L3: Building, Testing and Debugging Scientific Software

P. de Oliveira Castro, M. Jam

August 29, 2025

Master Calcul Haute Performance et Simulation - GLHPC | UVSQ

Objectives

- Build systems: Advanced Makefiles, introduction to CMake for managing multi-file and multi-platform projects.
- Debugging: GDB, Valgrind for detecting memory errors and leaks.
- Software testing:
  - Principles: Unit testing, integration testing.
  - Test frameworks in C (e.g., Unity).
  - Importance of testing for regression prevention and validation.
- Code documentation: Doxygen.

Makefiles

Dependency Management

- How to determine which files have changed?

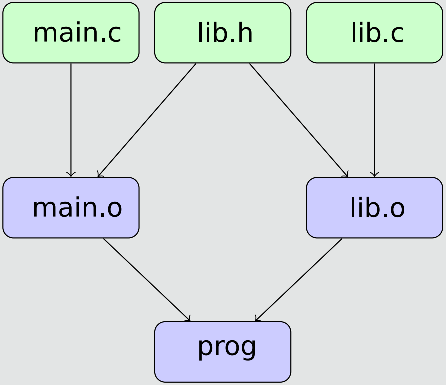

Figure 1: makefile-dependencies

- **dependencies**: `main.o` depends on changes in `lib.h`

Makefile

- A `Makefile` uses a declarative language to describe targets

CMake

Why CMake?

- **Advantages of Makefiles:**
  - Simplicity and transparency.
  - No additional tools required.
  - Direct control over the build process.
- **Advantages of CMake:**
  - Cross-platform support (Linux, Windows, macOS).
  - Generates build files for multiple build systems (Make, Ninja, etc.).
  - Modular and target-based design.
  - Built-in support for testing, installation, and packaging.

General Design of CMake

- **CMake as a Meta-Build System:**
  - Generates build files for different generators (e.g., Make, Ninja).
  - Abstracts platform-specific details.
- **Workflow:**
  1. Write `CMakeLists.txt` to define the project.
  2. Configure the project:

     ```
     cmake -B build
     ```

  3. Build the project:

     ```
     cmake --build build
     ```

Debugging Tools

GDB: GNU Debugger

Valgrind: memory debugging and leak detection

Other tools: ASAN, UBSAN

Software Testing

Importance of Software Testing

- 1996: Ariane-5 self-destructed due to an unhandled floating-point exception, resulting in a $500M loss.
- 1998: Mars Climate Orbiter lost due to navigation data expressed in imperial units, resulting in a $327.6M loss.
- 1988-1994: FAA Advanced Automation System project abandoned due to management issues and overly ambitious specifications, resulting in a $2.6B loss.
- 1985-1987: Therac-25 medical accelerator malfunctioned due to a thread concurrency issue, causing five deaths and numerous injuries.

Technical Debt

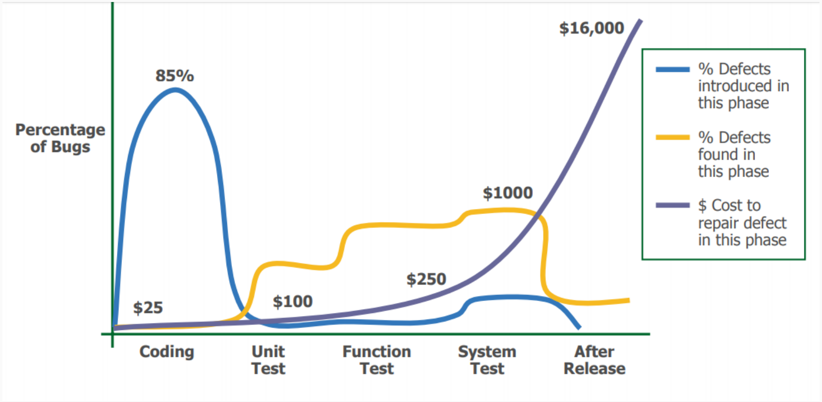

Figure 2: Software Costs (Applied Soft. Measurement, Capers Jones)

Software Costs

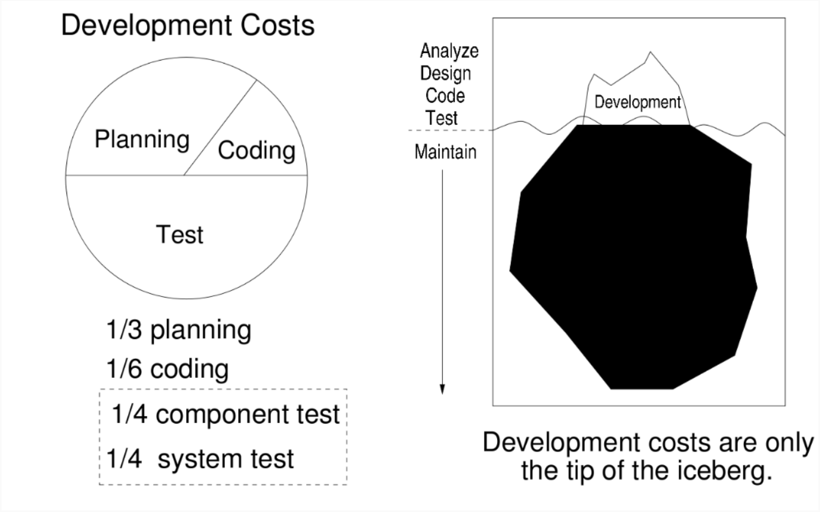

Figure 3: Software Costs (Nancy Leveson)

Verification and Validation (V&V)

- **Validation**: Does the software meet the client's needs?
  - "Are we building the right product?"
- **Verification**: Does the software work correctly?
  - "Are we building the product right?"

Approaches to Verification

- Formal methods
- Modeling and simulations
- Code reviews
- **Testing**

Software Testing

Testing Process (S. Bardin)

Figure 4: Testing Process (S. Bardin)

V Cycle Model

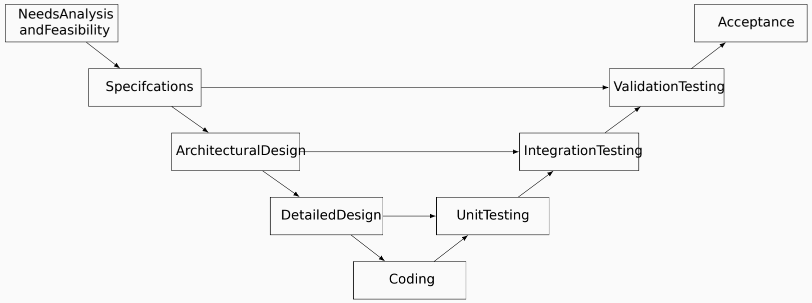

Figure 5: V-Model: Validation followed by Verification

Different Types of Tests

- **Unit Tests:**
  - Test individual functions in isolation.
  - Test-driven development (TDD): Focus on writing maintainable, simple, and decoupled code.
- **Integration Tests:**
  - Test the correct behavior when combining modules.
  - Validate only functional correctness.
- **Validation Tests:**
  - Test compliance with specifications.
  - Test other characteristics: performance, security, etc.
- **Acceptance Tests:**
  - Validate requirements with the client.
- **Regression Tests:**
  - Ensure that fixed bugs do not reappear.

Black-Box and White-Box Testing

Black-Box Testing (Functional)

- Tests are generated from specifications.
- Uses assumptions different from the programmer's.
- Tests are independent of implementation.
- Difficult to find programming defects.

White-Box Testing (Structural)

- Tests are generated from source code.
- Maximizes coverage by testing all code branches.
- Difficult to find omission or specification errors.

Both approaches are complementary.

What to Test?

- Running the program on all possible inputs is too costly.
- Choose a subset of inputs:
    - Partition inputs into equivalence classes to maximize coverage.
    - Test all code branches.
    - Test edge cases.
    - Test invalid cases.
    - Test combinations (experimental design).

Example of Partitioning

Specification

```
/* compare returns:
 *   0 if a is equal to b
 *   1 if a is strictly greater than b
 *  -1 if a is strictly less than b
 */
int compare (int a, int b);
```

What inputs should be tested?

Example of Partitioning

Equivalence Classes

Variable	Possible Values
a	{positive, negative, zero}
b	{positive, negative, zero}
result	{0, 1, -1}

Discussion

- Automatic test generation.
- Test coverage calculation.
- Mutation testing.
- Fuzzing.
- Importance of using automated testing tools.
- Importance of using continuous integration tools.

Credits and Bibliography

- Course “Automated Software Testing,” Sébastien Bardin.