Basics of Checking Component Construction

Overview of a Checking Component

```
#prgama once
// Filename: ComponentChecking.hpp

template <class Component>
class ComponentChecking: public Component
{
public:

   // Overriding operations appear here


} // end ComponentChecking
```

Only 1 Checking Component Required

```
#prgama once
// Filename: ~~Component~~Queue Checking.hpp

template <class ~~Component~~Queue>
class ~~Component~~Queue Checking: public ~~Component~~Queue
{
public:

   // Overriding operations appear here


} // end ~~Component~~Queue Checking
```

- One checking component works for *all* implementations of a *Component*
- For example, suppose Queue0, Queue1, Queue2, etc., are different implementations of the Queue concept
- All these Queue*X* versions have the same public interface and external contract
- So, one **QueueChecking** can check each of these different implementations

Checking Component Filename

```
#prgama once
// Filename: ComponentChecking.hpp

template <class Component>
class ComponentChecking: public Component
{
public:

   // Overriding operations appear here


} // end ComponentChecking
```

- Place the checking component in its own .hpp file
- Convention:
  - Name the file after the component that it checks
  - Add ‘Checking’ to the component’s name
  - Example: *Component*Checking.hpp

Checked Component – Template Parameter

```
#prgama once
// Filename: ComponentChecking.hpp

template <class Component>
class ComponentChecking: public Component
{
public:

   // Overriding operations appear here


} // end ComponentChecking
```

- The component to be checked must be supplied as a template parameter

Checked Component – Inheritance

```
#prgama once
// Filename: ComponentChecking.hpp

template <class Component>
class ComponentChecking: public Component
{
public:

   // Overriding operations appear here


} // end ComponentChecking
```

- The checking component itself inherits from the template parameter
- That is, the checking version inherits from the *unchecked* component that is to be checked
- This is C++'s syntax for making:
  - Component a superclass
  - ComponentChecking a subclass

```
#prgama once
// Filename: ComponentChecking.hpp

template <class Component>
class ComponentChecking: public Component
{
public:
  // Overriding operations appear here


} // end ComponentChecking
```

Checked Component – Overriding

- Override only those operations that have non-trivial requires clause
- For example, with a Queue:
  - *dequeue* – has a non-trivial requires clause requiring that the controlling queue variable not be empty
  - *length* – has a trivial requires clause because it can be called with any queue variable
  - So *dequeue* would be overridden and *length* would not

Overriding – Technicalities Part 1

The overriding operation must have the *exact same signature* as the operation being overridden

- For example, here's the signature for Queue's *dequeue* operation

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

```
void dequeue (T& x)
{
   if (Queue::length() <= 0) {
      OutputDebugString (L"Operation: dequeue\n");
      OutputDebugString (L"Assertion failed: |q| > 0\n");
      DebugBreak ();
   } // end if

   Queue::dequeue (x);
} // dequeue
```

Overriding – Technicalities Part 2

To make a call-through to operations from the unchecked **Component**, use the C++ syntax: **Component::**

- For example, here is a call Queue's *length* operation

```
if (Queue::length() == 0) { … }
```

- And here is a call-through to Queue's *dequeue*

```
Queue::dequeue(x);
```

```
void dequeue (T& x)
{
   if (Queue::length() == 0) {
      OutputDebugString (L"Operation: dequeue\n");
      OutputDebugString (L"Assertion failed: |q| > 0\n");
      DebugBreak ();
   } // end if

   Queue::dequeue(x);
} // dequeue
```

Overriding – Technicalities Part 3

When a precondition violation is detected, output an error message stating what assertion was violated:

- In Visual Studio (Windows and C++) use *OutputDebugString()*

```
void dequeue (T& x)
{
   if (Queue::length() <= 0) {
      OutputDebugString (L"Operation: dequeue\n");
      OutputDebugString (L"Assertion failed: |q| > 0\n");
      DebugBreak ();
   } // end if

   Queue::dequeue (x);
} // dequeue
```

Overriding – Technicalities Part 4

Then cause the program to break

- In Visual Studio (Windows and C++) call *DebugBreak()* which will transfer control to the VisualStudio debugger
- This is different from raising an exception because once in the debugger, the call stack will still be intact and can be traversed to see where things might have gone wrong

```
void dequeue (T& x)
{
   if (Queue::length() <= 0) {
      OutputDebugString (L"Operation: dequeue\n");
      OutputDebugString (L"Assertion failed: |q| > 0\n");
      DebugBreak ();
   } // end if

   Queue::dequeue (x);
} // dequeue
```