Interface Versioning in C++

Steve Love

ACCU London October 2010

The problem

One library, many clients

Recompilation causes redeployment

Synchronising release cycles is painful

Quarantined release environments are prone to error

The goal

Design a library that:

1. Can grow as requirements arise
2. Doesn’t force redeployment of clients every upgrade
3. Is easy to use without undue effort

Dependency Management

Requires a loosening of the coupling between the library and its clients

Cause and effect

Changes to a class cause clients to recompile when:

- Public member functions change
- Base class list is changed
- Data members change
- Protected and Private member functions change

```
#pragma once

namespace inventory
{
  class part
  {
    public:
      unsigned id() const;

    private:
      unsigned num;
  };
}
```

Solution

Hide changes behind an interface

```
#pragma once

namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual int id() const = 0;
  };

  part * create_part();
}
```

```
#include "part.h"

namespace
{
  class realpart : public inventory::part
  {
    public:
      realpart();
      int id() const;

    private:
      int num;
  };
}

namespace inventory
{
  part * create_part()
  {
    return new realpart;
  }
}
```

Indirectly

Interfaces are only part of the solution...

- **Public member functions change**
- Base class list is changed
- Data members change
- Protected and Private member functions change

A false hope

Necessity and sufficiency

Interfaces are vital

...but not sufficient on their own

Padding interfaces at the bottom

...is common.

But wrong.

Why not?

```
class properties
{
public:
  virtual int integer() const { return 0; }
  virtual double floatingpoint() const { return 0; }
};

int main()
{
  properties p;
}
```

```
cl /EHsc /FAs test.cpp
```

```
??_7properties@@6B@ DD FLAT:??_R4properties@@6B@  ; properties::‘vftable’
  DD  FLAT:?integer@properties@@UBEHXZ
  DD  FLAT:?floatingpoint@properties@@UAENXZ
; Function compile flags: /Odtp
```

Changed interface

```
class properties
{
public:
  virtual int integer() const { return 0; }
  virtual double floatingpoint() const { return 0; }

  virtual void integer( int ) { }
  virtual void floatingpoint( double ) { }
};

int main()
{
  properties p;
}
```

```
cl /EHsc /FAs test.cpp
```

The result

Old:

```
??_7properties@@6B@ DD FLAT:??_R4properties@@6B@  ; properties::‘vftable’
  DD  FLAT:?integer@properties@@UBEHXZ
  DD  FLAT:?floatingpoint@properties@@UAENXZ
; Function compile flags: /Odtp
```

New:

```
??_7properties@@6B@ DD FLAT:??_R4properties@@6B@  ; properties::‘vftable’
  DD  FLAT:?integer@properties@@UAEXH@Z
  DD  FLAT:?integer@properties@@UBEHXZ
  DD  FLAT:?floatingpoint@properties@@UAEXN@Z
  DD  FLAT:?floatingpoint@properties@@UAENXZ
; Function compile flags: /Odtp
```

The true path

Old new thing

Adding a method to an interface is bad.

Adding a new interface to a library is benign.

Simple interfaces

```
#pragma once

#include <string>

namespace inventory
{
  class part
  {
    public:
      virtual ~part();

      virtual unsigned id() const = 0;
      virtual const std::string & name()
            const = 0;
  };

  part * create_part();
}
```

```
#include <part.h>
#include <iostream>
#include <memory>

// Also link to inventory.lib

int main()
{
  using namespace std;
  using namespace inventory;

  unique_ptr< part > p( create_part() );
  cout << p->id() << endl;
  cout << p->name() << endl;
}
```

Extended interface

```
#pragma once

#include <string>

namespace inventory
{
  class part
  {
    public:
      virtual ~part();
      virtual unsigned id() const = 0;
      virtual const std::string & name()
            const = 0;
  };
  class part_v2 : public part
  {
    public:
      using part::id;
      using part::name;

      virtual void id( unsigned val ) = 0;
      virtual void name( const std::string
            & val ) = 0;
  };

  part * create_part();
}
```

```
#include <part.h>
#include <iostream>
#include <memory>

// Also link to inventory.lib

int main()
{
  using namespace std;
  using namespace inventory;

  unique_ptr< part_v2 > p( dynamic_cast<
       part_v2* >( create_part() ) );
  p->id( 100 );
  p->name( "wingnut" );
  cout << p->id() << endl;
  cout << p->name() << endl;
}
```

Implementing part_v2

```
#include "part.h"

namespace
{
  class realpart : public inventory::part_v2
  {
    public:
      realpart();

      unsigned id() const;
      const std::string & name() const;

      void id( unsigned val );
      void name( const std::string & val );

    private:
      unsigned num;
      std::string namestr;
  };
}
```

Wrong turn!

```
#include <string>

class part
{
  public:
    virtual int id() const = 0;
    virtual const std::string & name() const = 0;
};
class part_v2 : public part
{
  public:
    virtual void id( int val ) = 0;
    virtual void name( const std::string & val ) = 0;
};
```

It’s not the interface that’s the problem...

v1 and v2 side-by-side

...it's the implementation.

```
class realpart : public part
{
  public:
    int id() const { return num; }
    const std::string & name() const { return namestr; }

  private:
    int num;
    std::string namestr;
};

class realpart_v2 : public part_v2
{
  public:
    int id() const { return num; }
    const std::string & name() const { return namestr; }
    void id( int val ) { }
    void name( const std::string & val ) { }

  private:
    int num;
    std::string namestr;
};
```

More vtables

```
??_7realpart@@6B@ DD FLAT:??_R4realpart@@6B@    ; realpart::‘vftable’
  DD  FLAT:?id@realpart@@UBEHXZ
  DD  FLAT:?
    name@realpart@@UBEABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@XZ

; Function compile flags: /Odtp
```

```
??_7realpart_v2@@6B@ DD FLAT:??_R4realpart_v2@@6B@  ; realpart_v2::‘vftable’
  DD  FLAT:?id@realpart_v2@@UBEHXZ
  DD  FLAT:?
    name@realpart_v2@@UBEABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@XZ

  DD  FLAT:?id@realpart_v2@@UAEXH@Z
  DD  FLAT:?
    name@realpart_v2@@UAEXABV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@@Z

; Function compile flags: /Odtp
```

DANGER!

This will appear to work, but depends on the implementation

Split the implementations

```
#include "part.h"

namespace
{
  class realpart : public inventory::part
  {
    public:
      realpart();

      unsigned id() const;
      const std::string & name() const;

    protected:
      unsigned num;
      std::string namestr;
  };

  class realpart_v2 : public inventory::part_v2, public
        realpart
  {
    public:
      using part::id;
      using part::name;

      void id( unsigned val );
      void name( const std::string & val );
  };
```

```
namespace inventory
{
  part::~part()
  {
  }

  part * create_part()
  {
    return new realpart_v2;
  }
}
```

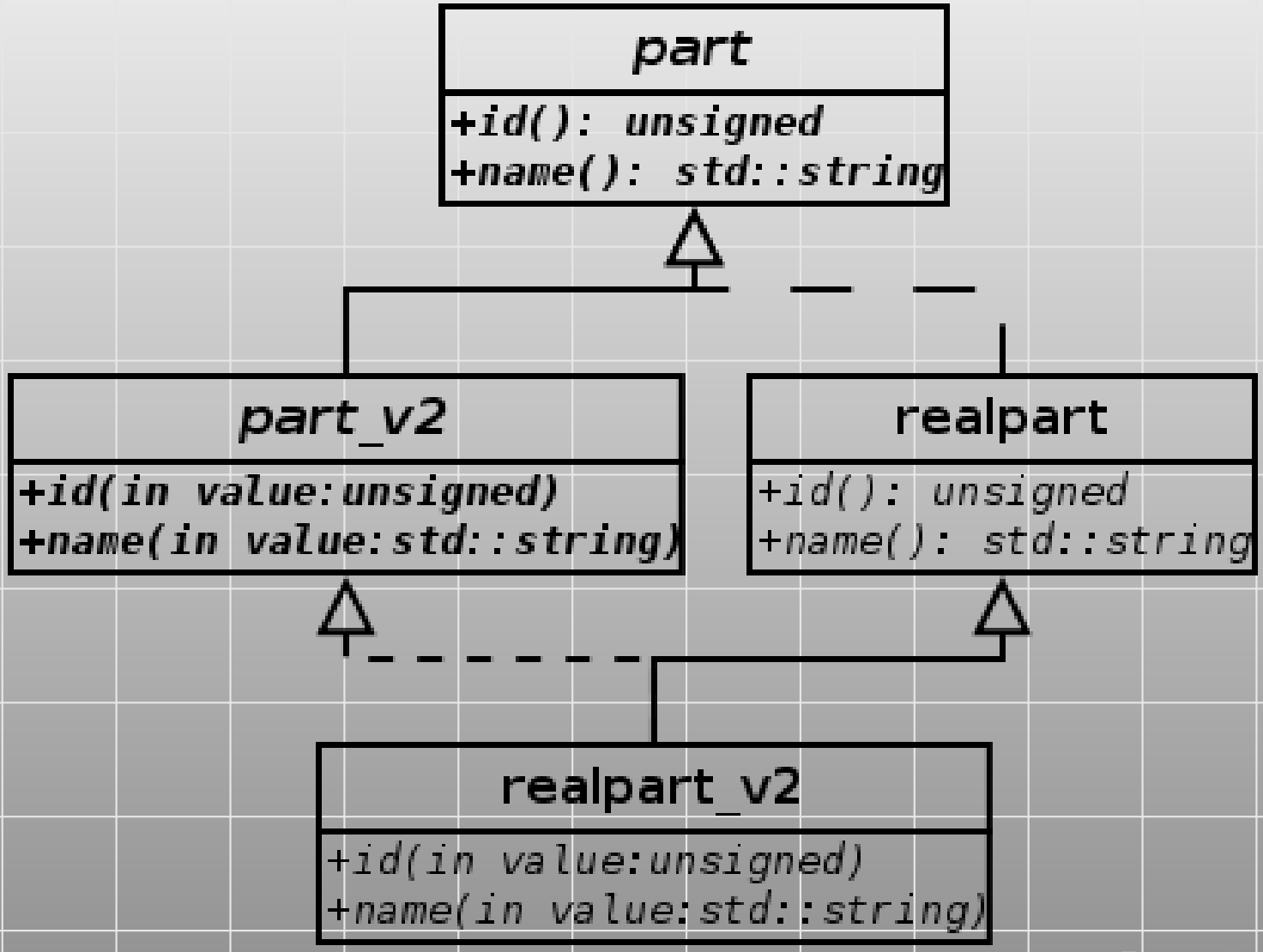
Split the implementations

```
#include "part.h"

namespace
{
  class realpart : public inventory::part
  {
    public:
      realpart();

      unsigned id() const;
      const std::string & name() const;

    protected:
      unsigned num;
      std::string namestr;
  };

  class realpart_v2 : public inventory::part_v2, public
        realpart
  {
    public:
      using part::id;
      using part::name;

      void id( unsigned val );
      void name( const std::string & val );
  };
```

```
namespace inventory
{
  part::~part()
  {
  }

  part * create_part()
  {
    return new realpart_v2;
  }
}
```

...except...
it doesn't compile!

Ambiguity


part
+id(): unsigned
+name(): std::string
part_v2
+id(in value:unsigned)
+name(in value:std::string)
realpart
+id(): unsigned
+name(): std::string
realpart_v2
+id(in value:unsigned)
+name(in value:std::string)

Resolved

```
#pragma once

#include <string>

namespace inventory
{
  class part
  {
  };
  class part_v2 : public virtual part
  {
  };
}
```

```
#include "part.h"

namespace
{
  class realpart : public virtual inventory::part
  {
  };

  class realpart_v2 : public virtual inventory::part_v2, public realpart
  {
  };
}
```

Finishing polish

Names have power

```
namespace inventory
{
  class part
  {
  };
  class part_v2 : public virtual part
  {
  };

  part * create_part();
}
```

The factory returns a part.

Changing this breaks the ABI (ODR).

Clients of part_v2 must now find all references to “part”, and use the factory like this:

```
  unique_ptr< part_v2 > p( dynamic_cast< part_v2* >( create_part() ) );
```

Called by a common name

Class `part` becomes class `part_v1`

```
typedef part_v1 * part;
```

Called by a common name

Class `part` becomes class `part_v1`

```
typedef part_v1 * part;
```

Or, even better....

```
#include "part_v1.h"

#include <memory>

namespace inventory
{
  typedef part_v1 part_current;
  typedef std::unique_ptr< part_current > part;
}
```

No more casts

```
namespace inventory
{
  INVENTORY_LIB part_v1 * create_part_ptr();

  template< typename type_version >
  part create_part()
  {
    return part( static_cast< type_version * >( create_part_ptr() ) );
  }
}
```

```
#include <part_factory.h>

#include <iostream>

int main()
{
  using namespace std;
  using namespace inventory;

  part p = create_part< part_current >();
  cout << p->number() << endl;
  cout << p->name() << endl;
}
```

Version up!

New interface

```
namespace inventory
{
  class INVENTORY_LIB part_v2 : public virtual part_v1
  {
  };
}
```

The real thing

New implementation

```
#include <part_v2.h>

namespace inventory_impl
{
  class realpart_v2 : public virtual inventory::part_v2, public realpart_v1
  {
  };
}
```

Update factory

```
namespace inventory
{
  using namespace inventory_impl;

  INVENTORY_LIB part_v1 * create_part_ptr()
  {
    return new realpart_v2;
  }
}
```

Status quo

Update the typedef

```
namespace inventory
{
  typedef part_v2 part_current;
  typedef std::unique_ptr< part_current > part;
}
```

The template factory stays the same

```
#include <part.h>

namespace inventory
{
  INVENTORY_LIB part_v1 * create_part_ptr();

  template< typename type_version >
  part create_part()
  {
    return part( static_cast< type_version * >( create_part_ptr() ) );
  }
}
```

The client

V1

```
#include <part_factory.h>

#include <iostream>

int main()
{
  using namespace std;
  using namespace inventory;

  part p = create_part< part_current >();
  cout << p->number() << endl;
  cout << p->name() << endl;
}
```

v2

```
#include <part_factory.h>

#include <iostream>

int main()
{
  using namespace std;
  using namespace inventory;

  part p = create_part< part_current >();
  p->number( 100 );
  p->name( "steve" );
  cout << p->number() << endl;
  cout << p->name() << endl;
}
```

Dependency management

Code that has objects passed to it

(i.e. does not need to create objects)

should have no dependency on the factory.

Paramaterize from Above

and all that

Divide and conquer

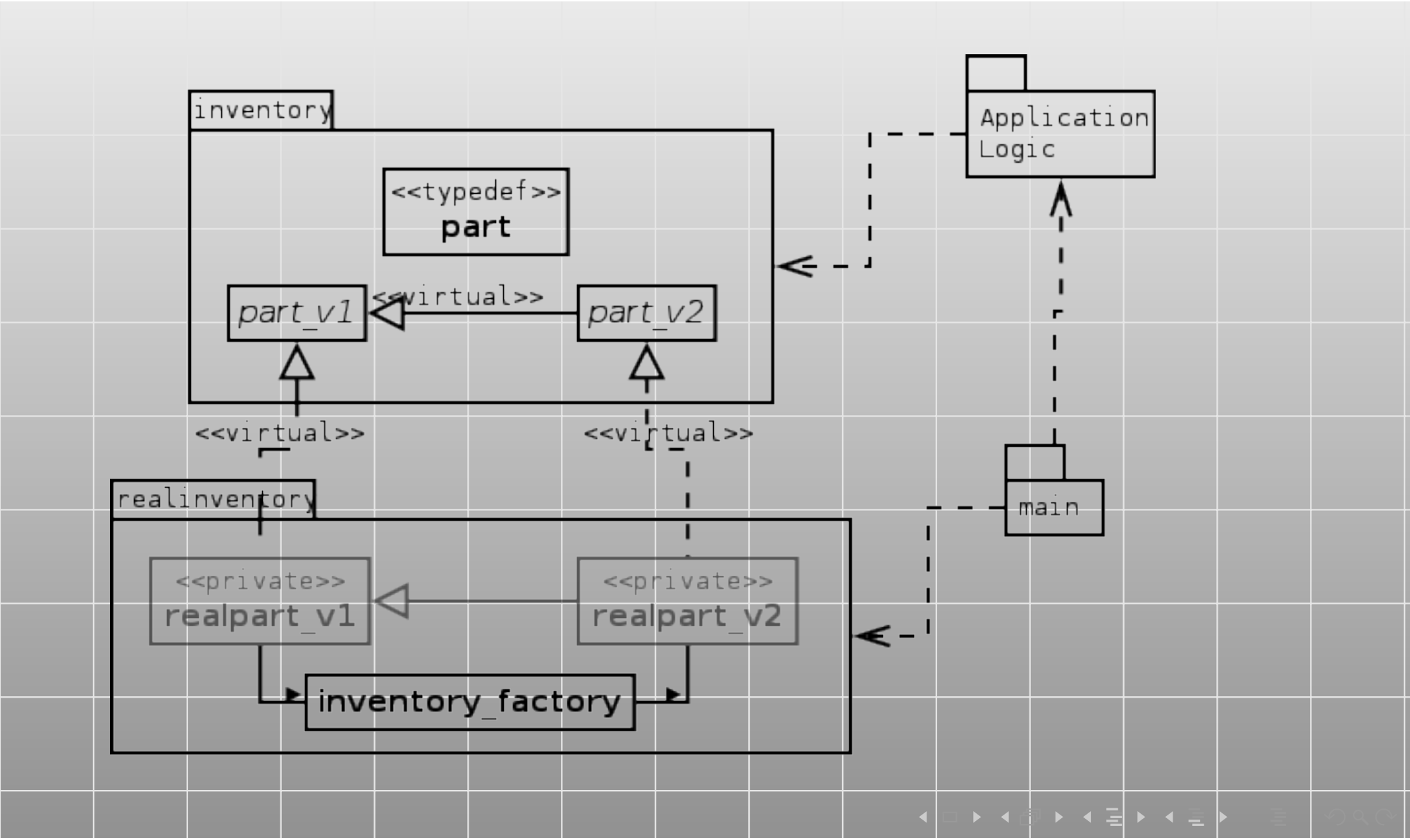
IINVENTORY.DLL

- Headers for `part_v1/v2`
- Implementation file for `part_v1`
- The `part` type and `part_current` typedef

INVENTORY_IMPL.DLL

- Headers and implementation of `realpart`
- The factory

In other words...



Justifying the means

The means

An extensible, safe and conforming way of extending interfaces without causing clients to redeploy.

The end

Loosening the coupling between the library and its clients.