

http://goo.gl/zEjoee

Visit

```
template< typename TShape, typename Dispatcher >
void VisitImpl( TShape& _self, Dispatcher& _dispatcher )
{
    IDispatchPart_< TShape >& part = _dispatcher;
    part.Dispatch( _self );
}

void Rect_::Visit( Dispatcher& _dispatcher ) {
  VisitImpl( *this, _dispatcher );
}
```