Understand Software Quality's Three-Legged Stool: Static Analysis, Dynamic Analysis and Unit Testing

Mark Richardson

Software Quality

- Static Analysis
- Dynamic Analysis
- Unit Testing

Types of Defects

- Coding Defects
  - Ex: accessing outside the bounds of an array
  - Can be detected with Static Analysis

- Application Defects
  - Requires knowledge of exactly what the application is supposed to do
  - Can be detected with High Level Tests that verify the High Level Requirements and with Low Level Tests to verify the Low Level Requirements

Static Analysis

- Code Review:
  - Allows the code to be checked against a coding standard such as MISRA C:2012/AMD1 as well as checking naming conventions and even coding style guidelines
- Quality Review:
  - Allows a number of metrics to be measured on the code in order to get an idea of the quality of the code. For example measuring how many parameters each function has
- Control Coupling:
  - Allows the creation of call graphs to understand the control coupling between all the functions
- Data Coupling:
  - Allows the data coupling to be measured to understand exactly where and how each variable is used

Code Review

- Roughly 80% of software defects when using the C or C++ language, are attributable to the incorrect usage of 20% of the language constructs
- If the usage of the language can be restricted to avoid this subset that is known to be problematic, then the quality of the ensuing software is going to greatly increase

If you don’t want defects in your code, then don’t put them there!

Automation

- Use of tools for automation of standards can greatly improve the efficiency of their application
- Over time engineers will quickly tend to alter their habits and write compliant code
- Coding Standards should be adopted from the outset of the project
- If a coding standard is used with existing code that has a proven track record, then the benefits of using the coding standard may be outweighed by the risk of introducing defects in making the code compliant

Checking Compliance

- Check the code manually
  - Needs to be done on MISRA C:2012 “undecidable” rules
  - But don’t really want to do it on all the code!
- Use a lightweight tool, such as is often built into compilers
  - Fast (Checks just a subset)
  - Detects the easy to find defects
  - Tends to be “Optimistic” – False Negatives

- Use a heavyweight tool
  - Slow (Deep analysis, Check all rules)
  - Detects the easy and hard to find defects (The ones that occur once a year!)
  - Tends to be “Pessimistic” – False Positives

Code Review

- A code review can be performed on the code. Any violations should be fixed or justified (as long as the violated rule is not a Mandatory rule)

- TCI_CodeReview: Check compliant to MISRA C:2012/AMD1
  - None of 'Mandatory' MISRA-C:2012/AMD1 Violations
  - None of 'Required' MISRA-C:2012/AMD1 Violations
  - None of 'Advisory' MISRA-C:2012/AMD1 Violations
  - Output: Callgraph - Programming Standards for GCC_ARM_Cashregister (Artifact)
  - Code Review Report Artifact Placeholder fulfilled by 1 item
    - GCC_ARM_Cashregister.frm.htm
  - Deviation Justification Report fulfilled by 2 items
    - DeviationReport.xml
    - Generate_DeviationReport.bat

Low Complexity

- Many industrial standards require the enforcement of “Low Complexity”

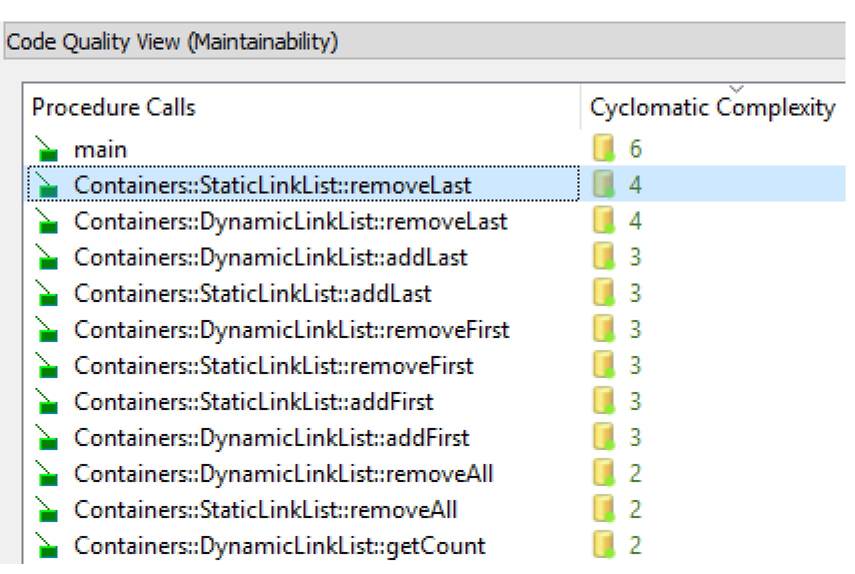
- ISO 26262 - Road Vehicles Safety Standard - Unfulfilled
  - Part 6: - Product Development : Software Level - Unfulfilled
    - Section 5 - Initiation of product development at the software level - Unfulfilled
      - Table 1 - Topics to be covered by modelling and coding guidelines - Unfulfilled
        - 1a - Enforcement of low complexity - Unfulfilled

- Low Complexity is essential to ensure that the code is:
  - Testable
  - Maintainable
  - Clear

- Trying to test, maintain or even to understand complex code is a waste of valuable time and sometimes is impossible, leading to defects remaining in the code

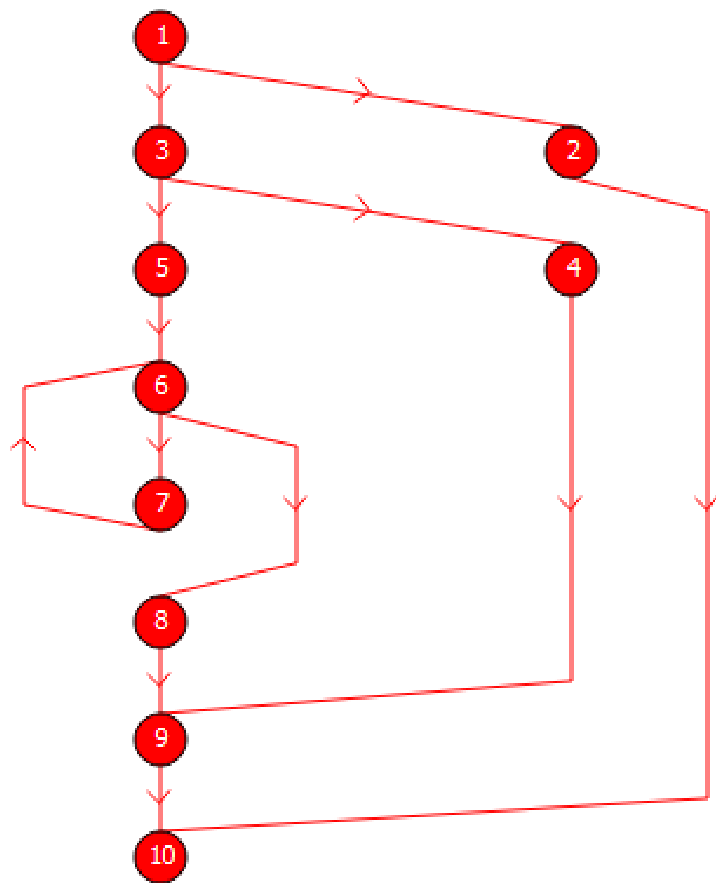
What is Low Complexity?

- Static analysis can be performed on the code and a number of metrics measured such as:
  - Number of lines of code
  - Number of exit points
  - Fan in / Fan out
  - McCabe Cyclomatic Complexity

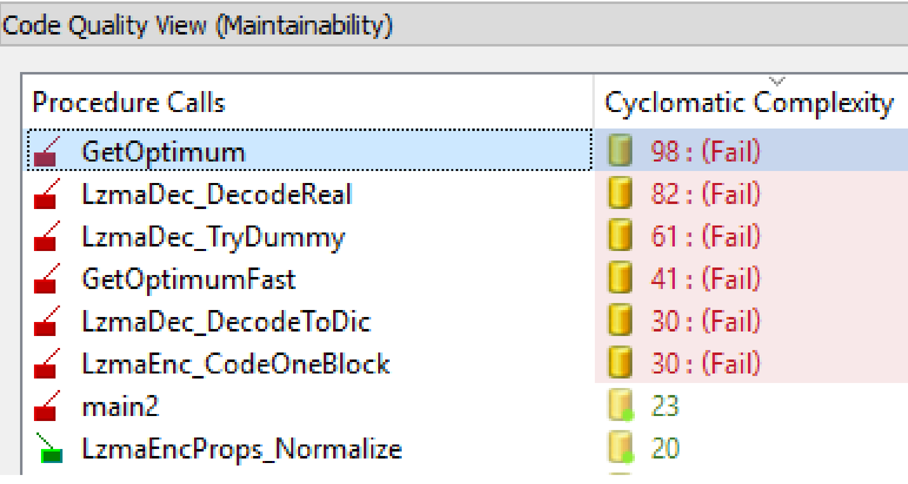


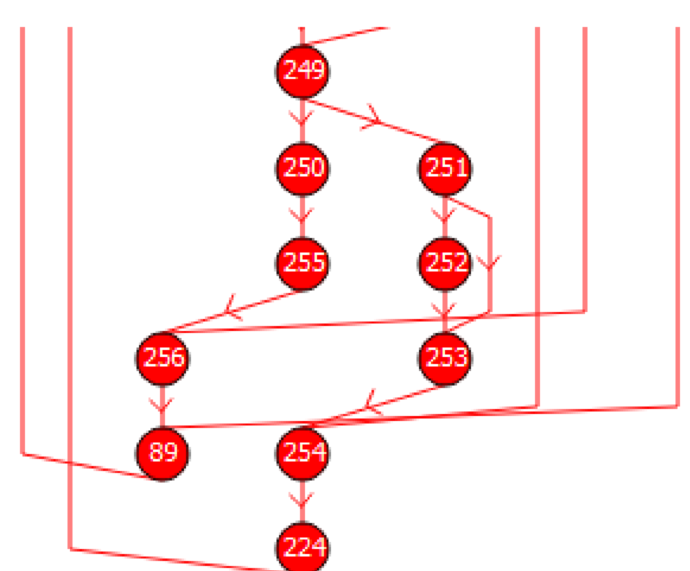
Code Quality View (Maintainability)

Procedure Calls	Cyclomatic Complexity
main	6
Containers::StaticLinkList::removeLast	4
Containers::DynamicLinkList::removeLast	4
Containers::DynamicLinkList::addLast	3
Containers::StaticLinkList::addLast	3
Containers::DynamicLinkList::removeFirst	3
Containers::StaticLinkList::removeFirst	3
Containers::StaticLinkList::addFirst	3
Containers::DynamicLinkList::addFirst	3
Containers::DynamicLinkList::removeAll	2
Containers::StaticLinkList::removeAll	2
Containers::DynamicLinkList::getCount	2



Example of High Complexity





Toyota – Unintended Acceleration

- Cost $1.2B Fine by U.S. Attorney’s Office
- Millions of vehicles recalled
- Barr Group study found:
  - Spaghetti Data Flow
    - 2005 Camry L4 >11,000 global variables
  - Spaghetti Control Flow code
    - Many long, overly complex function bodies
    - 67 functions with Cyclomatic Complexity over 50 (“untestable”)
    - Throttle angle function scored over 100 (“unmaintainable”)

Quality Review

- A quality review can be performed on the code and all the metrics checked to ensure that they are within the specified thresholds

- TCI_Quality_Review: Check that the code is Clear, Maintainable and Testable
  - Maintainability
  - Clarity
  - Output: Callgraph - All Metrics for GCC_ARM_Cashregister (Artifact)
  - Testability
  - Metrics threshold file fulfilled by 1 item
    - Metpen.dat
  - Quality Review Report Artifact Placeholder fulfilled by 1 item
    - GCC_ARM_Cashregister.frm.htm

Control Coupling

TCI_ControlCoupling: Control Coupling A...
Control Coupling Report Artifact Plac...
GCC_ARM_Cashregister.frm.htm

Verify Coupling with LDRA tool suite...
Unmap from Set

main
term_getchar Userinterface_parse
parse_key goodbye Cashregister_key
parse_commands
Cashregister_code randomShopping Userinterface_help
Cashregister_barcode rand Cashregister_end Cashregister_cancel Cashregister_start
identifyProduct generateTicket endSession startSession removeLastProduct
Productdatabase_getProduct countProducts Specialoffer_getPrice addProduct Productdatabase_getSpecificCountedProduct digit_to_char Userinterface_print
Productdatabase_getCountedProduct Productdatabase_resetCountedProducts strcat strcpy Userinterface_show
term_printf

Data Coupling

Variable Name	File	Procedure	Type Code	Attribute Code	Used on lines...						
lpmessage	Cashregister.c	removeLastProduct	L	E	246						
			L	R	256	257					
			L	D	255	256					
message	Cashregister.c	addProduct	L	E	65						
			L	R	74	75					
			L	D	73	74					
msgString											
						217					
						172	176	178	179	180	182
						184	185	186	188	191	193
						195	196	197	204	207	210
						213	214	215	216		
pIter											
						44	46	51			
						47	51				
price											

```
63  static void addProduct(const struct Product * aProduct)
64  {
65    LDRA_char_t message[MAX_STRING];
66
67    if (scannedProducts < MAX_PRODUCTS_IN_BASKET)
68    {
69      if (aProduct != NULL_POINTER)
70      {
71        ShoppingBasket[scannedProducts] = aProduct;
72        scannedProducts++;
73        (void)strcpy(message, "Adding ");
74        (void)strcat(message, aProduct->name);
75        Userinterface_show(&message[0]);
76      }
77    }
78    else
79    {
80      Userinterface_show("Basket is full");
81    }
82  }
```

Testing

- Once the Static Analysis has been performed and the code shown to be compliant to the required coding standard and also the metrics shown to be within the required threshold, then the code can start to be tested

GCC_ARM_Cashregister_SLR.docx
100% Verified
6 Items
6 Verified
0 Unverified

GCC_ARM_Cashregister_HLR.docx
100% Verified
19 Items
19 Verified
0 Unverified

GCC_ARM_Cashregister_HLT.xlsx
100% Verified
19 Items
19 Verified
0 Unverified

GCC_ARM_Cashregister_LLR.xlsx
100% Verified
63 Items
63 Verified
0 Unverified

GCC_ARM_Cashregister_LLT.xlsx
100% Verified
27 Items
27 Verified
0 Unverified

Requirements

- We can’t test without knowing what the code is supposed to do and each function must trace to a low level requirement ex:

SYS_100, Manage products
SYS_101, User Interface output
SYS_102, Help
SYS_103, Final ticket
SYS_104, Manage session
SYS_105, User Interface Input

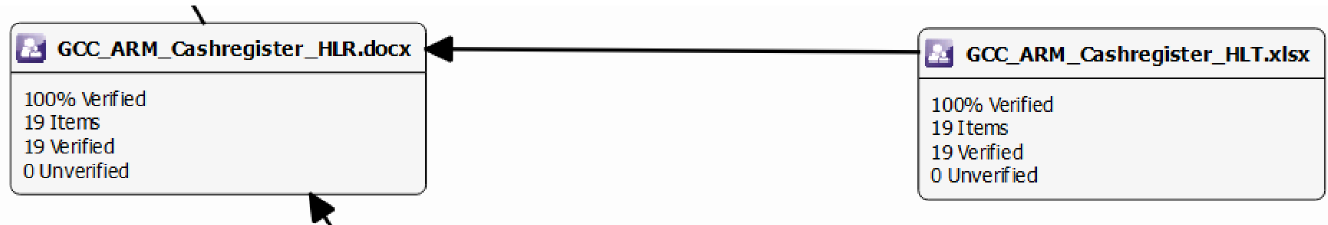
HLR_100, Add Products
HLR_101, Display when scanned
HLR_102, Empty when end
HLR_103, Generate ticket when end
HLR_104, Cancel session
HLR_105, Scan process barcodes
HLR_106, Start with empty
HLR_107, Key in process barcodes
HLR_108, Count shopping basket products
HLR_109, Generate ticket
HLR_110, Display removed product
HLR_111, Manage user interface session
HLR_112, Incorporate special offer
HLR_113, Inform end session
HLR_114, Diplay help when requested
HLR_115, User input
HLR_116, Display help when incorrect input
HLR_117, Print ticket
HLR_118, Display ticket

LLR_100, addProduct: add
LLR_101, addProduct: display
LLR_102, addProduct: too many
LLR_103, addProduct: check against null ...
LLR_104, countProducts: count
LLR_105, countProducts: check product e...
LLR_106, countProducts: check counted ...
LLR_107, endSession: empty basket
LLR_108, endSession: display message
LLR_109, generateTicket: print ticket
LLR_110, generateTicket: set idle mode
LLR_111, generateTicket: ignore if idle
LLR_112, identifyProduct: name
LLR_113, identifyProduct: non-existent
LLR_114, removeLastProduct: remove
LLR_115, removeLastProduct: no products
LLR_116, removeLastProduct: check prod...
LLR_117, startSession: empty
LLR_118, Cashregister_barcode: active
LLR_119, Cashregister_barcode: idle
LLR_120, Cashregister_cancel: remove pro...
LLR_121, Cashregister_cancel: idle
LLR_122, Cashregister_cancel: remove pro...
LLR_123, Cashregister_code: display prod...
LLR_124, Cashregister_code: idle
LLR_125, Cashregister_end: generate ticket
LLR_126, Cashregister_end: idle

```
LDRA_int32_t main();
LDRA_uint32_t Specialoffer_getPrice(const LDRA_uint32_
const struct Product * Productdatabase_getProduct(con
struct CountedProduct * Productdatabase_getCountedP
struct CountedProduct * Productdatabase_getSpecificC
void Cashregister_barcode(const LDRA_uint32_t aCode);
void Cashregister_cancel();
void Cashregister_code();
void Cashregister_end();
void Cashregister_key(const LDRA_uint32_t aKey);
void Cashregister_start();
void Productdatabase_resetCountedProducts();
void Userinterface_help();
void Userinterface_parse(const LDRA_char_t aChar);
void Userinterface_print(LDRA_const_char_pt printerMsg
void Userinterface_show(LDRA_const_char_pt displayMs
void addProduct(const struct Product * aProduct);
void countProducts();
void endSession();
void generateTicket();
void goodbye();
void identifyProduct(const LDRA_uint32_t aBarcode);
void parse_commands(const LDRA_char_t aChar);
void parse_key(const LDRA_char_t aChar);
void randomShopping();
void removeLastProduct();
void startSession();
```

High Level Tests

- First the High Level Requirements need to be verified, for example one way to test is to provide known inputs, capture the outputs and finally compare the outputs against a known reference



- TCI_HLT_100: Add Products
  - Input_File fulfilled by 1 item
    - HLR_Input_Add_Products.txt
  - Reference_File fulfilled by 1 item
    - HLR_Reference_Add_Products.txt
  - Output_File fulfilled by 1 item
    - HLR_Output_Add_Products.txt

Dynamic Analysis

- In order to find out how effective the tests are, dynamic analysis can be performed
- In this case the code being tested is instrumented and then the same High Level Test is performed
- After the test completes the structural coverage can be measured to understand how much of the code was exercised by the High Level Test

TCI_CodeCoverage: Structural coverage
- 100% Statement Coverage
- 100% Branch Coverage
- 100% Modified Condition / Decision Coverage
- Output: Callgraph - Pass/Fail Coverage for GCC_ARM_Cashregister (Artifact)
- Dynamic Coverage Report Artifact Placeholder fulfilled by 1 item
  - GCC_ARM_Cashregister.frm.htm

Structural Coverage

LDRA Coverage Pass/Fail Flowgraph of procedure : parse_commands Statement: 59% Branch/Decision: 53% MC/DC: n/a

Graph View Options Select Website Links Help

parse_commands

1 2 3 4 5 6 7 8 9

Source Viewer - parse_commands

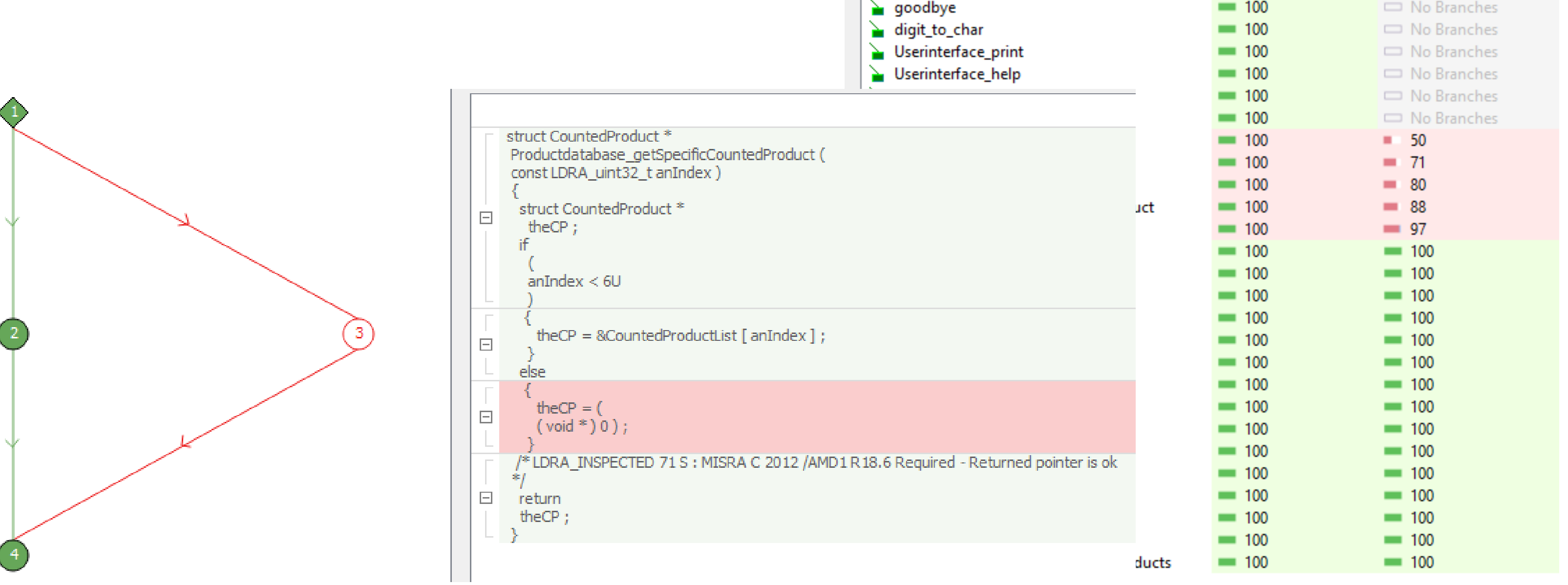
```
static void
  parse_commands (
  const LDRA_char_t aChar )
  {
   switch (
    aChar
    )
    {
    case 'b' :
      Cashregister_code () ;
      break ;
    case 'c' :
      Cashregister_cancel () ;
      break ;
    case 'e' :
      Cashregister_end () ;
      break ;
    case 's' :
      Cashregister_start () ;
      break ;
    case 'r' :

      randomShopping () ;
      break ;
    case '\n' :
          /* ignore crlf */
     case '\r' :
          /* ignore crlf */
      break ;
     /* For any other character, display the help message
```

Filtered File Explorer | Source Viewer - parse_commands

Branch/Decision Covered   Branch/Decision not Covered - (Diamond - Node has Branch/Decision | Circle - Node has no Branch/Decision)

Structural Coverage Gap Analysis

- Once all the High Level Tests have been executed, any code that remains un-exercised should be either removed, (if deemed to be dead-code), or new tests created to exercise it, or if in the case of defensive code (for example checking that a pointer is not null) then unit tests can be created to achieve 100% structural coverage



Unit Testing

- The main goal of unit testing is to ensure that the “unit under test” functions as expected
- A secondary goal of unit testing is to be able to exercise the parts of code that were not exercised during the High Level Tests. Functions can be stubbed so that they return values allowing all the defensive code to be exercised
- Another goal of unit testing is to perform robustness tests, this is where automatically tests can be created using min/mid/max values, upper and lower boundary values for conditions …

Low Level Requirement

- Here is an example of a low level requirement:

ID		Function
LLR_100	The function shall search in a string for any placeholder. If found return the numerical value as well as the placeholder. If not found return -1. A placeholder is defined in 3 parts: prefix, number, suffix prefix is "$(CSV" number is any value from 1 to 99 suffix is ")" ex: $(CSV4)	int getPlaceholder (const std::string& line, std::string& placeholder)

- In order to test this requirement, we first need to find some test cases

Normal Test Cases

- These are the normal types of input to this function:

LLR_100	int getPlaceholder (const std::string& line, std::string& placeholder)			
Test Case	Description	line	Return	placeholder
TC1	Lower bound at end of string	"String is $(CSV1)"	1	"$(CSV1)"
TC2	Upper bound at end of string	"String is $(CSV99)"	99	"$(CSV99)"
TC3	Lower bound in middle of string	"Start $(CSV1) End"	1	"$(CSV1)"
TC4	Upper bound in middle of string	"Start $(CSV99) End"	99	"$(CSV99)"
TC5	Lower bound at start of string	"$(CSV1) End"	1	"$(CSV1)"
TC6	Upper bound at start of string	"$(CSV99) End"	99	"$(CSV99)"
TC7	No placeholder	"This is a string"	-1	No change
TC8	Just the placeholder	"$(CSV18)"	18	"$(CSV18)"

Robustness Test Cases

- These are types of input that are not expected, but which need to be handled

LLR_100	int getPlaceholder (const std::string& line, std::string& placeholder)			
Test Case	Description	line	Return	placeholder
TC1	Missing suffix at end	“String is $(CSV1”	-1	No change
TC2	Missing suffix in middle	“Start $(CSV1 End”	-1	No change
TC3	Missing number	“Start $(CSV) End”	-1	No change
TC4	Text instead of number	“Start $(CSVone) End”	-1	No change
TC5	Number < 1	“$(CSV0) End”	-1	No change
TC6	Number > 99	“$(CSV100) End”	-1	No change
TC7	Missing prefix	“(CSV8)”	-1	No change
TC8	Malformed placeholder	“Start S(CSV42) End”	-1	No change
TC9	Double placeholder	“$(CSV5) , $(CSV6)”	5	“$(CSV5)”

Normal Test Cases : PASS

- These Test Cases can then be executed

Test Case View

Test Case	Regression P / F	Procedure	Name / Description
Tc 1	PASS	::getPlaceholder	Lower bound at end of string
Tc 2	PASS	::getPlaceholder	Upper bound at end of string
Tc 3	PASS	::getPlaceholder	Lower bound in middle of string
Tc 4	PASS	::getPlaceholder	Upper bound in middle of string
Tc 5	PASS	::getPlaceholder	Lower bound at start of string
Tc 6	PASS	::getPlaceholder	Upper bound at start of string
Tc 7	PASS	::getPlaceholder	No placeholder
Tc 8	PASS	::getPlaceholder	Just the placeholder

Variable I/O View

	Value	Name
I	"This is a string"	line
I	"anything"	placeholder
O	"anything"	placeholder
O	-1	%

TC7	No placeholder	“This is a string”	-1	No change

LLR_100_Robustness_Test_Cases

- In this case some of the robustness test cases fail
- This is exactly why unit testing is performed!

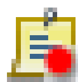
Test Case View

Test Case	Regression P / F	Procedure	Name / Description
Tc 1	PASS	::getPlaceholder	Missing suffix at end
Tc 2	PASS	::getPlaceholder	Missing suffix in middle
Tc 3	FAIL exception	::getPlaceholder	Missing number
Tc 4	FAIL exception	::getPlaceholder	Text instead of number
Tc 5	FAIL	::getPlaceholder	Number < 1
Tc 6	FAIL	::getPlaceholder	Number > 99
Tc 7	PASS	::getPlaceholder	Missing prefix
Tc 8	PASS	::getPlaceholder	Malformed placeholder
Tc 9	PASS	::getPlaceholder	Double placeholder

TC3	Missing number	“Start $(CSV) End”	-1	No change
TC4	Text instead of number	“Start $(CSVone) End”	-1	No change

Unit Testing

- Ideally Unit Testing would be performed on all the functions, or alternatively just on the functions which did not achieve 100% structural coverage



SYS_100, Manage products
- HLR_100, Add Products
  - LLR_100, addProduct: add
    - TCI_LLT_100: addProduct
      - Input: Unittest_addProduct.tcf (Asset)
      - Output: GCC_ARM_Cashregister.frm.htm (Artifact)
  - LLR_101, addProduct: display
  - LLR_102, addProduct: too many
  - LLR_103, addProduct: check against null pointer
  - TCI_HLT_100: Add Products

Summary

- As we have seen, to have high Quality Code, we need to perform all of the following activities:
  - Static Analysis
  - Dynamic Analysis
  - Unit Testing

Any Questions

Q & A

Contact Us

Need more information?

LDRA.com

Linkedin

info@ldra.com