Introduction to Software Engineering

Jason Coposky – Chief Technologist, iRODS Consortium

Rough Sketch

- Writing good software is hard
- Break down problems (and software) into simpler fragments
- Capture solutions to these simple problem fragments in code -- components
- Examples of Components and Reuse
- Examples of Functions
- Writing Good Functions - Best Practices
- Breaking it down

Writing Good Software is Hard

- Real world problems need to be captured in a way a computer can understand
- Interesting problems require considerable amounts of code
- There is never enough time to do it right the first time
- No one has time to actually learn -- we're too busy getting real work done
- Every real world problem is comprised of many other problems we have yet to discover
- One thing changes and the code breaks in some other random place

So we’re Doomed, now what?

- Perhaps, but... Problem Solving!
- Software is problem solving, but with more typing
- Divide and Conquer - break hard problems down into more easily conquered pieces
- Write them down! solutions to these small parts of the problem can be captured in code
- What does this have to do with software? Components!

Aside: How does the Software Industry do it?

- Focus on the Process - wrap machinery around writing software
  - Waterfall
  - Spiral
  - Agile flavors - XP, Scrum, Lean
- Tools - Revision Control, Static Analysis, Runtime Analysis, Testing Frameworks
- Automation - Continuous Integration, Continuous Deployment

Components == Reusable Software

- Provide a single and well defined piece of functionality
- Defines an interface that the component uses to interact with the outside world
- Take a certain number and flavor of inputs
- Provide a certain number and flavor of outputs
- Can be more easily tested for correct behavior
- Can be well documented for later reference (doxygen!)
- Can be shared with colleagues and the rest of the community (github!)
- Can be combined in new and interesting ways to solve larger problems

Types of Reusable Software

- Functions
- Classes
- Libraries
- Frameworks
- Domain Specific Languages

Examples: A Function in R

```
getPercent <- function( value, pct ) {
    # add error checking here
    result <- value * ( pct / 100 )
    return( result )
}

# calling the function
result <- getPercent( 10, 110 )
```

Examples: A Function in Python

```
def getPercent( value, pct ):
    # add error checking here
    result = value * ( pct / 100 )
    return result

# calling the function
result = getPercent( 10, 110 )
```

Examples: A Function in C

```
float getPercent( float value, float pct ) {
    /* add error checking here */
    float result = value * ( pct / 100 );
    return result;
}

/* calling the function */
float result = getPercent( 10, 110 );
```

Writing Good Functions: “Best Practices”

- KISS - Keep It Simple : Functional Cohesion (does one thing only)
- DRY - Don't Repeat Yourself : Single Source of Truth (no duplicate code)
- Function name should tell you exactly what the function does
- Validate Inputs - Garbage In, Garbage Out
- Well defined inputs and outputs - do not overload function parameters or returns
- Single return statement per function - this can be a point contention

A Note on Conditionals

- Blocks of decision logic - how we make choices in code
- Operates on statements which are Boolean : True or False
- Excellent choice for determining if inputs are valid
- Use well defined return codes for each flavor of error

Examples : Conditionals in R

```
if( a > b ) {
    # Note: this is what we mean by nesting logic
    if( c > d ) {
        # take some action
    }
} else if ( a == b ) {
    # take some other action
} else {
    return( INVALID_PARAMETER )
}
```

Examples: Conditionals in Python

```
if a > b:
    # Note: this is what we mean by nesting logic
    if c > d:
        # take some action
elif a == b:
    # take a different action
else:
    return INVALID_PARAMETER
```

Testing our Inputs in R

Note: we are overloading the return value in R since this language does not support references to primitive types - there is a solution but it is out of scope

```
INVALID_PARAMETER   <-  -1000
getPercent <- function( value, pct ) {
    if( pct <= 0 ) {
        print( "parameter 2 is invalid" )
        return( INVALID_PARAMETER  )
    } else {
        result <- value * ( pct / 100 )
        return( result )
    }
}
```

Too Much of a Good Thing

Warning signs that a function is too long and needs refactoring:

- Deeply nested logic
- Totally different behavior based on inputs
- Duplicated chunks of code (DRY Principle)
- What it does is not obvious from inspection
- Does not fit on a reasonably sized monitor