Checked Bridge

Object Structural

Intent

Allow different versions of Bridge interfaces and implementations to work together.

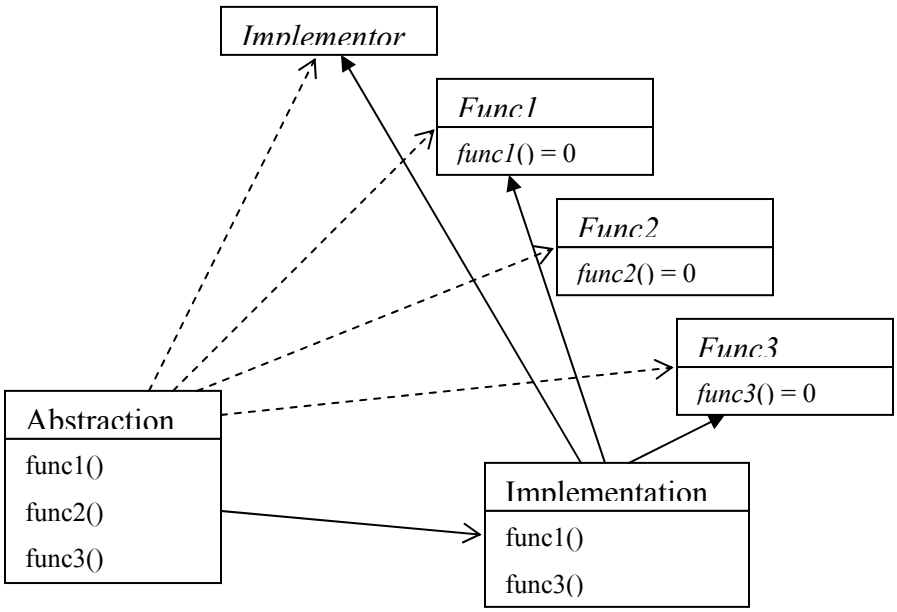
Motivation

When the Bridge pattern is used with different versions of software that are distributed at different times, it is often the case that the interface part of the Bridge may be paired with an implementation part that was developed for an earlier or later version of the interface. The Checked Bridge pattern shows how to export the implementation requirements of an interface to potential implementations, and perform a fine-grain capability query to ensure that a particular aspect of an interface's functionality is supported by its implementation.

Applicability

Use a Checked Bridge when an application may be composed of many components that are "discovered" at runtime, and may have been developed to somewhat different sets of interface requirements. Checked Bridge may also be used to permit safe, planned extension of capability without requiring older components to be replaced.

Structure



Consequences

Checked Bridge is more flexible than a traditional Bridge in that interface and implementation do not have to match up precisely.

Checked Bridge employs a runtime capability query on its implementation, and may therefore be slower than a traditional Bridge implementation.

Checked Bridge permits wide leeway in independent modification of implementations and interfaces over time, but assumes that an interface member function with a particular identifier and argument types will have the same abstract meaning in all interfaces and implementations. For instance, a user of a Checked Bridge could run into trouble if a "draw" function (meaning to render something to the screen) were added to an interface, a separate "draw" function (meaning to brandish a firearm) were added to an implementation, and the two matched at runtime.

Implementation and Sample Code

The interface part of the Checked Bridge (`Abstraction`) is defined in the usual way. The interface refers to its implementation through a pointer to an appropriate implementation (`Implementor`):

```
class Abstraction {
public:
   Abstraction();
   ~Abstraction();
   void f();
   int g(int);
   virtual void h();

   struct F { virtual void f() = 0; };
   struct G { virtual int g(int) = 0; };
   struct H { virtual void h() = 0; };
   struct Implementor { virtual ~Implementor() {} };
protected:
   Implementor *getImpl()
         { return i_; }
private:
   Implementor *i_;
};
```

However, the implementation of a Checked Bridge has a different structure from that of a traditional Bridge. Here, we know only that the implementation is derived from the polymorphic abstract base `Implementor`. The interface class also defines a set of capability interface classes in one-to-one correspondence with the functions that form its public interface. For example, in the code above, the member function declared

```
int g(int);
```

has an accompanying capability class:

```
struct G { virtual int g(int) = 0; };
```

The capability class contains a single pure virtual function with the same signature and return type as its corresponding interface member function; that is, `Abstraction::G::g` has the same signature and return type as `Abstraction::g`.

The set of capability classes defined by the interface specifies a set of capabilities that the interface would like—but not require—its implementation to provide.

An implementation of the interface specifies that it is an implementation of `Abstraction` by derivation from `Abstraction::Implementor`, and precisely what functionality it implements by derivation from the appropriate capability classes.

```
struct Implementation :    // this implementation…
  public Abstraction::Implementor,    // implements Abstraction,
  public Abstraction::F,         // handles f,
  public Abstraction::H,         // handles h, and
  public Abstraction::G {        // handles g
  void f()
        { cout << "called f" << endl; }
  int g( int a )
        { cout << "called g: " << a << endl; return a; }
  void h()
        { cout << "called h" << endl; }
};
```

Unlike a traditional Bridge implementation, the implementations of the interface's member functions are defined with the interface, rather than with the implementation. (Because the implementation is accessed through the `Abstraction::Implementor` interface class, the user of the Checked Bridge is still protected from implementation changes.) The interface-side functions perform a capability query on the implementation to ensure that their functionality is supported before forwarding the call:

```
int Abstraction::g( int a ) {
  G * const gp = dynamic_cast<G *>(i_);     // capability query
  if( !gp )                              // if impl doesn't support
        throw MissingImpl( "g" );        // let user know
  return gp->g( a );                     // else forward to impl
}
```

To see the utility of Checked Bridge, consider the case where the interface and implementation are out of sync. This is a common situation in software that is distributed or updated in pieces. For example, a newer module written to an updated interface may attempt to use a facility not supported by an earlier version of an implementation.

```
struct OldImplementation : // this implementation…
  public Abstraction::Implementor,    // implements Abstraction,
  public Abstraction::F,         // handles f,
  /* NOTE: no H! */              // DOES NOT handle h, and
  public Abstraction::G {        // handles g
  void f()
        { cout << "called f" << endl; }
  int g( int a )
        { cout << "called g: " << a << endl; return a; }
};
```

A user of the Checked Bridge can work with out-of-sync implementations without disastrous results:

```
try {
  Abstraction a;
  a.f();
  a.g(12);
  a.h();
}
catch( MissingImpl &m ) {
  cout << "Called unknown function: " << m.what() << endl;
}
```

Caching Queries

The use of a `dynamic_cast` can be expensive for repeated calls. If the implementation is to remain the same for the life of the interface, the result of the capability query can be cached:

```
int Abstraction::g( int a ) {
  static G * const gp = dynamic_cast<G *>(i_); // cache result
  if( !gp )
        throw MissingImpl( "g" );
  return gp->g( a );
}
```

Thread-safe caching can be attempted with the Double-Checked Locking pattern (although this approach has been declared to be "broken"[1]).

```
void Abstraction::f() {
  static F *fp = 0;
  static volatile bool set = false;
  if( !set ) {
        Mutex m; // get a mutex
        if( !set ) {
              fp = dynamic_cast<F *>(i_);
              set = true;
        }
  }
  if( !fp )
        throw MissingImpl( "f" );
  fp->f();
}
```

[1] *The "Double-Checked Locking is Broken" Declaration*, http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

Nested Classes

The use of nested classes to define the capability classes helps to avoid name conflicts with other interface classes with similar interfaces.

Selecting an Implementation

Typically, we would not expect the user on the interface side of a Checked Bridge to select an implementation directly. Typically, there would be a number of candidate implementations, the most appropriate of which would be selected dynamically.

The capability query interface can also be used to select an implementation that supports a minimal functionality required by the interface's user.

```
inline bool
Abstraction::isAdequate( const Implementor *i ) {
  return            // this Abstraction requires support of
        dynamic_cast<const F*>(i) &&  // f and
        dynamic_cast<const H*>(i);    // h
}
```

Naming Conventions

Use of naming conventions for capability class names is important for clarity. The approach used above that employs nested capability classes with the "same" identifier (up to capitalization or some other trivial transformation) of the function they represent is probably best. It's probably also a good idea to give the pure virtual function within the capability class the same name as the member function it represents.

It's probably simplest to avoid overloaded functions in the interface. However, overloaded functions in the interface may be accommodated through use of a capability class name that encodes the signature of its corresponding member function. (Cf. the traditional use of "name mangling" by C++ compilers to distinguish overloaded function names.)

Default Implementations

The interface class can provide a default implementation of the interface function (rather than simply throwing an exception) in the event the implementation does not support that function.

Use of Derivation

As with the traditional Bridge approach, we can also derive from an interface:

```
class RefinedAbstraction : public Abstraction {
public:
  void i(); // new member function
  void h(); // overrides base virtual
  struct I { virtual void i() = 0; };
  struct H { virtual void h() = 0; };
};
```

Of course, an overriding derived class function like `RefinedAbstraction::h` will expect a corresponding implementation on the implementation side of the Checked Bridge; that is, the implementation of `RefinedAbstraction::h` will not be found (since we're casting to `RefinedAbstraction::H`, not `Abstraction::H`), and you'll get a "not implemented" exception if you call `RefinedAbstraction::h` with the `Implementation` implementation above. It's necessary to supply a different implementation:

```
struct RefinedImplementation :
  public Implementation,
  public RefinedAbstraction::I,
  public RefinedAbstraction::H {
  void i() { cout << "called derived i" << endl; }
  void h() { cout << "called derived h" << endl; }
};
```

Note that an implementation may be derived from an existing implementation, as `RefinedImplementation` has done above.

Related Patterns

The structure of Checked Bridge is basically a Bridge, with a fine grain check on the capability of the implementation.

The use of multiple inheritance of capability interface classes and accompanying capability query is reminiscent of the structure of Acyclic Visitor.