PointC2.h : Example

```
class PointC2
{
public:
              PointC2();
              PointC2(const Origin &);
              PointC2(const PointC2<FT> &p);
              ~PointC2();

  PointC2<FT> &operator=(const PointC2<FT> &p);
  bool        operator==(const PointC2<FT> &p) const;
  bool        operator!=(const PointC2<FT> &p) const;

  FT          x() const;
  FT          y() const;

  PointC2<FT> transform(const Aff_transformationC2<FT> &) const;

private:
  _Twotuple<FT>*  ptr() const;
};
```

CGAL Point Inheritance

```
class PointC2 : public Handle
  {basic methods and rought representations ....}

class CGAL_Point_2 : public Cartesian::Point_2
  {Methods the user will use}

class Cartesian
{
  typedef PointC2               Point_2;
  typedef SegmentC2             Segment_2;
  typedef LineC2                Line_2;
...
}
```

Overriding limitation

```
class Triangulation
{
  void insert(point* p1, point* p2)
{
    while (p1!=p2) {
    insert(*p1); p1++;
}
void insert(point p)
{
// insert the point in the triangulation
}
};

class Delaunay : public Triangulation
{
  void insert(point p)
{
// insert the point in the delaunay triangulation
}
};
```

Template : PointC2.h

```
template < class FT >
class PointC2 : public Handle
public:
              PointC2(const FT &x, const FT &y);

  PointC2<FT> &operator=(const PointC2<FT> &p);

  bool        operator==(const PointC2<FT> &p) const;
  bool        operator!=(const PointC2<FT> &p) const;

  FT          x() const;
  FT          y() const;
private:
  _Twotuple<FT>*  ptr() const;
};
```

Template parameters requierments

```
template<class NumType>
class My_Vector
{
private:
NumType X1,X2;

public:
  My_Vector(NumType x1, NumType x2)
  {
    X1=x1 ; X2=x2 ;
  }
  My_Vector<NumType>  operator+(const
                          My_Vector<NumType> v2)
  {
    return (My_Vector(X1 + v2.X1 , X2 + v2.X2));
  }
};
main()
{
  typedef My_Vector<float> float_vector;
  float_vector v1=float_vector(0.5,0.3);
  float_vector v2=float_vector(1.7,1.3);
  v1=v1+v2;
}
```

CGAL Polygon

```
template <class _R, class _FT, class _Point>
class CGAL_Polygon_traits_2_aux : public _R
{
  public:
    typedef _FT                           FT;
    typedef _Point                        Point_2;
    typedef CGAL_Segment_2<_R>            Segment_2;
    typedef CGAL_Vector_2<_R>             Vector_2;
    typedef CGAL_p_Less_xy<Point_2>       Less_xy;
    typedef CGAL_p_Less_yx<Point_2>       Less_yx;

    FT cross_product_2(const Vector_2& p, const Vector_2& q) const
      { return p.x() * q.y() - q.x() * p.y(); }

    FT determinant_2(const Point_2& p,const Point_2& q,const Point_2& r) const
      { return cross_product_2(p-q, p-r); }

    bool do_intersect(const Point_2& p1,
                      const Point_2& q1,
                      const Point_2& p2,
                      const Point_2& q2) const
      { return CGAL_do_intersect(Segment_2(p1,q1), Segment_2(p2,q2)); }

    CGAL_Orientation orientation(const Point_2& p,
                                 const Point_2& q,
                                 const Point_2& r) const
      { return CGAL_orientation(p, q, r); }
};

template <class _R>
class CGAL_Polygon_traits_2 :
  public CGAL_Polygon_traits_2_aux<_R, typename _R::FT, CGAL_Point_2<_R> >
{};
```

CGAL Polygon

```
template <class _Traits, class _Container>
class CGAL_Polygon_2 {
  private:
    _Container d_container;

  public:
    typedef _Traits Traits;
    typedef _Container Container;

    typedef typename _Traits::FT FT;
    typedef typename _Traits::Point_2 Point_2;
    typedef typename _Traits::Segment_2 Segment_2;

    typedef typename _Container::difference_type difference_type;
    typedef typename _Container::value_type value_type;

    CGAL_Polygon_2()
      { }
    CGAL_Polygon_2(const CGAL_Polygon_2<_Traits,_Container>& polygon)
      : d_container(polygon.d_container) { }

    template <class InputIterator>
    void insert(Vertex_iterator position,
                InputIterator first,
                InputIterator last)
      { d_container.insert(position, first, last); }

    bool is_simple() const
      { return CGAL_is_simple_2(d_container.begin(),
                                d_container.end(),
                                Traits()); }
...
};
```

CGAL Polygon instanciaton

```
#include<Cartesian.h>
#include<CGAL/Polygon_2.h>
#include<list.h>

typedef CGAL_Cartesian<double>            R;
typedef CGAL_Polygon_traits_2<R>          Traits;
typedef Traits::Point_2                   Point;
typedef list<Point>                       Container;
typedef CGAL_Polygon_2<Traits, Container>  Polygon;

int main()
{
  Polygon p;
  p.push_back(Point(0,0));
  p.push_back(Point(1,0));
  p.push_back(Point(1,1));
  p.push_back(Point(0,1));
  assert(p.is_convex());
}
```

Containers and iterators

```
#include<Cartesian.h>
#include<CGAL/Polygon_2.h>
#include<list.h>
#include<vector.h>

typedef CGAL_Cartesian<double>            R;
typedef CGAL_Point_2<R>                   Point;

main()
{
  list<Point> lp;
  vector<Point> vp;
  list<Point>::iterator lit;
  vector<Point>::iterator vip;
  lp.push_back(Point(0,0)); lp.push_back(Point(1,0));
  lp.push_back(Point(1,1)); lp.push_back(Point(0,1));
  vp.push_back(Point(0,0)); vp.push_back(Point(1,0));
  vp.push_back(Point(1,1)); vp.push_back(Point(0,1));
  for(lip=lp.begin() ; lip!=lp.end() ; lip++)
     cout << *lip << endl;
  for(vip=vp.begin() ; vip!=vp.end() ; vip++)
     cout << *vip << endl;
}
```

Genericity in the Kernel

```
main()
{
  typedef double              NT;
  typedef CGAL_Cartesian<NT>  space_rep;
  typedef CGAL_Point_2<space_rep> Point;
  typedef CGAL_Circle_2<space_rep> Circle;

  NT sprt2= sqrt(NT(2));
  Point p(sqrt2, sqrt2);
  Circle C(p,2);
  assert(C.has_on_boundary(p)); // false because double is used
}
```

Kernel evolution

Past

```
template<class _RT,class _FT>
CGAL_Homogeneous{
public:
    typedef _RT                                 RT;
    typedef _FT                                 FT;

    typedef CGAL_PointH2<FT,RT>                Point_2;
    typedef CGAL_VectorH2<FT,RT>               Vector_2;
...
}
```

Present

```
template<class _RT>
CGAL_Homogeneous{
public:
    typedef _RT                                 RT;
    typedef CGAL_Quotient<RT>                   FT;

    typedef CGAL_PointH2<FT,RT>                Point_2;
    typedef CGAL_VectorH2<FT,RT>               Vector_2;
...
}
```

Kernel evolution: problem of flexibility

```
class Cartesian
{
  typedef PointC2              Point_2;
  typedef SegmentC2            Segment_2;
  typedef LineC2               Line_2;
...
}
// LineC2 depend on PointC2. No flexibility
class LineC2
{
LineC2(PointC2 p1, PointC2 p2) {....}
}

class MyCartesian: public Cartesian
{
  typedef MyPointC2            Point_2;
}
```

Halfedge data structure : base classes

```
class CGAL_Vertex_min_base {
protected:
    void* halfedge;
public:
    typedef void*        Point;
    void*       halfedge()       {return halfedge;}
};


class CGAL_Halfedge_min_base {
protected:
    void* opp;
    void* nxt;
public:
void*       opposite()       { return opp;}
};

class CGAL_Facet_min_base {
protected:
    void* halfedge;
public:
    typedef void*        Plane;
    typedef void*        Normal;
    void*       halfedge()       { return halfedge;}
};
```

Halfedge vertex, halfedge and facet

```
template <class V, class H, class F>
class CGAL__HDS_list_vertex: public  V
{
public:
    typedef V                                       Base;
    typedef CGAL__HDS_list_vertex<V,H,F>            Vertex;
    typedef CGAL__HDS_list_halfedge<V,H,F>          Halfedge;
    typedef CGAL__HDS_list_facet<V,H,F>             Facet;
    typedef typename V::Point                       Point;

    Halfedge*       halfedge()       {return(Halfedge*)(V::halfedge());}
};

template <class V, class H, class F>
class CGAL__HDS_list_halfedge: public  H
{
public:
    typedef H                                       Base;
    typedef CGAL__HDS_list_vertex<V,H,F>            Vertex;
    typedef CGAL__HDS_list_halfedge<V,H,F>          Halfedge;
    typedef CGAL__HDS_list_facet<V,H,F>             Facet;

   Halfedge*       opposite()       {return(Halfedge*)(H::opposite());}
};

template <class V, class H, class F>
class CGAL__HDS_list_facet: public  F
{
    typedef F                                       Base;
    typedef CGAL__HDS_list_vertex<V,H,F>            Vertex;
    typedef CGAL__HDS_list_halfedge<V,H,F>          Halfedge;
    typedef CGAL__HDS_list_facet<V,H,F>             Facet;

    Halfedge*       halfedge()       {return(Halfedge*)(F::halfedge());}
};
```

Polyhedron

```
// Class with handle, iterator, circulator...
template <class HDS>
class CGAL__Polyhedron_vertex  : public HDS::Vertex  {
    public:
        typedef CGAL__Polyhedron_vertex<HDS>       Vertex;
        typedef CGAL__Polyhedron_halfedge<HDS>     Halfedge;
        typedef CGAL__Polyhedron_facet<HDS>        Facet;
...
};

template <class HDS>
class CGAL__Polyhedron_halfedge : public HDS::Halfedge {
...
};

template <class HDS>
class CGAL__Polyhedron_facet : public HDS::Facet {
...
};

template < class TR, class HDS >
class CGAL_Polyhedron_3 {
    public:
        typedef HDS                                Halfedge_data_structure;
        typedef CGAL__Polyhedron_vertex<HDS>       Vertex;
        typedef CGAL__Polyhedron_halfedge<HDS>     Halfedge;
        typedef CGAL__Polyhedron_facet<HDS>        Facet;
...
};
```

Drawback

```
template<class R>
struct Facet: public CGAL_Polyhedron_facet_min_base<R> {
  CGAL_color color;
};
// nothing more to change

template <class V, class H, class F>
struct list_vertex: public CGAL__HDS_list_vertex
{....
};
// need to modify list_halfedge, list_facet and Data structure too...
```

Polyhedron instanciation

```
#include<CGAL/Cartesian.h>
#include<CGAL/Halfedge_data_structure_polyhedron_default_3.h>
#include<CGAL/Polyhedron_defaulr_traits_3.h>
#include<CGAL/Polyhedron_3.h>

typedef CGAL_Cartesian<double>                                R;
typedef CGAL_Halfedge_data_structure_polyhedron_default_3<R> HDS;
typedef CGAL_Polyhedron_default_traits_3<R>                   Traits;
typedef CGAL_Polyhedron_3<Traits, HDS>                        Polyhedron;

int main()
{
  Polyhedron P;
  P.make_tetrahedron();
  return 0;
}
```

triangulation traits : Point type

```
class CGAL_Triangulation_test_point {
  public:
    typedef CGAL_Triangulation_test_point Point;
  protected:
    double _x, _y;
  public:
    typedef double TESTFT;
    CGAL_Triangulation_test_point() {}
    CGAL_Triangulation_test_point(const Point &p) : _x(p.test_x()), _y(p.test_y()) {}
    CGAL_Triangulation_test_point(double x, double y) : _x(x), _y(y) {}
    CGAL_Triangulation_test_point(double hx, double hy, double hw) : _x(hx/hw), _y(hy/hw) {}

    TESTFT test_x() const { return _x; }
    TESTFT test_y() const { return _y; }
    bool   compare(const Point &p) const { return test_x()==p.test_x() && test_y()==p.test_y(); }
    bool   uncompare(const Point &p) const { return !compare(p); }
    Point  &operator=(const Point &p) { _x=p.test_x(); _y=p.test_y(); return *this; }
    void   test_set(TESTFT x, TESTFT y) { _x=x; _y=y; }
    bool operator==(const Point &p) const {return this->compare(p);}
};
```

triangulation traits : segment and direction types

```
class CGAL_Triangulation_test_segment {
  public:
    typedef CGAL_Triangulation_test_point Point;
  protected:
    Point _p, _q;
  public:
  CGAL_Triangulation_test_segment() {}
    CGAL_Triangulation_test_segment(const Point &p, const Point &q)
      : _p(p), _q(q) {}

    void test_set(const Point &p, const Point &q) { _p=p; _q=q; }

 };

class CGAL_Triangulation_test_direction {
  public:
    typedef CGAL_Triangulation_test_point Point;
  protected:
    Point _p, _q;
  public:
    CGAL_Triangulation_test_direction() {}
    CGAL_Triangulation_test_direction(const Point &p, const Point &q)
      : _p(p), _q(q) {}

    CGAL_Triangulation_test_direction perpendicular(const CGAL_Orientation &o) const {
     return *this;
    }
    void test_set(const Point &p, const Point &q) { _p=p; _q=q; }
};
```

triangulation traits : Line and Ray type

```
class CGAL_Triangulation_test_line {
  public:
    typedef CGAL_Triangulation_test_point Point;
  protected:
    Point _p, _q;
  public:
    CGAL_Triangulation_test_line() {}
    CGAL_Triangulation_test_line(const Point &p, const Point &q)
      : _p(p), _q(q) {}

    CGAL_Triangulation_test_direction direction() {
     return CGAL_Triangulation_test_direction(_p,_q);
    }
  CGAL_Triangulation_test_line opposite() {
     return CGAL_Triangulation_test_line(_q, _p);
    }
    void test_set(const Point &p, const Point &q) { _p=p; _q=q; }
};

class CGAL_Triangulation_test_ray {
  public:
    typedef CGAL_Triangulation_test_point Point;
    typedef CGAL_Triangulation_test_direction Direction;
  protected:
    Point _p;
    Direction _d;
  public:
    CGAL_Triangulation_test_ray() {}
    CGAL_Triangulation_test_ray(const Point &p, const Direction &d)
      : _p(p), _d(d) {}
};
```

triangulation traits : Distance type

```
class CGAL_Triangulation_test_distance {
  public:
    typedef CGAL_Triangulation_test_point Point;
  protected:
    Point _p0, _p1, _p2;
    Point::TESTFT sqr(const Point::TESTFT x) const { return x*x; }
  public:
    CGAL_Triangulation_test_distance() {}
    CGAL_Triangulation_test_distance(const Point& p0) : _p0(p0) {}
    CGAL_Triangulation_test_distance(const Point& p0, const Point& p1) : _p0(p0), _p1(p1) {}
    CGAL_Triangulation_test_distance(const Point& p0, const Point& p1, const Point& p2)
        : _p0(p0), _p1(p1), _p2(p2) {}

    void set_point(int i, const Point& p)
    {   switch(i){
        case 0: _p0 = p; break;
        case 1: _p1 = p; break;
        default: _p2 = p;
        }
    }

    Point get_point(int i) const
    { switch(i){
      case 0: return _p0;
      case 1: return _p1;
      }
      return _p2;
    }

    CGAL_Comparison_result compare() const
    {
      Point::TESTFT d01 = sqr(_p0.test_x()-_p1.test_x()) + sqr(_p0.test_y()-_p1.test_y());
      Point::TESTFT d02 = sqr(_p0.test_x()-_p2.test_x()) + sqr(_p0.test_y()-_p2.test_y());
      return CGAL_compare( d01,d02);
    }
};
```

triangulation traits : Triangle type

```
class CGAL_Triangulation_test_triangle {
  public:
    typedef CGAL_Triangulation_test_point Point;
  protected:
    Point _p, _q, _r;
  public:
    CGAL_Triangulation_test_triangle() {}
    CGAL_Triangulation_test_triangle(const Point &p, const Point &q, const Point &r)
      : _p(p), _q(q), _r(r) {}

    const Point &p() const { return _p; }
    const Point &q() const { return _q; }
    const Point &r() const { return _r; }

    void test_set(const Point &p, const Point &q, const Point &r) { _p=p; _q=q; _r=r; }
};
```

triangulation traits : Traits type (1)

```
class CGAL__Triangulation_test_traits {
public:
  typedef CGAL_Triangulation_test_point     Point;
  typedef CGAL_Triangulation_test_segment   Segment;
  typedef CGAL_Triangulation_test_line      Line;
  typedef CGAL_Triangulation_test_ray       Ray;
  typedef CGAL_Triangulation_test_direction Direction;
  typedef CGAL_Triangulation_test_triangle  Triangle;
  typedef CGAL_Triangulation_test_distance  Distance;

  CGAL__Triangulation_test_traits() {}
  CGAL__Triangulation_test_traits(const CGAL__Triangulation_test_traits &) {}
  CGAL__Triangulation_test_traits &operator=
       (const CGAL__Triangulation_test_traits &) { return *this; }

  static
  CGAL_Comparison_result compare_x(const Point &p, const Point &q)
    { return CGAL_compare(p.test_x(),q.test_x()); }

  static
  CGAL_Comparison_result compare_y(const Point &p, const Point &q)
    { return CGAL_compare(p.test_y(), q.test_y()); }

  static
  bool compare(const Point &p, const Point &q)
    { return p.compare(q); }

  static
  CGAL_Orientation orientation(const Point &p, const Point &q, const Point &r)
    {
      typedef Point::TESTFT RT;
      RT det = (q.test_x()-p.test_x()) * (r.test_y()-p.test_y()) - (r.test_x()-p.test_x()) * (q.test_y()-p.test_y());
      if ( det < RT(0) ) return CGAL_CLOCKWISE;
      if ( RT(0) < det ) return CGAL_COUNTERCLOCKWISE;
      return CGAL_COLLINEAR;
    }
```

triangulation traits : Traits type (2)

```
static
CGAL_Oriented_side side_of_oriented_circle(const Point &p, const Point &q,
      const Point &r, const Point &t)
  {
    typedef Point::TESTFT RT;
    RT px( p.test_x()); RT py( p.test_y());
    RT qx( q.test_x()); RT qy( q.test_y());
    RT rx( r.test_x()); RT ry( r.test_y());
    RT tx( t.test_x()); RT ty( t.test_y());
    RT RT0(0);RT RT1(1);
    RT det = CGAL_det4x4_by_formula(px, py, px*px + py*py, RT1,
                                    qx, qy, qx*qx + qy*qy, RT1,
                                    rx, ry, rx*rx + ry*ry, RT1,
                                    tx, ty, tx*tx + ty*ty, RT1);
    return (det<RT0) ? CGAL_ON_NEGATIVE_SIDE
             : ((det==RT0) ? CGAL_ON_ORIENTED_BOUNDARY : CGAL_ON_POSITIVE_SIDE);
  }

static
Point circumcenter(const Point &p, const Point &q, const Point &r)
  {
    typedef Point::TESTFT FT;
    FT px( p.test_x()); FT py( p.test_y());
    FT qx( q.test_x()); FT qy( q.test_y());
    FT rx( r.test_x()); FT ry( r.test_y());
    FT px_qx( px - qx); FT py_qy( py - qy);
    FT qx_rx( qx - rx); FT qy_ry( qy - ry);
    FT rx_px( rx - px); FT ry_py( ry - py);
    FT px2_py2( px*px + py*py); FT qx2_qy2( qx*qx + qy*qy); FT rx2_ry2( rx*rx + ry*ry);
    FT num_x( px2_py2*qy_ry + qx2_qy2*ry_py + rx2_ry2*py_qy);
    FT num_y( px2_py2*qx_rx + qx2_qy2*rx_px + rx2_ry2*px_qx);
    FT den_x( ( px*qy_ry + qx*ry_py + rx*py_qy) * FT( 2));
    FT den_y( ( py*qx_rx + qy*rx_px + ry*px_qx) * FT( 2));
    return( Point( num_x/den_x, num_y/den_y));
  }
};
```

Triangulation instanciation

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Segment_2.h>

#include <CGAL/Triangulation_2.h>

typedef CGAL_Cartesian<double> rep_t;
typedef CGAL_Point_2<rep_t> point_t;

typedef CGAL__Triangulation_test_traits< rep_t > Ttraits;
typedef CGAL_Triangulation_vertex_base_2<Ttraits>      Vertex_base ;
typedef CGAL_Triangulation_face_base_2<Ttraits>        Face_base ;
typedef CGAL_Triangulation_default_data_structure_2<Ttraits,Vertex_base,Face_base> TDS ;
typedef CGAL_Triangulation_2< Ttraits , TDS> Triangulation;

int main()
{
  typedef Triangulation::Face                     Face;
  typedef Triangulation::Vertex                   Vertex;
  typedef typename Triangulation::Vertex_handle Vertex_handle;
  typedef typename Triangulation::Face_handle   Face_handle;
  typedef typename Triangulation::Edge_iterator Edge_iterator;
  typedef typename Triangulation::Face_iterator Face_iterator;
  Triangulation tr;
  assert(tr.dimension()==0);
  Vertex_handle v = tr.insert(point_t(10.0,10.0));
  assert(tr.dimension()==1);
  list<point_t> lp; // add some points in lp
  int i = tr.insert(lp.begin(), lp.end());
  assert(tr.dimension()==2);
  tr.remove(v);
}
```