

Layering a Component A Detailed Example Using Queue

Part 1 – Public Interface

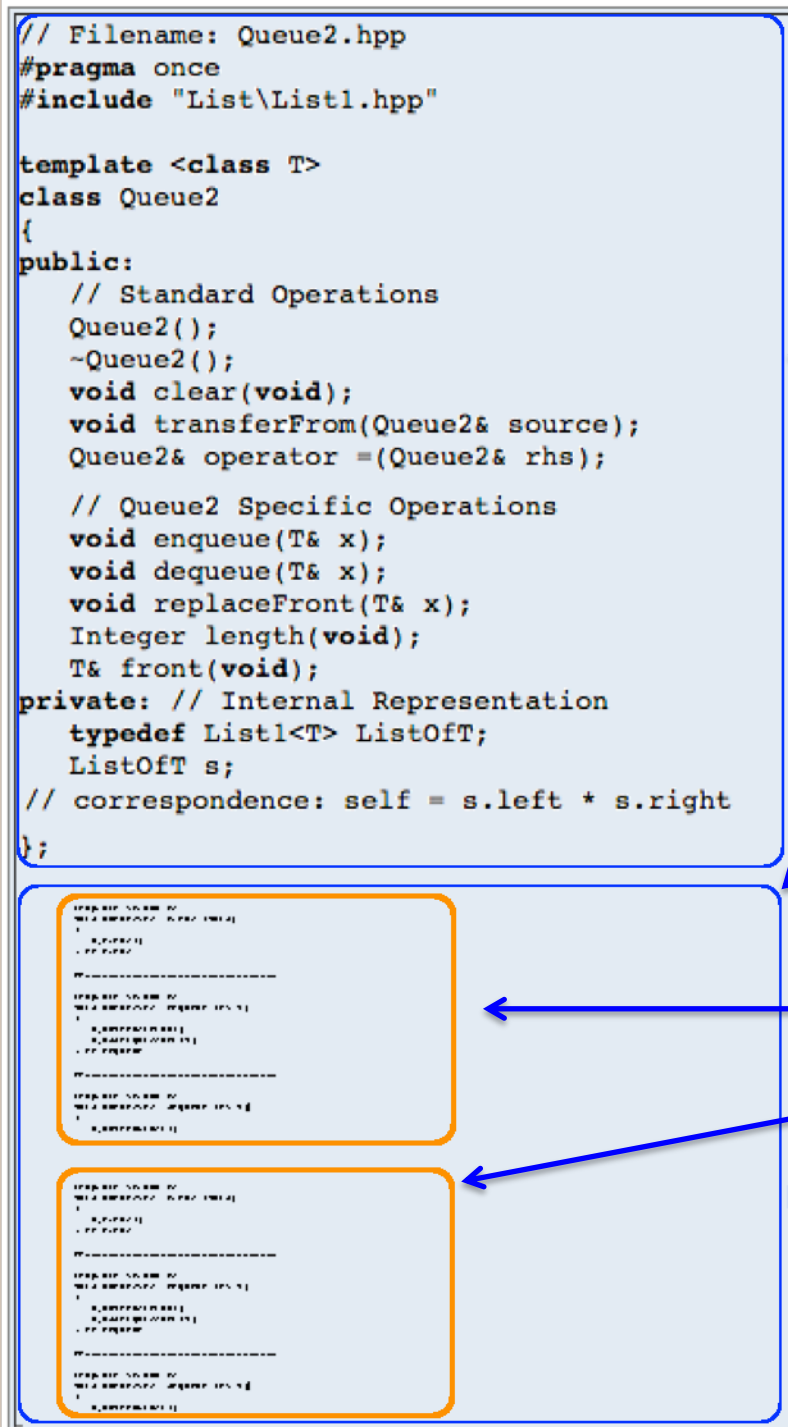
Component Layout 50,000 Foot View

Two major parts make up a component's .hpp file:

1. The public interface part – utilized by the client programmer
2. The member function implementation part – contains implementations for the component's public and private member functions

Two parts to the member function implementation part:

1. Implementations for all the *Standard Operations*
2. Implementations for all the *Specific Operations*



The Top Part of the File


```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The filename:

- Use the component name for the filename
- Append a number to the end of the component name to distinguish it from other implementations of the same component
- Use the filename extension .hpp for a template class component in C++

The Top Part of the File



```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The Top Part of the File

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"
```



The preprocessor directives:

#pragma once

- Tells the C preprocessor to only include this file one time into the the target .cpp file that is being compiled
- This eliminates *redefinition* compiler errors
- It is a non-standard but widely supported preprocessor directive
- If this directive is not available, then use `#ifdef .. #endif` directives

```
template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The Top Part of the File

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"
```

← The preprocessor directives:

#include

- Include the other component (or components) upon which this current component is going to be layered
- For this example we are going to layer Queue on top of a List component

```
template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The Top Part of the File

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"
```

```
template <class T>
class Queue2
```

```
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The component definition begins with:

- Template parameters
- Component's name

The Top Part of the File

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
    /*! is modeled by string of T
    /*! exemplar self

{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The component's abstraction introduces the *mathematical model* used for reasoning abstractly about variables declared from the component

The Top Part of the File

The abstraction also includes the *requires* & *ensures* clauses for each exported operation

dequeue's external contract is shown here

external contract:

- consists of the *requires* and *ensures* clause for an operation
- parameter modes for each formal parameter
- specifies the behavior required of the calling client (in the *requires* clause) and the service provided by the called operation (*ensures* clause)

For example:

- *dequeue*'s *requires* clause dictates that the client must call *dequeue* when the non-empty queue
- the *ensures* clause dictates that the front item on the queue will be removed and produced back to the caller through parameter *x*

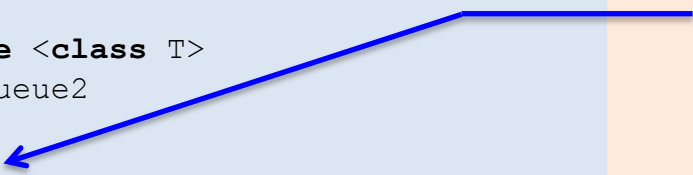
```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
        //! updates self
        //! replaces x
        //! requires: self != <>
        //! ensures: <x> is prefix of #self and
        //! self = #self[1, |#self|)
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The Exported Operations

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
    public: // Standard Operations
        Queue2();
        ~Queue2();
        void clear (void);
        void transferFrom (Queue2& source);
        Queue2& operator = (Queue2& rhs);
    // Queue2 Specific Operations
        void enqueue (T& x);
        void dequeue (T& x);
        void replacefront (T& x);
        T& front (void);
        Integer length (void);
    private: // representation
        // ...
};
```



public:

- This keyword tells the compiler that the member functions that follow can be called by the client program
- These are the *exported member functions* – i.e., those operations that are callable by a client program
- We will often have *internal member functions*, which will be declared in the private part of the component – these are called by the exported member functions as helpers

The Exported Operations

member functions:

- *Member function* is the name used in C++ to refer to the operations that are members of a class or a template class
- In this example there are 5 *standard* member functions and 5 *Queue specific* member functions
- In Java, these operations are referred to as *methods*

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The Exported Operations


member functions – two types:

1. *procedure* – is a member function that has a **void** return type, i.e., does not return a value to the caller
For example, *enqueue*
1. *function* – is a member function that has a non-**void** return type, and does return a value to the caller
For example, *length* has a return type of Integer

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
    // Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);

private: // representation
    // ...
};
```



The Exported Operations

There are two parts to the component's exported operations

- The Standard Operations
- The Component Specific Operations

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
    public: // Standard Operations
        Queue2();
        ~Queue2();
        void clear (void);
        void transferFrom (Queue2& source);
        Queue2& operator = (Queue2& rhs);

        // Queue2 Specific Operations
        void enqueue (T& x);
        void dequeue (T& x);
        void replacefront (T& x);
        T& front (void);
        Integer length (void);

    private: // representation
        // ...
};
```

The Exported Operations

There are *5 standard operations* exported by *all* of the components that we will be using

1. constructor
2. destructor
3. clear
4. transferFrom
5. operator =

This *homogenous look and feel* allows one component to more easily be used by other components and is a very important software engineering design technique

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);

// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);
private: // representation
    // ...
};
```

The Exported Operations

Component Specific Operations

Typically there are two types of operations:

1. Operations that permit the client to *update* the value of a variable
2. Operations that permit the client client to *inspect* various aspects of a variable

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);

    // Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);

private: // representation
    // ...
};
```

The Exported Operations

Operations for **updating** a Queue variable

- enqueue
- dequeue
- replaceFront

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    ...

    // Queue2 Specific Operations

    void enqueue (T& x);
        //! updates self
        //! clears x
        //! ensures: self = #self * <#x>

    void dequeue (T& x);
        //! updates self
        //! replaces x
        //! requires: self /= <>
        //! ensures: <x> is prefix of #self
        //! and self = #self[1, |#self|)

    void replacefront (T& x);
        //! updates self, x
        //! requires: self /= <>
        //! ensures: <x> is prefix of #self
        //! and self = <#x> * #self[1, |#self|)

    T& front (void);

    Integer length (void);

private: // representation
    // ...

};
```


The Exported Operations

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    ...

    // Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
```

```
T& front (void);
    //! restores self
    //! requires: self != <>
    //! ensures: <front> is prefix of self

    Integer length (void);
    //! restores self
    //! ensures: length = |self|
```

```
private: // representation
    // ...

};
```

Operations for inspecting a Queue variable

- length
- front
- Note: Inspecting does not change the abstract value of the variable, i.e., the queue's value is **restored**
- The word *restores* was chosen to indicate that:
 - the parameter can be temporarily changed during the called operation's execution
 - but must have its outgoing value = to its original incoming value
 - Here: self = #self

Being a Container Component

In order to qualify as a *container component*, the component must export two specific operations for altering a variable:

1. For *inserting* a value into a container variable
2. For *removing* a value from a container variable

For Queue:

```
void enqueue (T& x);  
    ///! updates self  
    ///! clears x  
    ///! ensures: self = #self * <#x>
```

```
void dequeue (T& x);  
    ///! updates self  
    ///! replaces x  
    ///! requires: self /= <>  
    ///! ensures: <x> is prefix of #self  
    ///! and self = #self[1, |#self|)
```

```
void replacefront (T& x);
```

```
T& front (void);
```

```
Integer length (void);
```

```
private: // representation
```

```
    // ...
```

```
};
```

- enqueue – for inserting
- dequeue – for removing

The Internal Representation

In the next set of slides, we dig into the Internal Representation which appears in the component's private part

```
// Filename: Queue2.hpp
#pragma once
#include "List/List1.hpp"

template <class T>
class Queue2
{
public: // Standard Operations
    Queue2();
    ~Queue2();
    void clear (void);
    void transferFrom (Queue2& source);
    Queue2& operator = (Queue2& rhs);
// Queue2 Specific Operations
    void enqueue (T& x);
    void dequeue (T& x);
    void replacefront (T& x);
    T& front (void);
    Integer length (void);

private: // representation
    typedef List1<T> ListOfT;
    ListOfT s;

};
```

