Debugging an Operation
A Formal Methods Approach

Part 1 – A Standalone Operation

Iterative

5 Places to Hunt for Defects

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.dequeue(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

Example Operation: *appendV1*

- Standalone operation, i.e., it is not a member of a class
- Uses iteration
- Makes calls to other operations

- Take a few moments to convince yourself this implementation meets its spec, i.e., is correct

5 Places to Hunt for Defects

Assume:

- The operation's specs are correct
- But the operation fails under test

Claim about the debugging process:

- There is a systematic approach (based on design-by-contract ideas) that can be taken when searching for a defect

- This approach provides at least 5 locations to inspect when hunting for a defect

5 Places to Hunt for Defects

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.dequeue(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

- Work with your neighbor(s)
- Try to identify the 5 locations where defects can pop up
- *Important*: Each location should somehow be related to how the code is tied to the spec (or at least *supposed* to be tied to the spec)
- *Remember*: The specs of called operations are also involved

- *Again*: There are no defects in this implementation
- So don't look for actual defects

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() >= 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.dequeue(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

#1 – Blows a Precondition

The calling operation:

- appendV1 is a calling operation
- It fails as a client
- It does not always meet a called operation's precondition

In this example:

- dequeue's *requires* clause is violated

Why?

- Off-by-one error caused by incorrect loop exit condition
- *Note*: a defect was introduced on this slide to aid in seeing how a *requires* clause might be violated

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.replaceFront(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

#2 – Misunderstands a Postcondition

The developer of the calling operation:

- Misunderstands, expects, or assumes that a called operation does something different than what call operation's *ensures* clause guarantees

In this example:

- The developer used *replaceFront* thinking it worked similar to *dequeue*

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 1) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.dequeue(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

#3 – Fails to Satisfy Own *ensures*

The operation’s code does not meet its own *ensures* clause

In this example:

- There is an off-by-one error in the loop exit condition

#4 – Fails to Maintain Loop Invariant

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y, z;

      g.dequeue(y);

      r.enqueue(z);

   } // end while

} // appendV1
```

The operation's loop invariant does not hold:

1. either on first encounter
2. or at bottom of the loop body

In this example:

- The loop body is defective so at the bottom of the loop, the loop invariant does not hold

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.replaceFront(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

#5 – Loop Progress Problems

The operation's decreases clause does not hold

In this example:

- The call to *replaceFront* will not cause the queue's length to decrease on each pass through the loop body
- So the loop's exit condition will not be reached

5 Places to Hunt for Defects

```
void appendV1 (QueueOfT& r, QueueOfT& g){
//! updates r
//! clears g
//! ensures r = #r * #g

   while(g.length() > 0) {
      //! updates g, r
      //! maintains
      //! r * g = #r * #g
      //! decreases |g|

      T y;

      g.dequeue(y);

      r.enqueue(y);

   } // end while

} // appendV1
```

Summary:

1. Blows a Precondition
2. Developer Misunderstands a Postcondition
3. Fails to Satisfy Own *ensures*
4. Fails to Maintain Loop Invariant
5. Loop Progress Problems