

---

## ITERABLE CONTAINER

---

Containers, C++ Related

### Intent

Provide a common interface to those containers whose elements can be iterated over one at a time.

### Motivation

In C++ it has been common practice to create iterators that are specific for a particular kind of container. For example, `SetIterator` for `Set` containers, and `QueueIterators` for `Queue` containers. However this means that you must know the kind of container you are iterating over before you can create the iterator.

There are many kinds of containers that can be iterated over. Yet iteration itself has little or no relation to the kind of container. It is often desirable to iterate over a container without knowing its kind. For example, I might create a function which prints every element of a container, regardless of whether the container is a `Bag`, `Set`, `Queue`, `Stack`, or what have you. To do this, I need a scheme and an interface for iteration that is independent of the type of a container.

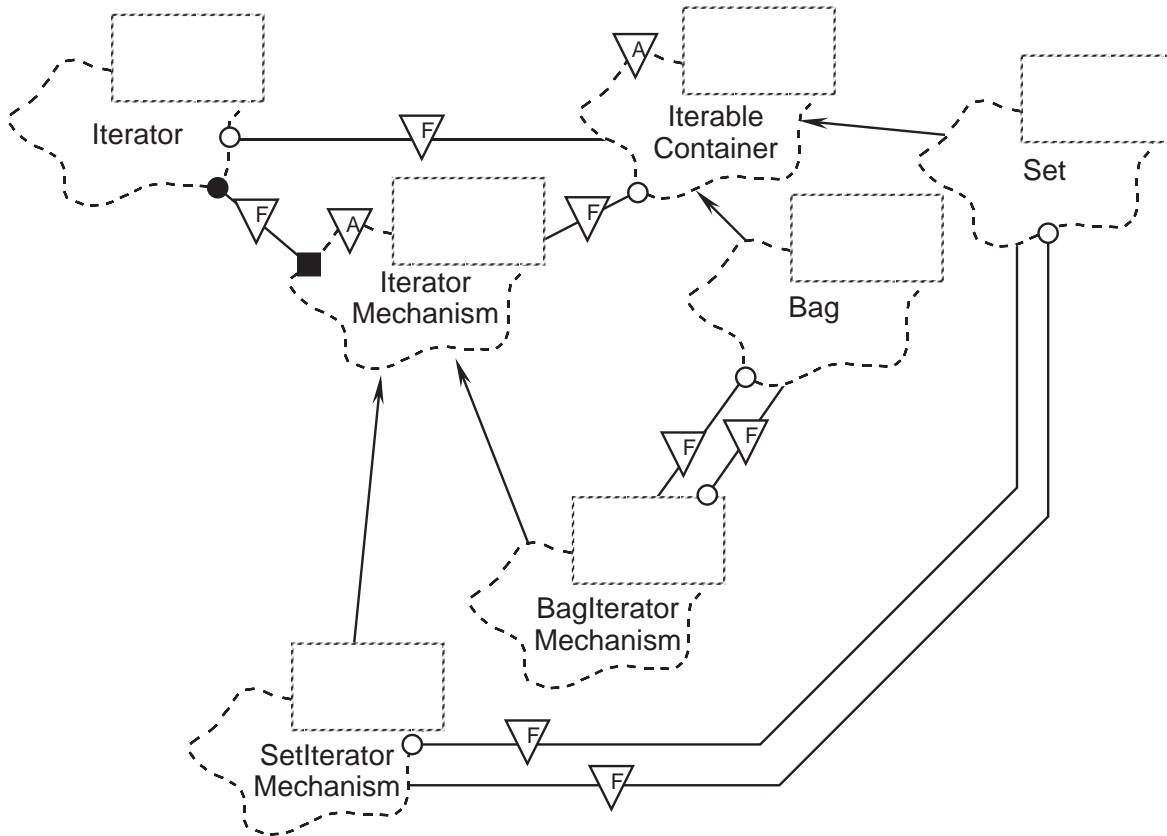
### Solution

We can use the `FACTORY`, `BRIDGE`, and `PROTOTYPE` patterns to create a way for users to use the same concrete `Iterator` object regardless of the container type. All containers are derived from an abstract class named `IterableContainer`. This class provides a pure virtual function named `MakeIteratorMechanism`. Thus `IterableContainer` becomes the `ABSTRACT FACTORY` for specific kinds of `IteratorMechanisms`. Each derived container is the concrete factory that creates the appropriate derivative of `IterableContainer` that knows how to iterate that particular container.

The class `Iterator` is a concrete class which has a constructor that takes a reference to an `IterableContainer`. The constructor invokes the `MakeIteratorMechanism` method and then uses the `BRIDGE` pattern to delegate the iteration functions to the `IteratorMechanism`. `Iterators` provide “copy” semantics by using the contained `IteratorMechanism` as a `PROTOTYPE` and asking it to `Clone` itself.

## Structure

The structure below shows how this pattern can be used with two different containers, Bag and Set. It is trivial to expand its use to other containers as well.



## Implementation

One possible implementation of this Pattern of C++:

```

template <class T> class Iterator;
template <class T> class IteratorMechanism;

//-----
// Name
//  IterableContainer
//
// Description
//  This class is the abstract base class from which all containers
//  that can be linearly iterated derive. It supplies the pure
//  virtual MakeIteratorMechanism function.

```

```

//
template <class T>
class IterableContainer
{
public:
    virtual ~IterableContainer() {};

private:
    virtual IteratorMechanism<T>* MakeIteratorMechanism() const = 0;
    friend Iterator<T>;
};

//-----
// Name
// IteratorMechanism
//
// Description
// This is the abstract base class from which all IteratorMechanisms
// derive. It supplies the pure interfaces by which all
// IteratorMechanisms are controlled.
//
// IsNotAtEnd -- returns true if the iterator has yet reached the
// end of the container. Otherwise returns false.
// Next -- Moves the Iterator to the next item in the
// container. (Undefined if container is at end.)
// Item -- returns the value of the item in the container
// at the position currently referred to by the
// Iterator.
// Clone -- Create an exact copy of the IteratorMechanism
// on the heap, and return its address.
//
template <class T>
class IteratorMechanism
{
private:
    virtual bool IsNotAtEnd() const = 0;
    virtual void Next() = 0;
    virtual T Item() = 0;
    virtual IteratorMechanism<T>* Clone() const = 0;

    IteratorMechanism& operator= (const IteratorMechanism&);
    friend IterableContainer<T>;
    friend Iterator<T>;
};

//-----
// Name
// Iterator
//
// Description
// This class is a concrete class that creates and then manipulates
// IteratorMechanisms. It presents the standard Iterator interface:
//
// operator void* Used to test the iterator for completion.
// This is similar to the error test used in
// iostreams. The void* type is allowed in

```

```

//          conditional expressions. Zero void* values
//          taken to be false, and non-zero void* values
//          are taken to be true. Thus one can test an
//          iterator for completion as follows:
//          Iterator i(someIterableContainer);
//          while (i) // loops till end of container.
//
// operator++      Used to move the iterator to the next item in
//                  the container.
//
// operator*       Used to access the item currently referred to
//                  by the Iterator.
//
// With this interface, the following idiom can be used:
//
//   for(Iterator<T> i(someIterableContainer); i; i++)
//   {
//       T t = *i;
//       // use the item
//   }
//
template <class T>
class Iterator
{
public:
    Iterator(const IterableContainer<T>& theIterableContainer)
    {itsMechanism = theIterableContainer.MakeIteratorMechanism();}

    virtual ~Iterator() {delete itsMechanism;}

    Iterator(const Iterator<T>& theIterator)
    {itsMechanism = theIterator.itsMechanism->Clone();}

    Iterator& operator= (const Iterator<T>& theIterator)
    {
        delete itsMechanism;
        itsMechanism=theIterator.itsMechanism->Clone();
        return *this;
    }

    virtual operator void*() const
    {return reinterpret_cast<void*>(itsMechanism->IsNotAtEnd());}

    virtual void operator++(int) {itsMechanism->Next();}
    virtual T operator*() {return itsMechanism->Item();}

private:
    IteratorMechanism<T> *itsMechanism;
};

```

## Applicability

Use the `IterableContainer` pattern when you have many different kinds of containers, and you need to iterate over each kind of container without depending upon the kind of container. This applies when:

1. You have containers which supply the same containment semantics but use different implementations to tweak efficiency of storage or execution. (e.g. `Sets` that use linked lists, `Sets` that use arrays, `Sets` that use hash lookups, `Sets` that use linear lookups). Clients of these different kinds of containers very likely do not want to know the difference.
2. You have clients that need to iterate over many different kinds of containers, irrespective of their containment semantics. (e.g. clients that need to iterate over the elements of a container, regardless of whether that container is a `Set`, `Queue`, `Stack`, or `Bag`, etc.)

## Consequences

There are slight inefficiencies related to the use of the BRIDGE and FACTORY patterns. Creation of an `Iterator` involves the virtual deployment of the `MakeIteratorMechanism` function, and also the allocation of `IteratorMechanism` on the heap. Special purpose allocators can mitigate this inefficiency quite a bit.

Also, every use of an `Iterator` object is delegated through virtual deployment to its `IteratorMechanism`. While virtual deployment in C++ is very fast, it is not as fast as direct function calls. In certain application domains where efficiency is extremely important, the virtual deployment of functions as common as iterator manipulations may be inappropriate.

## Sample Code

The following is an example of a function which takes a container, regardless of its type, and then prints every element:

```
void PrintElements(IterableContainer<int>& c)
{
    for (Iterator<int> i(c); i; i++)
        cout << *i << endl;
}
```

## Related Patterns

BRIDGE, FACTORY, PROTOTYPE

---

## MEMBERCONTAINER

---

Containers, C++

### Intent

Provide a common interface for containers whose underlying principle is “membership”; so that clients can manipulate those containers without having to be aware of their specific type.

### Motivation

There are many types of containers which have the concept of “membership”. Such containers, such as `Bag` or `Set` allow members to be added, removed and checked for membership.

Usage of these containers is often general, and does not depend upon the specific type of a container. For example, a function that scans a container looking for elements that meet particular criteria and then removes them from the container should not care if the container is a `Set`, `Bag` or `Queue`.

There are a standard set of “membership operators” that should apply to such containers. These operators are *intersection*, *union* and *difference*. These operations can be supplied by overloading the operators `&`, `+` and `-` respectively.

Consider the following code:

```
Bag<int> b;  
Set<int> s;  
Set<int> common = b & s;
```

The intent is clear. We would like to find all the elements that are common to both the `Set` and the `Bag` and place them in another `Set` named ‘common’.

However, what is the return type of the ‘&’ operation? It appears to be calling `b.operator&(s)`. Since `b` is a `Bag`, the operator is probably returning a `Bag`. How can a `Bag` be used to initialize the `Set` named ‘common’. We could create a constructor for `Set` which takes a `Bag` argument; but this promises to end in a geometric explosion of constructors as new types of containers are added. It also means that none of the container classes can be closed against new types of containers.

Worse, what if `Bag` and `Set` are abstract classes. Then what does `b.operator&(s)` return?

## Solution

All containers that have membership semantics derive from an abstract base class named `MemberContainer`. `MemberContainer` uses the `ITERABLECONTAINER` pattern and the `ENVELOPELETTER`<sup>1</sup> pattern to allow clients to operate upon the membership semantics of a container without depending upon the type of the container.

`MemberContainer` inherits from `IterableContainer` so that iteration can be achieved without depending upon the actual type of the container. It adds the simple methods `Add`, `Remove`, `Clear`, `IsMember` and `Cardinality` so that clients can manipulate the membership semantics of any derived containers. `MemberContainer` also implements the membership operators for union, intersection and difference.

The `ENVELOPELETTER` pattern allows `MemberContainer` to be instantiated as a concrete class which delegates all its operations to a contained derivative. The membership operators for union, intersection and difference return such an instantiation of `MemberContainer` by value. The derivative contained by that `MemberContainer` instance is created by using the `PROTOTYPE` pattern on the left hand operand.

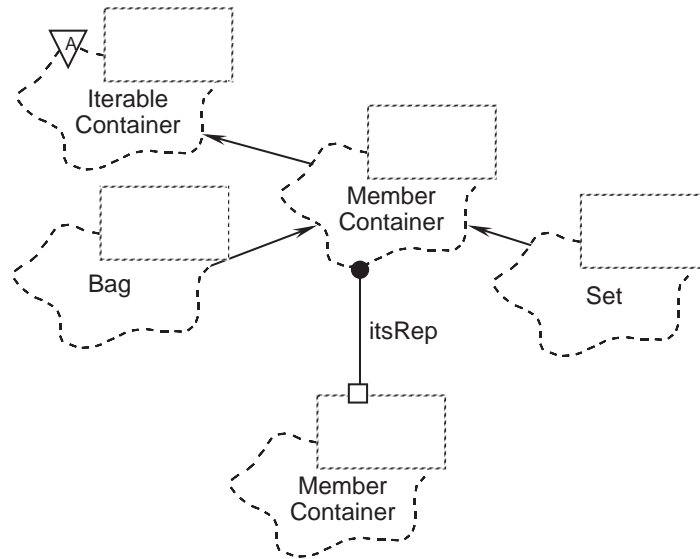
This allows the membership operators for intersection, union and difference to be closed against new kinds of containers. Functions which use these operators do not have to know what type of container they are operating upon.

## Structure

The structure below shows that the `MemberContainer` class is independent of its derivatives. Any derivative of `MemberContainer` can be used as the delegate of the `ENVELOPELETTER` pattern.

---

1. *Advanced C++ Programming Styles and Idioms*, James O. Coplien, Addison Wesley, 1992



## Implementation

A possible C++ implementation of this pattern is as follows.

```

//-----
// Name
// MemberContainer
//
// Description
// Concrete base class for all containers that have membership
// semantics. The class supplies interfaces for all the membership
// manipulations. The default behavior of these interfaces
// delegates to the contained MemberContainer derivative.
//
// MemberContainer is not an abstract class, and so MemberContainer
// objects can be instantiated. However the only constructor
// available is a copy constructor. Thus MemberContainer objects
// can only be constructed from previously existing derivatives of
// MemberContainers.
//
template <class T>
class MemberContainer : public IterableContainer<T>
{
public:
    MemberContainer(const MemberContainer<T>& t);
    // copy constructor. MUST NOT BE CALLED FROM DERIVED
    // COPY CONSTRUCTORS!!! Infinite recursion will be the
    // result....

    virtual ~MemberContainer() {delete itsRep;};
    MemberContainer<T>& operator=(const MemberContainer<T>&);
};
  
```

```

virtual void Add(const T& t)      {itsRep->Add(t);}
virtual bool Remove(const T& t)  {return itsRep->Remove(t);}
virtual void Clear()             {itsRep->Clear();}

virtual bool IsMember(const T& t) const
{return itsRep->IsMember(t);}

virtual int Cardinality() const
{return itsRep->Cardinality();}

virtual MemberContainer<T>* Clone() const
{return new MemberContainer<T>(*this);}

MemberContainer<T>& operator+= (const MemberContainer<T>&);
MemberContainer<T>& operator-= (const MemberContainer<T>&);
MemberContainer<T>& operator&= (const MemberContainer<T>&);

MemberContainer<T> operator+ (const MemberContainer<T>&) const;
MemberContainer<T> operator- (const MemberContainer<T>&) const;
MemberContainer<T> operator& (const MemberContainer<T>&) const;

protected:
    // Default constructor is protected and degenerate.
    MemberContainer() : itsRep(0) {};
    void CopyItems(const MemberContainer&);

private:
    virtual IteratorMechanism<T>* MakeIteratorMechanism() const
    {return itsRep->MakeIteratorMechanism();}

    MemberContainer<T>* itsRep;
};

template <class T>
MemberContainer<T>::MemberContainer(const MemberContainer<T>& t)
: itsRep(0)
{
    if (t.itsRep)
        itsRep = t.itsRep->Clone();
    else
        itsRep = t.Clone();
}

template <class T>
MemberContainer<T>& MemberContainer<T>::operator=(const
MemberContainer<T>& theMemberContainer)
{
    if (this != &theMemberContainer)
    {
        Clear();
        CopyItems(theMemberContainer);
    }
    return *this;
}

template <class T>

```

```

void MemberContainer<T>::CopyItems(const
MemberContainer<T>&
theMemberContainer)
{
    for (Iterator<T> i(theMemberContainer); i; i++)
        Add(*i);
}

template <class T>
MemberContainer<T>& MemberContainer<T>::
operator+= (const MemberContainer<T>& theMemberContainer)
{
    for (Iterator<T> i(theMemberContainer); i; i++)
        Add(*i);
    return *this;
}

template <class T>
MemberContainer<T>& MemberContainer<T>::
operator-= (const MemberContainer<T>& theMemberContainer)
{
    for (Iterator<T> i(theMemberContainer); i; i++)
        Remove(*i);
    return *this;
}

template <class T>
MemberContainer<T>& MemberContainer<T>::
operator&= (const MemberContainer<T>& theMemberContainer)
{
    MemberContainer<T>& difference = *Clone();
    difference -= theMemberContainer;
    operator-=(difference);
    return *this;
}

template <class T>
MemberContainer<T> MemberContainer<T>::
operator+(const MemberContainer<T>& theMemberContainer) const
{
    MemberContainer<T> x = *this;
    x += theMemberContainer;
    return x;
}

template <class T>
MemberContainer<T> MemberContainer<T>::
operator-(const MemberContainer<T>& theMemberContainer) const
{
    MemberContainer<T> x = *this;
    x -= theMemberContainer;
    return x;
}

template <class T>
MemberContainer<T> MemberContainer<T>::
operator&(const MemberContainer<T>& theMemberContainer) const

```

```

{
    MemberContainer<T> x = *this;
    x &= theMemberContainer;
    return x;
}

```

## Applicability

Use this pattern whenever many different kinds of containers must be operated upon using membership operators such as ‘intersection’, ‘union’ or ‘difference’. The containers may be different because they have differing membership semantics like `Bag` and `Set`; or because they maintain certain orderings of the data they contain, like `Queue` or `Stack`; or because they use different implementations to tweak memory or execution efficiency like `BoundedSet` and `UnboundedSet`.

## Consequences

Virtual deployment of the membership primitives such as `Add` and `IsMember` will cause a very slight execution overhead. Deployment through the `Envelope/Letter` for those objects that are returned from the overloaded operators will add yet another layer of virtual deployment and delegation overhead. The letter pointer in the `MemberContainer` base class will add one pointer of storage overhead.

## Sample Code

The following code uses `Sets` and membership operators to implement the sieve of Eratosthenes.

```

main()
{
    // Create a set of all the numbers 2..99
    UnboundedSet<int> numbers;
    for (int i=2; i<100; i++)
        numbers.Add(i);

    // Create the set that has the primes <= sqrt(100)
    UnboundedSet<int> primes;
    primes.Add(2);
    primes.Add(3);
    primes.Add(5);
    primes.Add(7);
    UnboundedSet<int> sieve = numbers;
    for (Iterator<int> p(primes); p; p++)
    {
        UnboundedSet<int> multiples;
        for (int n = *p * 2; n < 100; n += *p)
            multiples.Add(n);
        sieve -= multiples;
    }
}

```

```
    }  
    MemberContainer<int> composites = numbers - sieve;  
    for (Iterator<int> ci(composites); ci; ci++)  
        cout << *ci << ' ';  
}
```

## **Related Patterns**

ENVELOPELETTER, PROTOTYPE, ITERABLECONTAINER

---

## THREE LEVEL FSM

---

C++, Low Level

### Intent

Create Finite State Machines whose behavior is independent of their logic. This allows them to be derivable and extensible.

### Motivation

Finite State Machines are often used to describe the logic that an application uses to convert its incoming events to its resultant behaviors. When the logic and behaviors are intermixed in the same algorithms, they become difficult to change and subject to error.

It is difficult to separate control from logic because they often form a closed loop. The logic of the FSM invokes a behavior which in turn invokes another event in the FSM. Thus even when behavior and control are separated into two classes, as in the OBJECTSFORSTATES pattern, the two classes have source code dependencies upon each other. Thus it is difficult to use the behavior class with a different control class.

### Solution

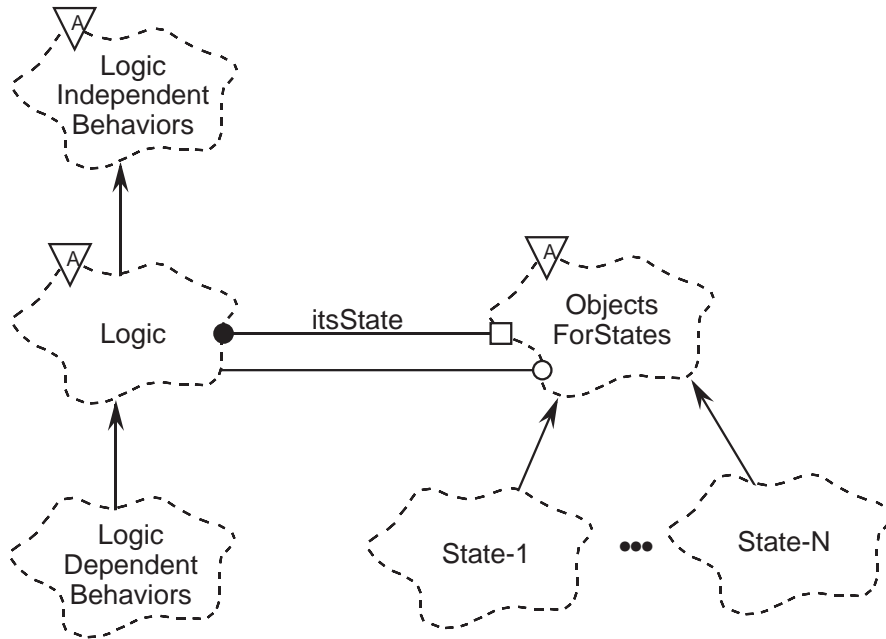
Describe the finite state machine in three layers of inheritance. The first layer supplies interfaces and implementations for the behaviors of the FSM. However, the events of the FSM are not known at this level, so this class is independent of control. This means that any of the behaviors that subsequently invoke events will be implemented as pure virtual functions at this level.

The second layer is derived from the first and adds the functions that respond to the events of the FSM. It also employs the OBJECTSFORSTATES pattern or the STRATEGY pattern to implement the control logic of the FSM. It is possible for this level to be automatically generated from a state table or STD since, except for their names, this layer is independent of the behaviors.

The third layer derives from the second and supplies those behaviors that must invoke subsequent events.

Thus, since the first layer is completely independent of the control mechanisms of the FSM, it can become the base class for derivations that alter or extend the behaviors. It can also be used with different finite state machines by the derivation of another second and third layers.

## Structure



## Implementation

The following code shows a typical implementation of the three levels. The second level has been automatically generated by a finite state machine compiler that employs the OBJECTS-FORSTATE pattern.

The finite state machine for this example is a model of a subway turnstyle. The state table input to the finite state machine compiler is shown below.

### State Table

```

Context TurnStyleLevel1 // the name of the context class
FSMName TurnStyleLevel2 // the name of the FSM to create
Initial Locked // Initial state.
{
  Locked < Lock // Lock upon entry
  {
    Coin   Unlocked  {}
    Pass   *         Alarm
    Failed Broken    LockError // indicate can't lock.
  }
  Unlocked <Unlock
  {
    Coin   *         Thankyou
    Pass   Locked    {}
  }
}

```

```

    Failed  Broken      UnlockError // indicate can't unlock
  }

  Broken <OutOfOrder >InOrder
  {
    Fixed   Locked     {}
  }
}

```

This is a simple state machine. It starts out in the `Locked` state. If a coin is detected, it transitions into the `Unlocked` state. When the turnstyle detects the person “Pass” through, it returns to the `Locked` state. If, while in the `Locked` state, the person forces his way through, then we stay in the `Locked` state, and sound the alarm. If, while in the `Unlocked` state, the person deposits another coin, we light up a little “Thankyou” light.

Whenever we enter the `Locked` state, we invoke the ‘Lock’ action. This action can fail, resulting in a ‘Failed’ event. Likewise, whenever we enter the `Unlocked` state we call the `Unlock` function. This function can fail, resulting in a ‘Failed’ event.

Upon failure, the FSM enters the `Broken` state. Upon entry into this state the `OutOfOrder` function is invoked, lighting a little light warning people away from the turnstyle. When the repairman fixes the turnstyle, the ‘Fixed’ event occurs and the system returns to the `Locked` state. Upon exiting the `Broken` state the system calls the `InOrder` function which turns off the little out of order light.

## Level 1 -- Behaviors Independent of Logic

The first level is a class which defines the interfaces for all of the behaviors of the turnstyle. It also provides implementations for most of those behaviors. Only the `Lock` and `Unlock` functions remain unimplemented because they must invoke the ‘Failed’ event which has not been defined at this level.

```

class TurnStyleLevel1
{
  public:
    virtual void Lock() = 0;
    virtual void Unlock() = 0;

    virtual void Alarm();
    virtual void LockError();
    virtual void UnlockError();
    virtual void Thankyou();
    virtual void OutOfOrder();
    virtual void InOrder();
  protected:
    bool LockAndCheck();
    bool UnlockAndCheck();
};

```

The last two member functions provide implementations for the Lock and Unlock functions. However the 'bool' that these functions return must be translated to an event for the finite state machine. This translation will occur at level 3.

## **Level 2 -- The Control Logic**

This level has been entirely generated by the finite state machine compiler from the text shown above in the state table.

```

class TurnStyleLevel2;
class TurnStyleLevel2State
{
public:
    virtual const char* StateName() const = 0;
    virtual void Coin(TurnStyleLevel2&);
    virtual void Pass(TurnStyleLevel2&);
    virtual void Failed(TurnStyleLevel2&);
    virtual void Fixed(TurnStyleLevel2&);
};
class TurnStyleLevel2BrokenState : public TurnStyleLevel2State
{
public:
    virtual const char* StateName() const
        {return "Broken";}
    virtual void Fixed(TurnStyleLevel2&);
};
class TurnStyleLevel2UnlockedState : public TurnStyleLevel2State
{
public:
    virtual const char* StateName() const
        {return "Unlocked";}
    virtual void Coin(TurnStyleLevel2&);
    virtual void Pass(TurnStyleLevel2&);
    virtual void Failed(TurnStyleLevel2&);
};
class TurnStyleLevel2LockedState : public TurnStyleLevel2State
{
public:
    virtual const char* StateName() const
        {return "Locked";}
    virtual void Coin(TurnStyleLevel2&);
    virtual void Pass(TurnStyleLevel2&);
    virtual void Failed(TurnStyleLevel2&);
};
class TurnStyleLevel2: public TurnStyleLevel1
{
public:
    // Static State Variables
    static TurnStyleLevel2BrokenState Broken;
    static TurnStyleLevel2UnlockedState Unlocked;
    static TurnStyleLevel2LockedState Locked;
    TurnStyleLevel2(); // Default Constructor

```

```

// Event functions
void Coin() {itsState->Coin(*this);}
void Pass() {itsState->Pass(*this);}
void Failed() {itsState->Failed(*this);}
void Fixed() {itsState->Fixed(*this);}
// State Accessor Functions
void SetState(TurnStyleLevel2State& theState)
{itsState = &theState;}
TurnStyleLevel2State& GetState() const {return *itsState;}
private:
    TurnStyleLevel2State* itsState;
};
TurnStyleLevel2BrokenState TurnStyleLevel2::Broken;
TurnStyleLevel2UnlockedState TurnStyleLevel2::Unlocked;
TurnStyleLevel2LockedState TurnStyleLevel2::Locked;
void TurnStyleLevel2State::Coin(TurnStyleLevel2& s) {}
void TurnStyleLevel2State::Pass(TurnStyleLevel2& s) {}
void TurnStyleLevel2State::Failed(TurnStyleLevel2& s) {}
void TurnStyleLevel2State::Fixed(TurnStyleLevel2& s) {}
void TurnStyleLevel2BrokenState::Fixed(TurnStyleLevel2& s)
{
    s.SetState(TurnStyleLevel2::Locked);
    s.InOrder();
    s.Lock();
}
void TurnStyleLevel2UnlockedState::Coin(TurnStyleLevel2& s)
{
    s.Thankyou();
}
void TurnStyleLevel2UnlockedState::Pass(TurnStyleLevel2& s)
{
    s.SetState(TurnStyleLevel2::Locked);
    s.Lock();
}
void TurnStyleLevel2UnlockedState::Failed(TurnStyleLevel2& s)
{
    s.UnlockError();
    s.SetState(TurnStyleLevel2::Broken);
    s.OutOfOrder();
}
void TurnStyleLevel2LockedState::Coin(TurnStyleLevel2& s)
{
    s.SetState(TurnStyleLevel2::Unlocked);
    s.Unlock();
}
void TurnStyleLevel2LockedState::Pass(TurnStyleLevel2& s)
{
    s.Alarm();
}
void TurnStyleLevel2LockedState::Failed(TurnStyleLevel2& s)
{
    s.LockError();
    s.SetState(TurnStyleLevel2::Broken);
    s.OutOfOrder();
}
TurnStyleLevel2::TurnStyleLevel2() : itsState(&Locked)
{

```

```
        Lock();  
    }
```

### **Level 3 -- Behaviors Dependent upon Control**

Finally, level 3 derives from level 2 and implement those behaviors that must invoke events in the FSM. These functions are `Lock` and `Unlock` which must call the `LockAndCheck` and `UnlockAndCheck` functions respectively, and then invoke the 'Failed' event if the functions return false.

```
class TurnStyleLevel3 : public TurnStyleLevel2  
{  
    public:  
        virtual void Lock()  
        {  
            if (!LockAndCheck()) Failed();  
        }  
        virtual void Unlock()  
        {  
            if (!UnlockAndCheck()) Failed();  
        }  
};
```

Since the first level provides all of the substantial behaviors that the finite state machine controls, but in no way depends upon any particular state machine, it can be reused by many state machines. It can also be the target of derivation so that its behaviors can be overridden or extended.

### **Applicability**

Use this pattern in any context where behaviors may be controlled by more than one finite state machine, or where such behaviors need to be overridden and/or extended through inheritance.

### **Consequences**

Virtual deployment of the action functions may add a small amount of execution time overhead. Also, the `OBJECTSFORSTATES` pattern used in level 2 adds lots of classes. Fortunately, since those classes can be generated by a compiler, they do not have to add to the programmers conceptual burden.

## **Related Patterns**

OBJECTSFORSTATES, STRATEGY

---

## ABSTRACT CLIENT

---

C++, High Level

### Intent

To provide clients with an interface which servers understand.

### Motivation

Server classes sometimes need to send messages to their clients. However, in C++, this means that the server must know the interface of the client. This makes it difficult to create servers with many different, unrelated, clients.

We would like the clients of a server to be unrelated so that new clients can be created without affecting either the server or the other clients.

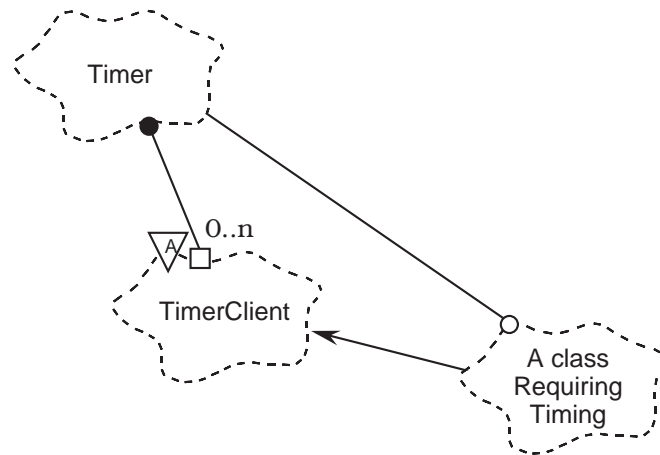
### Solution

Servers that need to send messages to their clients provide the client with an abstract base class containing pure virtual functions that represent those messages. The client can then derive from this abstract base, often using multiple inheritance, and implement the pure virtual functions to handle the messages.

For example, consider a `Timer` class whose job it is to send a 'Tick' message to a group of clients once every second. Each client sends a message to the `Timer` registering itself to receive the `Tick` message.

### Structure

The figure below shows the "Timer" example. Notice that the `Timer` class does not depend upon the derived client in any way. Thus if that client changes, the `Timer` class will be unaffected.



## Implementation

Here is one way that the Timer class could be implemented.

```
class TimerClient
{
    public:
        virtual void Tick() = 0;
};

class Timer
{
    public:
        void Register(TimerClient& t) {itsClients.Add(&t);}

    private:
        void SendTick()
        {
            for (Iterator<TimerClient*>i(itsClients); i; i++)
                (*i)->Tick();
        }

        UnboundedSet<TimerClient*> itsClients;
};
```

## Applicability

Use this pattern wherever there are servers that must send messages to their clients, and you don't want the server to have source code dependencies upon the client.

## **Consequences**

The virtual deployment of the messages through the abstract base class will add a small amount of overhead. Use of this pattern will also promote use of multiple inheritance since the ability to be called by a server will probably need to be mixed-in to an already existing hierarchy.

## **Related Patterns**

BRIDGE -- The abstract class represents a special case of a BRIDGE from the server to the client.

---

## WRITE A LOOP ONCE

---

C++, Low Level

### Intent

To prevent very common loops from being written over and over again.

### Motivation

An application will often need to loop through a container in many different parts of the code. Each loop is virtually identical, except for its body. Moreover, the body of each of these loops may contain similar components. Yet if these loops are all coded separately, then when changes occur to the code that affect the way these loops are written, then each loop must be found and corrected.

It ought to be possible to write such loops just once, and then plug in different loop bodies where applicable. Thus, when changes occur to the structure of the data or the looping mechanism, those changes can be made in one place.

For example, consider an application which models the floorplan of a building in order to determine the quality of the design. Such an application has an extensive data model describing rooms, walls, doors, corridors, windows, etc. The application must score the structure of the floorplan against a battery of design rules. Each rule demands that we traverse the data looking for various components and applying certain scoring algorithms. So each rule must use a variation of the exact same loop, changing only the criteria by which certain elements are selected for processing, and the processing itself.

### Solution

Write an abstract base class which has a function that implements the loop and then calls pure virtual functions for selection and processing. Derived classes can then implement their own selection mechanisms and processing algorithms.

### Implementation

Consider the following example which provides a loop which scans for Doors.

```
class DoorScanner
{
public:
    DoorScanner(IterableContainer<Door*>& c) : itsContainer(c) {}
    virtual bool SelectDoor(Door&) = 0;
```

```

    virtual void ProcessDoor(Door&) = 0;
    void Scan();

private:
    IterableContainer<Door*>& itsContainer;
};

void DoorScanner::Scan()
{
    for (Iterator<Door*> di(itsContainer); di; di++)
    {
        Door& d = **di;
        if (SelectDoor(d))
            ProcessDoor(d);
    }
}

class OpenDoorScanner : public DoorScanner
{
public:
    OpenDoorScanner(IterableContainer<Door*>& c)
        : DoorScanner(c) {}

    bool SelectDoor(Door& d) {return d.IsOpen();}
};

class OpenDoorCounter : public OpenDoorScanner
{
public:
    OpenDoorCounter(IterableContainer<Door*> c)
        : OpenDoorScanner(c), itsOpenDoors(0) {}

    virtual void ProcessDoor(Door& d) {itsOpenDoors++;}
    int GetOpenDoors() const {return itsOpenDoors;}

private:
    int itsOpenDoors;
};

void f()
{
    OpenDoorCounter c(DoorList);
    c.Scan();
    int openDoors = c.GetOpenDoors();
}

```

## Applicability

Use this pattern wherever you have to write a variation of a loop many times.

## Consequences

Virtual deployment of the selection criteria and the processing function will add a small execution overhead.

## **Related Patterns**

BRIDGE -- The selection function and the Processing function are applications of the BRIDGE pattern in a very specific context.