## Appendix II. Supporting DBC using Genericity.

We have aimed to write generic classes that reduce to a minimum the amount of effort required by the programmer to enjoy Design by Contract support across an inheritance hierarchy in C++. At this stage we have only had time to consider single inheritance.

Note that there is no consideration given to the issue of what is to be done once constraint violation is detected: an exception is thrown as a default action. This is not because the issue is insignificant, but because it is beyond the scope of the discussion. The templates are easily amended if a different action is required.

Supporting class invariant, and method pre and post condition monitoring now only require the following:

(i) a typedef in each class in the hierarchy to define the name 'Base'. For all classes except the root of the hierarchy, this is defined to be the class from which the class inherits. For the root class, it is defined to be the class itself. This distinction is vital when utilising partial specialisation to identify the root of the hierarchy.

(ii) a global typedef for each method that requires Design by Contract support.

(iii) a typedef per class, per method for each definition of pre or post that the programmer wishes to give.

(iv) to enjoy full support for the 'old' mechanism – access for post conditions to a copy of the object state as it was when a method was entered – it is expedient for the root class of the hierarchy to inherit the struct 'preHandler'.

(v) an encoding of the pre(), post() and invariant() routines is required. These routines are provided with a means of accessing the object, the 'old' object, and the parameters of the method being called.

(vi) each routine to be monitored must declare a variable of a particular type on entry.

The method relies on the observation that constructors and destructors are automatically called across an the inheritance hierarchy, an observation which we first came across when posted to `comp.lang.c++.moderated` by Andrei Alexandrescu.

The idea is to create a hierarchy that parallels the inheritance heirarchy (but in completely different sense to that outlined in the body of this paper), whose constructors and destructors can be used to chain the calls to pre and post across the hierarchy, respectively.

The example takes three classes, PerfectNumberList derived from EvenNumberList derived from PositiveNumberList. (It might be considered more natural to define PerfectNumber derived from EvenNumber derived from PositiveNumber, and then define PerfectNumberList, but the definition given is convenient for illustrating the idea). The invariant (implicitly part of the post condition of any method) gets stronger as the hierarchy is descended. We define three methods, insert(..), erase(..) and push_back(..) for which we want to define pre and post conditions. The redefinition of insert(..) in PerfectNumberList is for illustration only. Like invariants, post conditions get stronger whilst pre conditions get weaker. The mechanism caters for this.

Prog 1 shows the code with no consideration of constraints.

Fragment 1 shows the coding of the constraints that we would like to add.

Fragment 2 shows the lines needed to support this. This is what is significant compared to, for instance, the work required to use the percolation pattern mentioned in the body of the paper.

Prog 2 shows the full program capable of monitoring constraints. It can be compiled and run using egcs 2.93.10 (February 28th 1999 release) or later.

Note first that the template classes DBC1<T1> and DBC2<T1, T2> shown in Prog 2 are identical except for the number of tempate parameters involved. A similarly identical DBC3<T1, T2, T3> etc. will be required for methods with more than two parameters. It is unfortunate that there is no mechanism to express an arbitrary number of template parameters. The discussion of DBC1<T1> below applies equally well to higher-order DBCns.

The template class DBC1<T1> contains a template class

DBC<typename Derived, bool Top=false, typename Dummy = typename Derived::Base>

This class defines methods pre() and post(), which are called by the constructor and destructor of the class respectively. Note that the destructor also checks the invariant, since this is implicitly part of the post condition of any method (indeed, it is also an implicit part of the precondition). Since pre conditions can be weakened as a class hierarchy is descended, no action is taken on detection of a violation unless and until all preconditions have been checked and found to be unsatisfied. For this reason it is necessary to identify the top of the hierarchy, which is the purpose of the 'Top' parameter.

The DBC<..> class also declares two pointers to objects of type Derived. The first of these is the object being monitored, the second is a copy of the object, as it was when the routine was entered. Again, the

Top parameter is used to identify the appropriate point at which to make this copy of the object and store it in _old.
The other significant feature of the class is the use of partial specialisation of a class member template of a template class to handle the point where the root of the hierarchy is encountered. In the primary template, the inner class DBC<..> is defined as being derived from
Derived::Base::DBC1<T1>::DBC<typename Derived::Base, false, Derived >
If we picture a version of inner class DBC<..> corresponding to each level of the hierarchy, then the class it is derived from is the version of DBC<..> corresponding to the previous level. This is what is required (so that the pre and post conditions are chained correctly).
However, this is not what is required for the root of the hierarchy. This case is distinguished by providing a partial specialisation of the DBC<..> class. This class is identified by having identical Derived and Dummy parameters. This is achieved by setting the typedef of the name Base in the root to the name of the root class, and passing the name of the class as the Dummy parameter. Note that all other classes in the hierarchy, there is no need to specify the Dummy parameter since it is given a suitable default value in the primary template that ensures that the Derived and Dummy parameters are not identical.
This base version stores copies of the parameters of the method so that they are accessible to the writer of the pre() and post() conditions.
Using these templates, all that is required to monitor the constraints defined in Fragment 1 in addition to the declarations shown in Fragment 2 is to declare a variable 'WatchDog' of instantiated-template-type at the beginning of each method that requires monitoring. The chain of constructors used to create this object will evaluate the disjunction of the preconditions of the method, and the chain of destructors called at the end of the method when the variable goes out of scope will evaluate the conjunction of the postconditions (including checking the conjunction of the invariants).
Note that it is not even necessary to declare the constraint definitions: since they are explicit specialisations, they are declared in the primary template or its partial specialisation.
The only problem that we see with this construction is that a rather inelegant device is required if two methods in the same hierarchy have the same signature. Consider, for example, the definition:
bool PositiveNumberList::Insert::pre() { /* ... */ }
The typedefs mask the fact that this definition is in fact
bool DBC2<vector<int>::iterator, int>::DBC<PositiveNumberList, false, PositiveNumberList>::pre() { /* ... */ }
which would be indistinguishable from the definition of another method in the same hierarchy with the same signature. Therefore in such a case it would be necessary to add an extra type parameter to one method in order to make the distinction apparent.
In the code shown we have not had time to consider issues such as inlining, const-ness, use of references instead of pointers, and privacy, but are aware that there is plenty of scope for their use, not least in the declaration of *_object.

```
/************************
*
*           Prog 1
*
************************/

#include <iostream>
#include <vector>

struct PositiveNumberList {
  vector<int> v;
  virtual vector<int>::iterator insert(vector<int>::iterator p, int i);
  virtual vector<int>::iterator erase(vector<int>::iterator p);
  virtual void push_back(const int&);
  friend ostream& operator<<(ostream &s, PositiveNumberList pnl);
  virtual const char* const className() { return "PositiveNumberList"; }
};

ostream& operator<<(ostream &s, PositiveNumberList pnl)
{
  s<<'(';
  for (vector<int>::iterator p = pnl.v.begin(); p<pnl.v.end(); p++) {
    s<<*p;
    if (p!=pnl.v.end()-1)
      s<<", ";
  }
```

```
  s<<')';
  return s;
}

vector<int>::iterator PositiveNumberList::insert(vector<int>::iterator p, int i)
{
  v.insert(p, i);
}

vector<int>::iterator PositiveNumberList::erase(vector<int>::iterator p)
{
  v.erase(p);
}

void PositiveNumberList::push_back(const int& i)
{
  v.push_back(i);
}

struct EvenNumberList : PositiveNumberList {
  virtual const char* const className() { return "EvenNumberList"; }
};

struct PerfectTriangleList : EvenNumberList {
  virtual vector<int>::iterator insert(vector<int>::iterator p, int i);
  virtual const char* const className() { return "PerfectTriangleList"; }
};

vector<int>::iterator PerfectTriangleList::insert(vector<int>::iterator p, int i)
{
  v.insert(p, i);
}

int main()
{
  PerfectTriangleList d;
  d.insert(d.v.begin(), 6);
  d.insert(d.v.begin(), 28);
  d.erase(d.v.begin());
  d.erase(d.v.begin());
  d.erase(d.v.begin());  // falls over
  return 0;
}

/************************
*
*           Fragment 1
*
************************/

#include <math.h>

bool PositiveNumberList::inv()
{
  bool result=true;
  vector<int>::iterator p=v.begin();
  for (; p<v.end(); ++p)
    if ((*p)<0)
      result=false;
  if (!result)
    throw(*this);
  else
    cout<<"invariant satisfied for PositiveNumber"<<endl;
  return result;
}

bool PositiveNumberList::Erase::pre()
{
  if (_p1<_object->v.begin() || _p1>=_object->v.end())
    throw(*this);
```

```
  cout<<"precondition evaluated for PositiveNumberList::Erase\n";
}

bool EvenNumberList::inv()
{
  bool result=true;
  vector<int>::iterator p=v.begin();
  for (; p<v.end(); ++p)
  if ((*p)%2!=0)
      result=false;
  if (!result)
    throw(*this);
  else
    cout<<"invariant satisfied for EvenNumber\n";
  return result;
}

bool PerfectTriangleList::inv()
{
  bool result=true;
  vector<int>::iterator p=v.begin();
  for (; p<v.end(); ++p) {
    int sum=1;
    for (int j=2; j<=(*p)/2; j++)
      if ((*p)%j==0)
        sum += j;
    if (sum!=(*p))
      result=false;
  }
  for (; p<v.end(); ++p) {
    int j=1;
    for (; j<2*(*p); j++) {
      double d=1+8*(*p);
      if (int(floor(sqrt(d)))^2==d)
        break;
    }
    if (j==*p)
      result=false;
  }
  if (!result)
    throw(*this);
  else
    cout<<"invariant satisfied for PerfectNumber\n";
  return result;
}

/************************
*
*           Fragment 2
*
************************/

typedef DBC2<vector<int>::iterator, int> Insert;
typedef DBC1<vector<int>::iterator> Erase;
typedef DBC1<int> Push_Back;

struct PositiveNumberList : preHandler {
  typedef PositiveNumberList Base;
  typedef ::Insert::DBC<PositiveNumberList, false, PositiveNumberList> Insert;
  typedef ::Erase::DBC<PositiveNumberList, false, PositiveNumberList> Erase;
  typedef ::Push_Back::DBC<PositiveNumberList, false, PositiveNumberList> Push_Back;
  bool inv();
  ...
};

vector<int>::iterator PositiveNumberList::insert(vector<int>::iterator p, int i)
{
  Insert WatchDog(this, p, i);
  ...
}
```

```
vector<int>::iterator PositiveNumberList::erase(vector<int>::iterator p)
{
  Erase WatchDog(this, p);
  ...
}

void PositiveNumberList::push_back(const int& i)
{
  Push_Back WatchDog(this, i);
  ...
}

struct EvenNumberList : PositiveNumberList {
  typedef PositiveNumberList Base;
  bool inv();
  ...
};

struct PerfectTriangleList : EvenNumberList {
  typedef EvenNumberList Base;
  typedef ::Insert::DBC<PerfectTriangleList, true> Insert;
  bool inv();
};

vector<int>::iterator PerfectTriangleList::insert(vector<int>::iterator p, int i)
{
  Insert WatchDog(this, p, i);
  ...
}

/************************
*
*           Prog 2
*
************************/
#include <iostream>
#include <vector>
#include <math.h>

template <typename T1> struct DBC1 {
  template <typename Derived, bool Top=false, typename Dummy = typename Derived::Base > struct
DBC : Derived::Base::DBC1<T1>::DBC<typename Derived::Base, false, Derived> {
    bool pre() { return true;}
    DBC(Derived* object, T1 p1) : Derived::Base::DBC1<T1>::DBC<typename Derived::Base, false,
Derived >(object, p1), _object(object)
    {
      cout<<"evaluating precondition for object of class: "<<_object->className()<<endl;
      if (Top)
        _old = new Derived(*object);
      _object->setPre(pre());
      if (Top) {
        if (!_object->_preDisjunct)
          throw(*_object);
        else
          _object->_preDisjunct=false;
      }
    }
    bool post() { return true;}
    ~DBC() {
      cout<<"evaluating postcondition for object of class: "<<_object->className()<<endl;
      post();
      _object->inv();
    }
    Derived* _object, *_old;
  };

  template < typename Derived > struct DBC<Derived, false, Derived> {
    bool pre() { return true;}
```

```
    DBC(Derived * object, T1 p1) : _object(object), _p1(p1)
    {
      cout<<"evaluating precondition for object of class: "<<_object->className()<<endl;
      _object->setPre(pre());
    }
    bool post() { return true;}
    ~DBC() {
      cout<<"evaluating postcondition for object of class: "<<_object->className()<<endl;
      post();
      _object->inv();
    }
    Derived * _object, *_old;
    T1 _p1;
  };
};

template <typename T1, typename T2 > struct DBC2 {
  template <typename Derived, bool Top=false, typename Dummy = typename Derived::Base > struct
DBC : Derived::Base::DBC2<T1, T2>::DBC<typename Derived::Base, false, Derived> {
    bool pre() { return true;}
     DBC(Derived* object, T1 p1, T2 p2) : Derived::Base::DBC2<T1, T2>::DBC<typename Derived::Base,
false, Derived >(object, p1, p2), _object(object)
    {
      cout<<"evaluating precondition for object of class: "<<_object->className()<<endl;
      if (Top)
        _old = new Derived(*object);
      _object->setPre(pre());
      if (Top) {
        if (!_object->_preDisjunct)
          throw(*_object);
        else
          _object->_preDisjunct=false;
      }
    }
    bool post() { return true;}
    ~DBC() {
      cout<<"evaluating postcondition for object of class: "<<_object->className()<<endl;
      post();
      _object->inv();
    }
    Derived* _object, *_old;
  };

  template <typename Derived> struct DBC<Derived, false, Derived > {
    bool pre() { return true;}
    DBC(Derived * object, T1 p1, T2 p2) : _object(object), _p1(p1), _p2(p2)
    {
      cout<<"evaluating precondition for object of class: "<<_object->className()<<endl;
      _object->setPre(pre());
    }
    bool post() { return true;}
    ~DBC() {
      cout<<"evaluating postcondition for object of class: "<<_object->className()<<endl;
      post();
      _object->inv();
    }

    Derived * _object, *_old;
    T1 _p1;
    T2 _p2;
  };
};

struct preHandler {
  bool _preDisjunct;
  bool setPre(bool b) { return _preDisjunct = _preDisjunct || b; }
  preHandler() { _preDisjunct=false; }
};
```

```
typedef DBC2<vector<int>::iterator, int> Insert;
typedef DBC1<vector<int>::iterator> Erase;
typedef DBC1<int> Push_Back;

struct PositiveNumberList : preHandler {
  typedef PositiveNumberList Base;
  typedef ::Insert::DBC<PositiveNumberList, false, PositiveNumberList> Insert;
  typedef ::Erase::DBC<PositiveNumberList, false, PositiveNumberList> Erase;
  typedef ::Push_Back::DBC<PositiveNumberList, false, PositiveNumberList> Push_Back;
  vector<int> v;
  bool inv();
  virtual vector<int>::iterator insert(vector<int>::iterator p, int i);
  virtual vector<int>::iterator erase(vector<int>::iterator p);
  virtual void push_back(const int&);
  friend ostream& operator<<(ostream &s, PositiveNumberList pnl);
 virtual const char* const className() { return "PositiveNumberList"; }
};

ostream& operator<<(ostream &s, PositiveNumberList pnl)
{
  s<<'(';
  for (vector<int>::iterator p = pnl.v.begin(); p<pnl.v.end(); p++) {
    s<<*p;
    if (p!=pnl.v.end()-1)
      s<<", ";
  }
  s<<')';
  return s;
}

bool PositiveNumberList::inv()
{
  bool result=true;
  vector<int>::iterator p=v.begin();
  for (; p<v.end(); ++p)
    if ((*p)<0)
      result=false;
  if (!result)
    throw(*this);
  else
    cout<<"invariant satisfied for PositiveNumber"<<endl;
  return result;
}

bool PositiveNumberList::Erase::pre()
{
  if (_p1<_object->v.begin() || _p1>=_object->v.end())
    throw(*this);
  cout<<"precondition evaluated for PositiveNumberList::Erase\n";
}

vector<int>::iterator PositiveNumberList::insert(vector<int>::iterator p, int i)
{
  Insert WatchDog(this, p, i); v.insert(p, i);
}

vector<int>::iterator PositiveNumberList::erase(vector<int>::iterator p)
{
  Erase WatchDog(this, p); v.erase(p);
}

void PositiveNumberList::push_back(const int& i)
{
  Push_Back WatchDog(this, i); v.push_back(i);
}

struct EvenNumberList : PositiveNumberList {
  typedef PositiveNumberList Base;
  bool inv();
```

```
  virtual const char* const className() { return "EvenNumberList"; }
};

bool EvenNumberList::inv()
{
  bool result=true;
  vector<int>::iterator p=v.begin();
  for (; p<v.end(); ++p)
  if ((*p)%2!=0)
      result=false;
  if (!result)
    throw(*this);
  else
    cout<<"invariant satisfied for EvenNumber\n";
  return result;
}

struct PerfectTriangleList : EvenNumberList {
  typedef EvenNumberList Base;
  typedef ::Insert::DBC<PerfectTriangleList, true> Insert;
  bool inv();
  virtual vector<int>::iterator insert(vector<int>::iterator p, int i);
  virtual vector<int>::iterator erase(vector<int>::iterator p);
  virtual void push_back(const int&);
  virtual const char* const className() { return "PerfectTriangleList"; }
};

bool PerfectTriangleList::inv()
{
  bool result=true;
  vector<int>::iterator p=v.begin();
  for (; p<v.end(); ++p) {
    int sum=1;
    for (int j=2; j<=(*p)/2; j++)
      if ((*p)%j==0)
        sum += j;
    if (sum!=(*p))
      result=false;
  }
  for (; p<v.end(); ++p) {
    int j=1;
    for (; j<2*(*p); j++) {
      double d=1+8*(*p);
      if (int(floor(sqrt(d)))^2==d)
        break;
    }
    if (j==*p)
      result=false;
  }
  if (!result)
    throw(*this);
  else
    cout<<"invariant satisfied for PerfectNumber\n";
  return result;
}

vector<int>::iterator PerfectTriangleList::insert(vector<int>::iterator p, int i)
{
  Insert WatchDog(this, p, i);
  v.insert(p, i);
}

int main()
{
  PerfectTriangleList d;
  try {
    d.insert(d.v.begin(), 6);
    d.insert(d.v.begin(), 28);
    d.erase(d.v.begin());
```

```
    d.erase(d.v.begin());
    d.erase(d.v.begin());
  }
  catch (PerfectTriangleList::Erase) { cout<<"constraint violation for PerfectTriangleList::Erase"<<endl; }
  catch (EvenNumberList::Erase) { cout<<"constraint violation for EvenNumberList::Erase"<<endl; }
  catch (PositiveNumberList::Erase) { cout<<"constraint violation for PositiveNumberList::Erase"<<endl; }
  return 0;
}
```