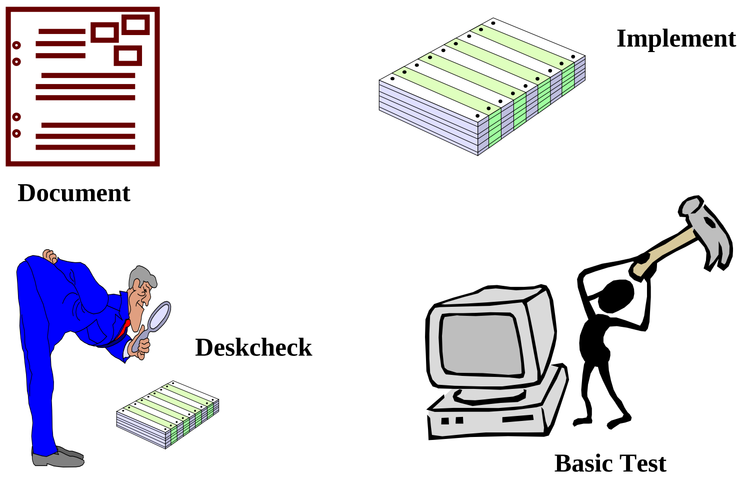
Implementation
(Low Level Design)

Low Level Design Activities



Document

Implement

Deskcheck

Basic Test

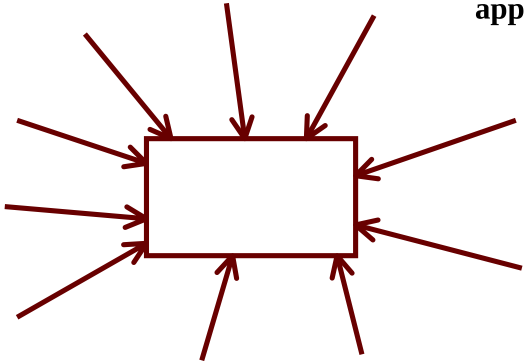
What is a Good Low Level Module?

- Black box aspects
- White box aspects

Black Box Aspects

- Fulfilled functionality
- Fulfilled characteristics
- Easy to use
- Integratable
- Reusable
- Testable
- Traceable
- Backward Compatible
- Balanced role

Fulfilled Functionality



Stimuli from asynchronously applied use cases.

Fulfilled Characteristics

- Response times
- Processor load
- Static data size
- Dynamic data size
- Code size

Depends on used algorithms and data structure!

Easy to Use

- Well documented
  - “Users Manual”
- Understandable role
- Intuitive functionality
- Simple interface
- Powerful functionality
- Low dependency

Integratable

- Correct tolerance level
  - Avoid unnecessary limitations, but...
  - Also avoid defensive programming
    - Never hide a fault!
- Strive for self containment
- No cyclic dependencies
- Design by Contract
  - Preconditions
  - Postconditions
  - Invariants

Reusable

- More generic than what is explicitly required.
  - Broader value ranges
  - Arbitrary data types
  - ….
  - Caution: Do not spend time on this!
- Document *actual* functionality

Testable

- Good test script
- Test interface
  - State observability
  - Flow observability
- Testable algorithms
- Observable test harness

Traceable

(Valid for traceable functionality only)

- Future impacts must be verified against all existing users.
- Existing solutions can not be understood if already implemented requirements are lost.
- Risk that existing system functionality stops working after modifications.
- Other modules may require change to allow changes in this module.

Backward Compatible

- Requirement for non-traceable modules.
- Desired for traceable modules.
- All existing use must be supported by new releases.
- Tough and expensive requirement to fulfill.

Balanced Role

- Inappropriate functional decomposition often discovered during low level design.
- Functionality and responsobilities may better be moved to other modules.
- Deviations from input design documentation sometimes acceptable.
- Deviations must be approved by project management and higher level design.
- Use with caution. If possible, wait until next iteration. (The bigger the project, the harder to deviate.)

White Box Aspects

- Deductable
- Understandable
- Modifiable
- Fault free

Deductable

- Try to find an intuitive coupling between problem and solution.
- Internal structure should reflect the modules role.
- Implementation correctness should be possible to determine by desk check.

Understandable (1)

- “Repairmans Manual”
- Comments
  - What is the role of arguments
  - What is the purpose of the next code segment?
  - Why is a decision taken?
  - ...
- Low complexity
  - Structured programming
  - Use the most understandable solution unless in conflict with characteristics requirements.

Understandable

- Self explanatory code
  - Expressive function names
    - Imperative or functional names. Be consequent, follow standards.
  - Expressive variable names
  - Expressive data typing
- Standards!!! (Rules & recommendations)
  - Naming
  - “Body language” (indentations, bracket placement...)
  - Commenting style
  - Idioms (code patterns)

Modifiable (1)

- Code will be modified by someone else.
  - It’s your fault he misunderstands anything you did.
- No hidden side effects.
  - Use explicit communication
  - Avoid widely scoped variables
  - Sophisticated OO constructs requires experience and discipline. Don’t get carried away!

Modifiable (2)

- Keep influence local.
  - Encapsulation
  - Limit scope of data, functions, definitions
  - Encapsulate base classes and local classes as well.
  - Avoid C++ friend relationships outside file scope.

Fault Free

- Uninitialized variables
- Incorrect loop terminations
- Invalid pointers
- Incorrect type casting
- Data outside valid value ranges
- Index outside array bounds
- Memory leaks
- Unexpected signals
- Unexpected recursion
- Syntactical pitfalls *if (i = 0) ...*
- Copy & paste mistakes

The Detailed Design Document

Service Information

- a Abstract
- b TOC
- c Document status and history

PART 1—General Description

1 Introduction

- 1.1 Purpose
- 1.2 Scope
- 1.3 Glossary
- 1.4 References
- 1.5 Overview

2 Project Standards, Conventions and Procedures

- 2.1 Design standards
- 2.2 Documentation standards
- 2.3 Naming conventions
- 2.4 Programming standards
- 2.5 Software development tools

PART 2—Component Design Specifications

I Component i *(its name)*

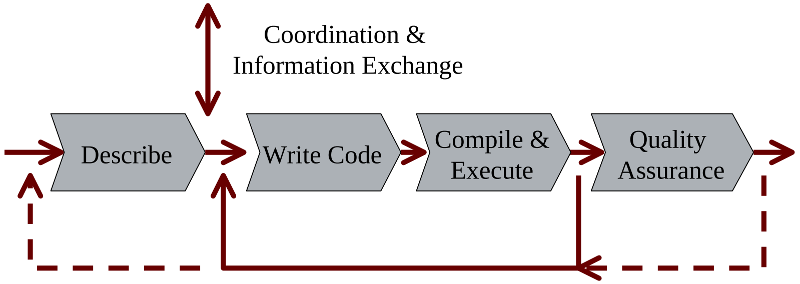
- I.1 Type
- I.2 Purpose
- I.3 Function
- I.4 Subordinates
- I.5 Dependencies
- I.6 Interfaces
- I.7 Resources
- I.8 References
- I.9 Processing
- I.10 Data

Appendix A: Source Code Listings

Appendix B: Software Requirements Vs. Components Traceability Matrix

Slightly adapted from ESA's Software Engineering Standards PSS-05-0 (see [ESA 96])

Implementation Work Flow



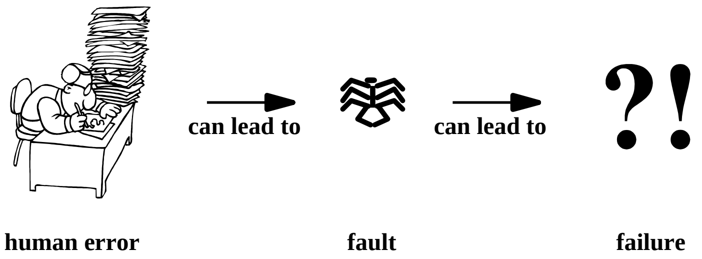
Low Level Quality Assurance (1)

- Basic test
  - ❑ Execution of code on lowest level
  - ❑ Automated tools
  - ❑ Test scripts
  - ❑ Test harnesses
- Desk check
  - ❑ Check list for common faults
  - ❑ Checking rate ~100 LoC / hour

Low Level Quality Assurance (2)

- Tool supported analysis
  - ❑ Execution coverage
  - ❑ Performance
  - ❑ Memory leaks
  - ❑ Common pitfalls
  - ❑ Complexity
  - ❑ Array bounds

Fault vs Failure



human error can lead to fault can lead to ?! failure

Error Handling

- Highlight faults
  - Never hide a fault
  - Disastrous symptoms are good during testing
  - Use error logs for delivered systems
- Avoid failures
  - Try to reduce effect in target system.
  - Failure avoidance strategy depends on criticality
- Unusual conditions are not faults (e.g. disk full)
  - Lack of handling of them are!

Criticality

- Consumer products
- Professional tools
- Industrial systems
- Medical systems
- Auto pilots
- ….

More on Quality Assurance

Coming soon
to a theatre near You!