Unit Testing From the Trenches

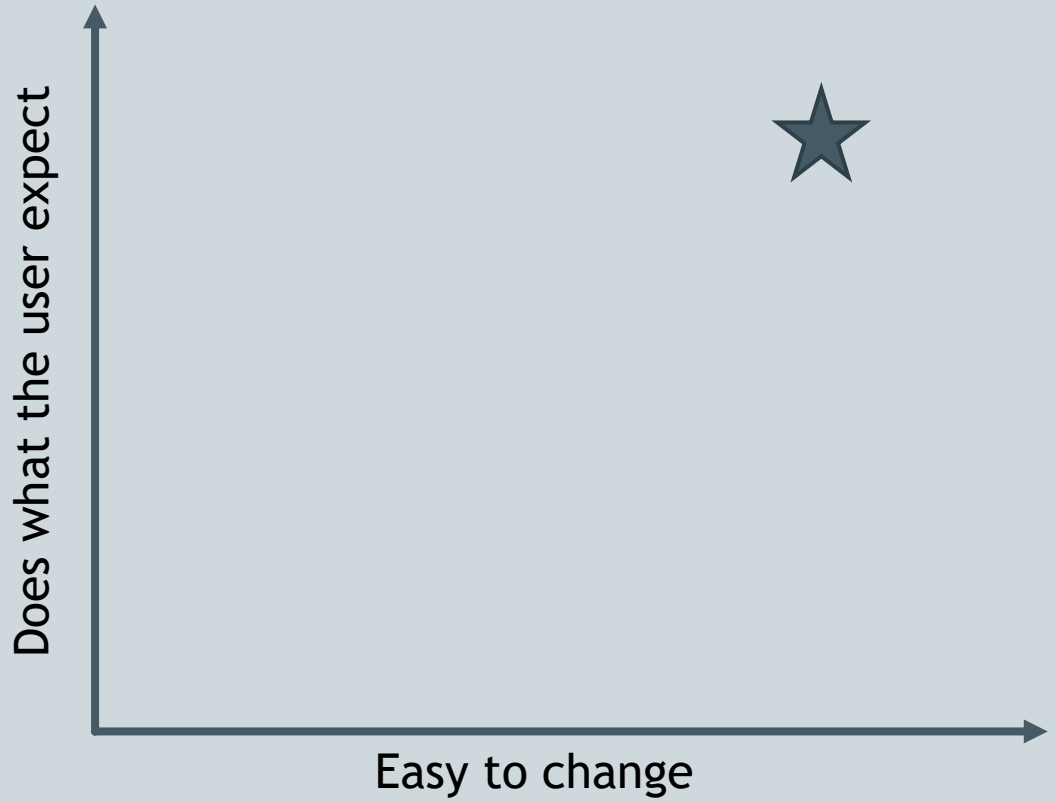
Anton Sundqvist

Introduction

- Test Driven Development and Design
- Real world examples
- Challenges with testing

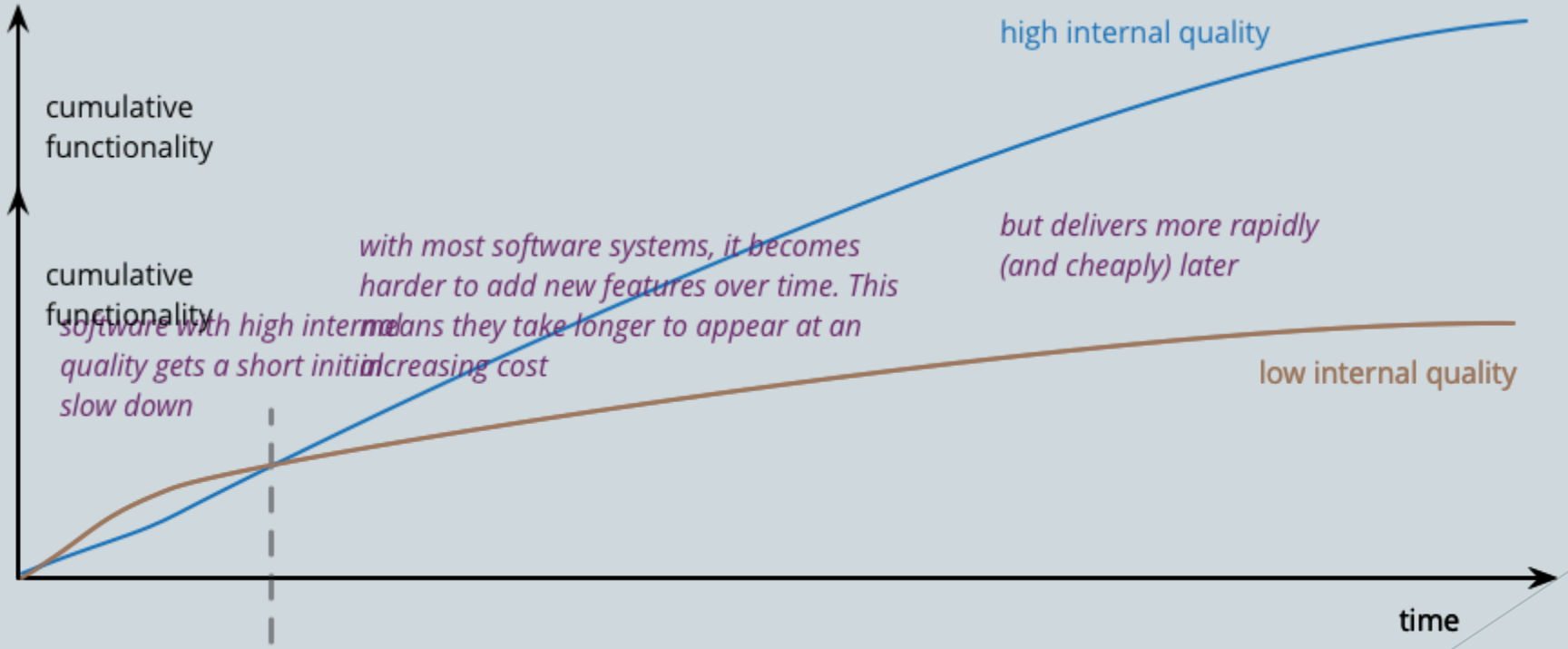
https://github.com/astemes/astemes-gdevcon-2022

Software Quality



How to deliver high quality code at a high pace?

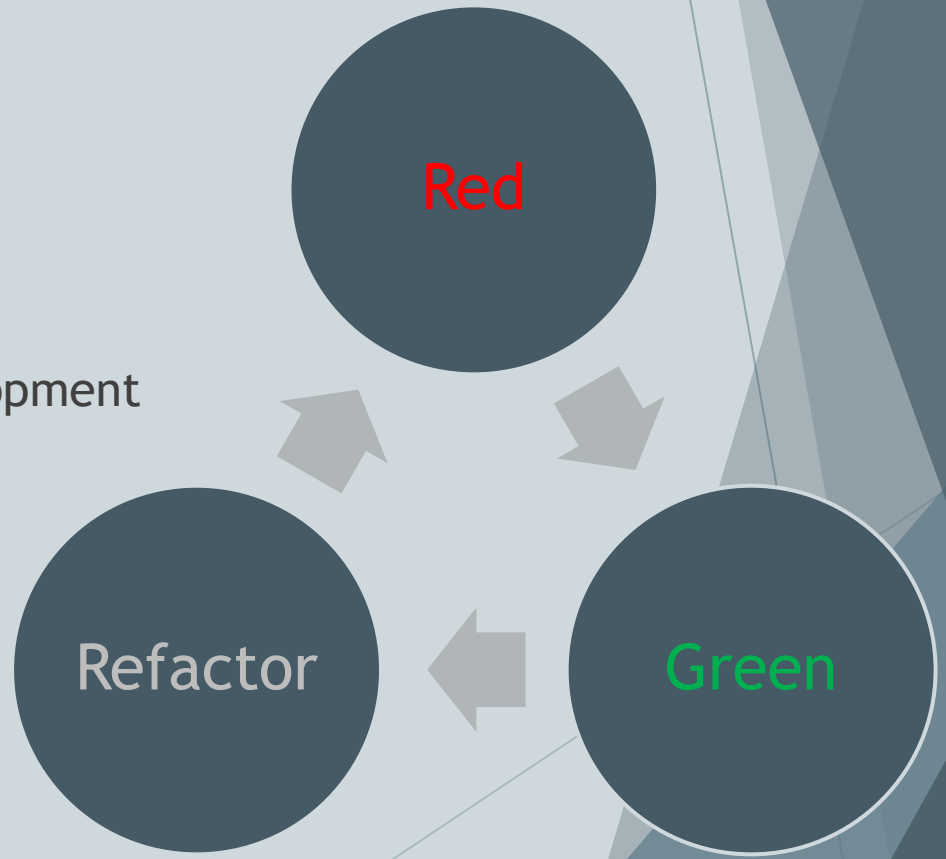
- The only way to go fast is to not make a mess while going



Source: https://martinfowler.com/articles/is-quality-worth-cost.html

Test-Driven Development

- A test is created **before** writing **any** production code
- Code is developed to make the test pass
- Refactor to reduce duplication and clean up
- Red–Green–Refactor **cycle**
- Cycle time on the order of **minutes**
- Writing tests is not a separate activity from development



Why bother with Test Driven Development?

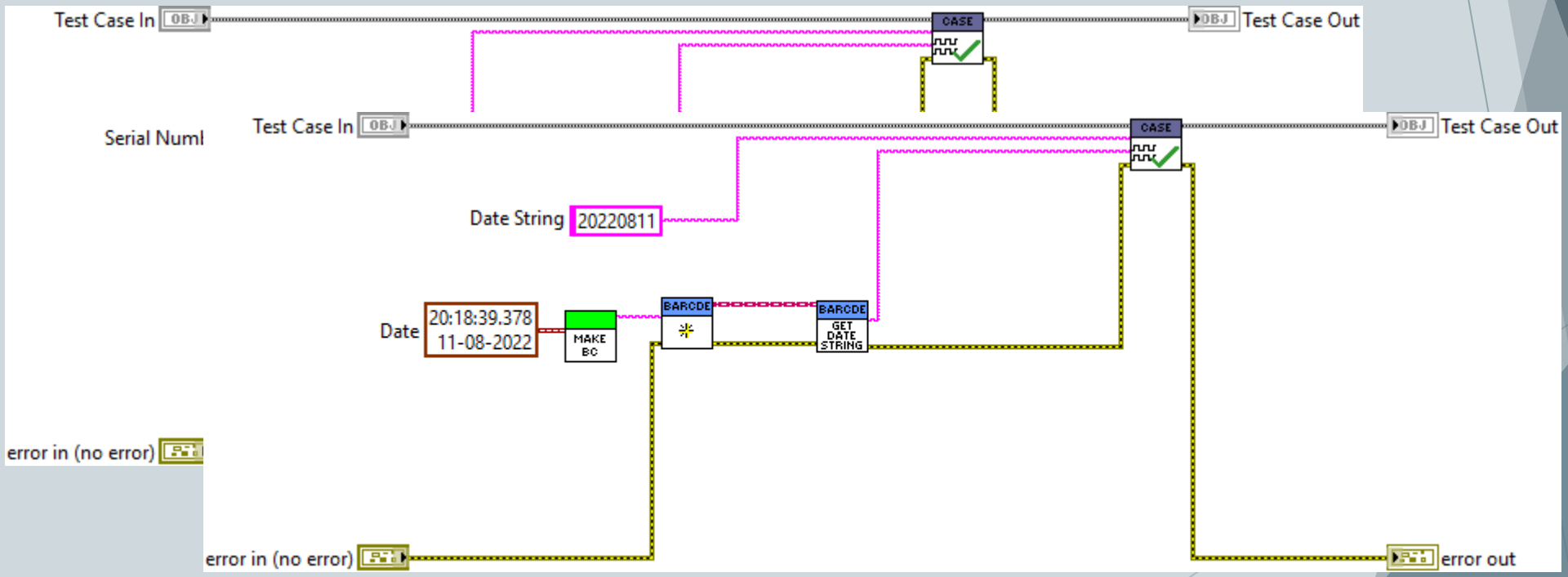
1. Improved Design
2. Self Testing Code
3. Less stress
4. Documentation

Test Driven Design

- What makes code testable?
  - Loose coupling
  - Clear APIs
  - Well managed dependencies
  - Clear responsibilities
  - Limited side effects
- What if we start with writing a test?

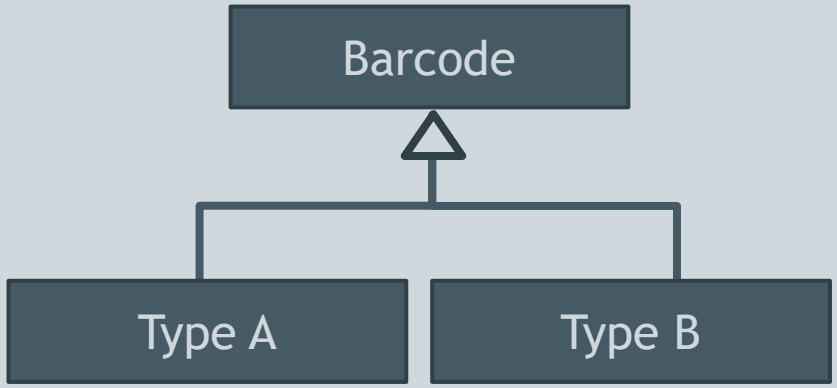
Example: Barcode Parser

$SN**12345**$DT**20220811**

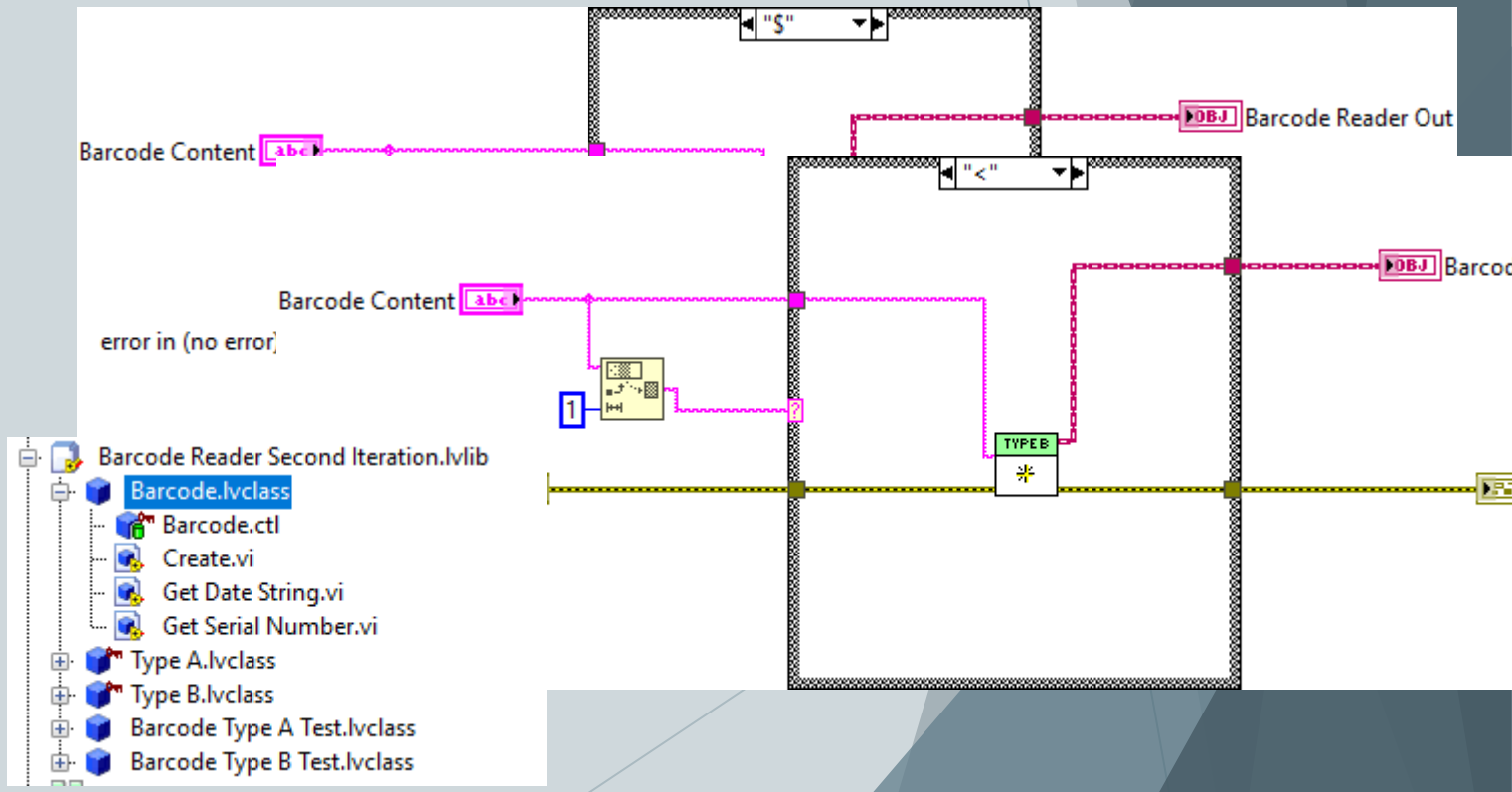


Example: Barcode Parser Extended

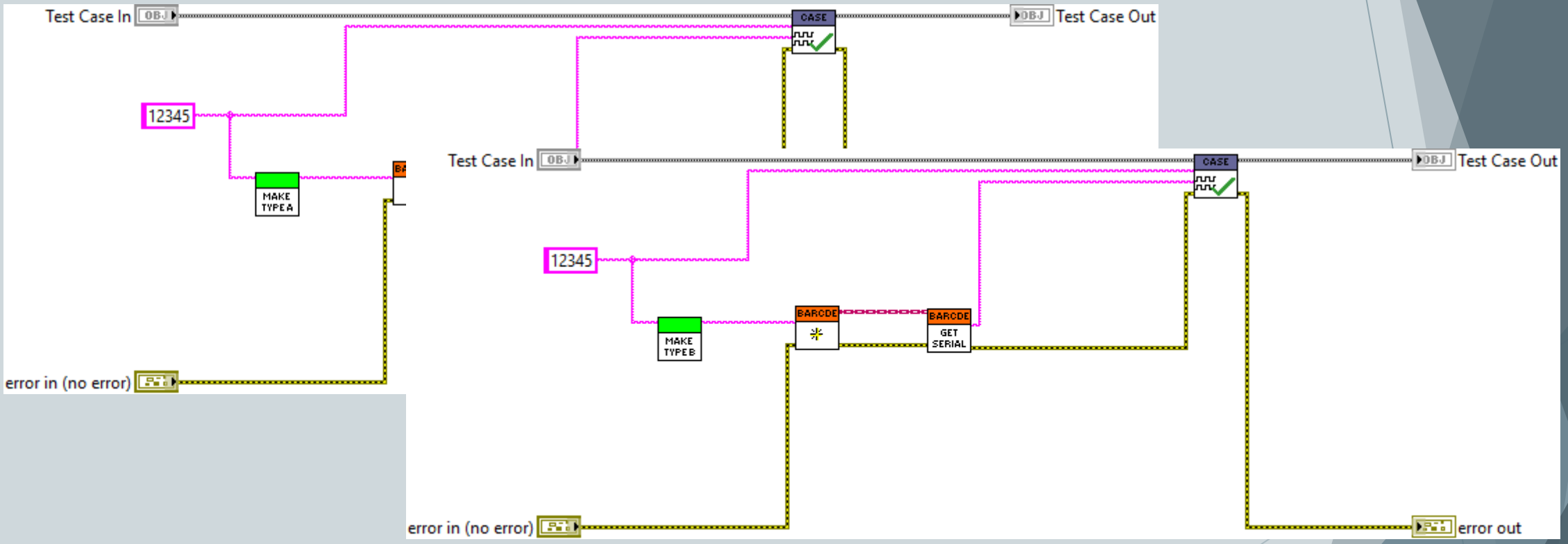
<SN>**12345**</SN><DATE>**20220811**</DATE>



Barcode.lvclass:Create.vi

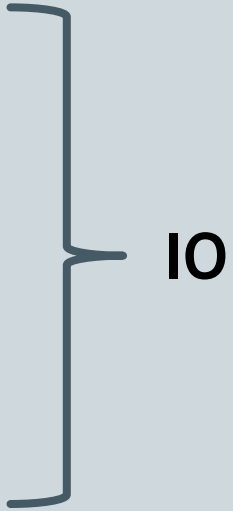


Example: Barcode Parser Extended Tests



Sounds good, but…

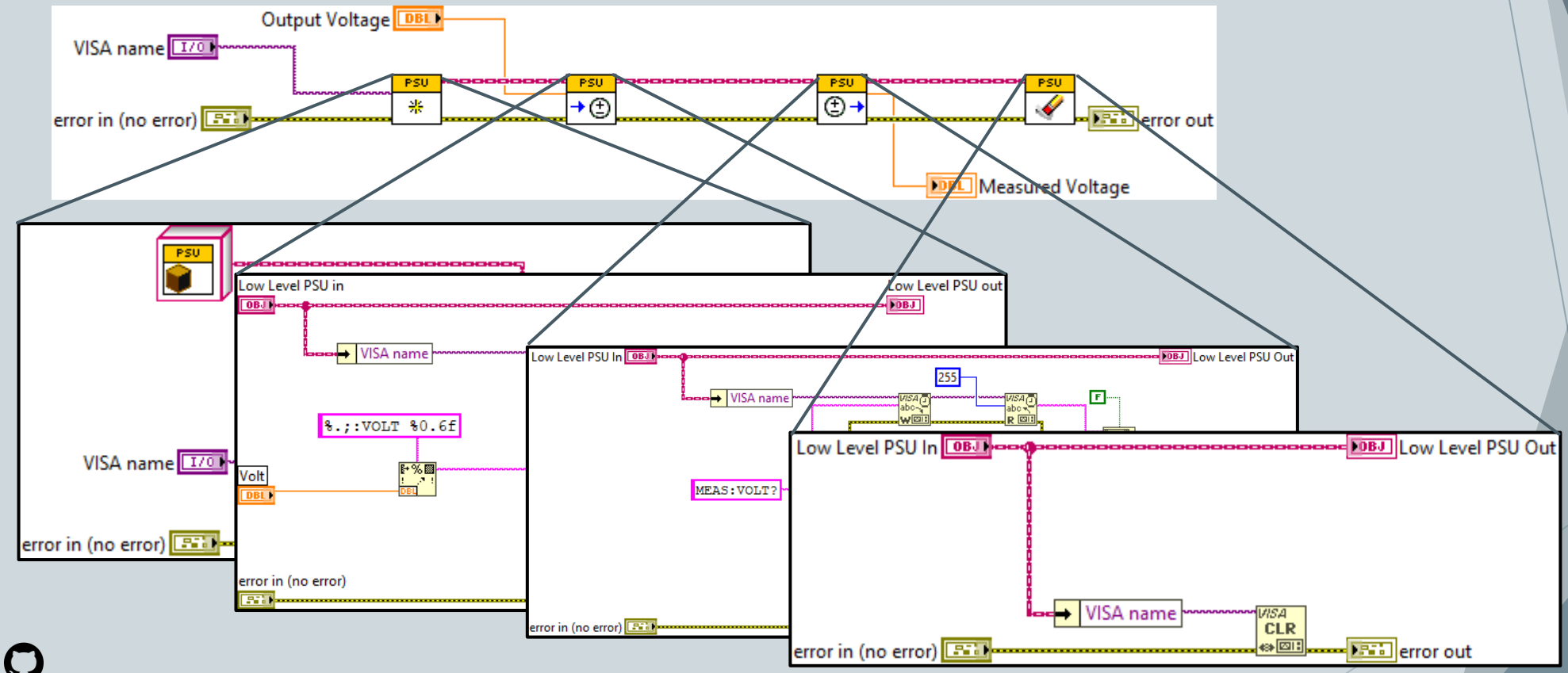
- Instrumentation
- Communication busses
- User interfaces
- Command Line
- Databases
- Reports



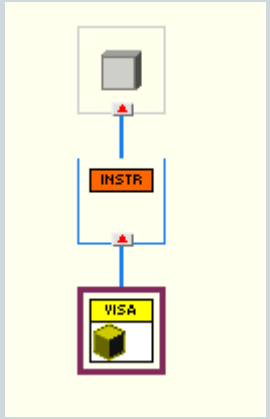
Testing at the Edges

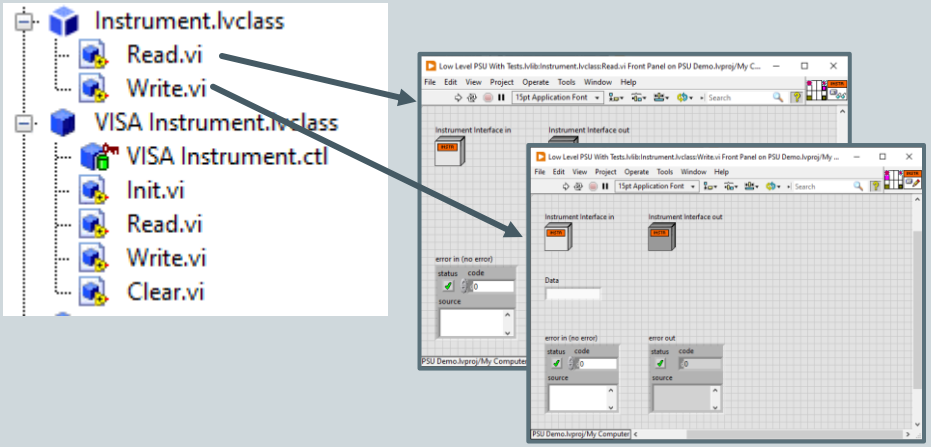
- When something is difficult to test – get rid of the stuff which makes testing difficult

Example: Low Level Hardware

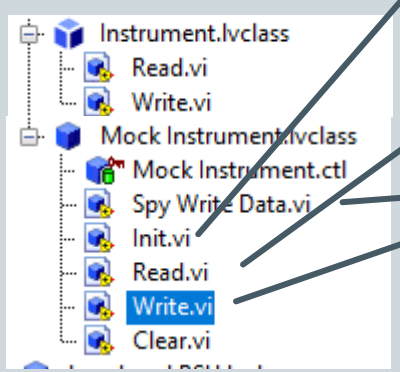


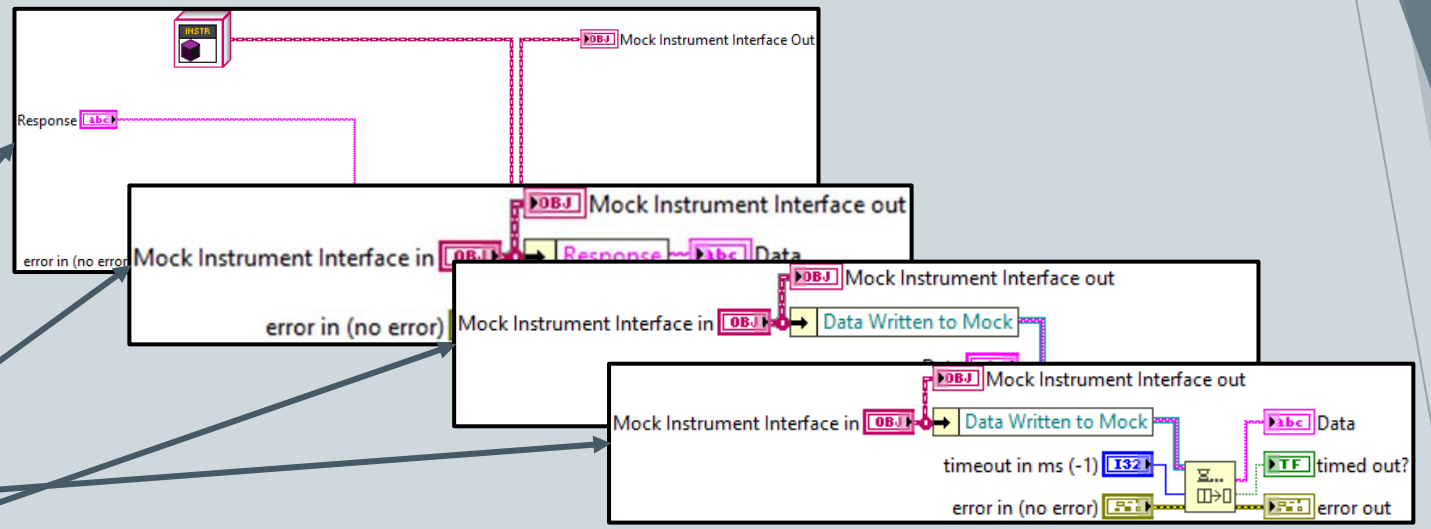
Pulling out VISA...

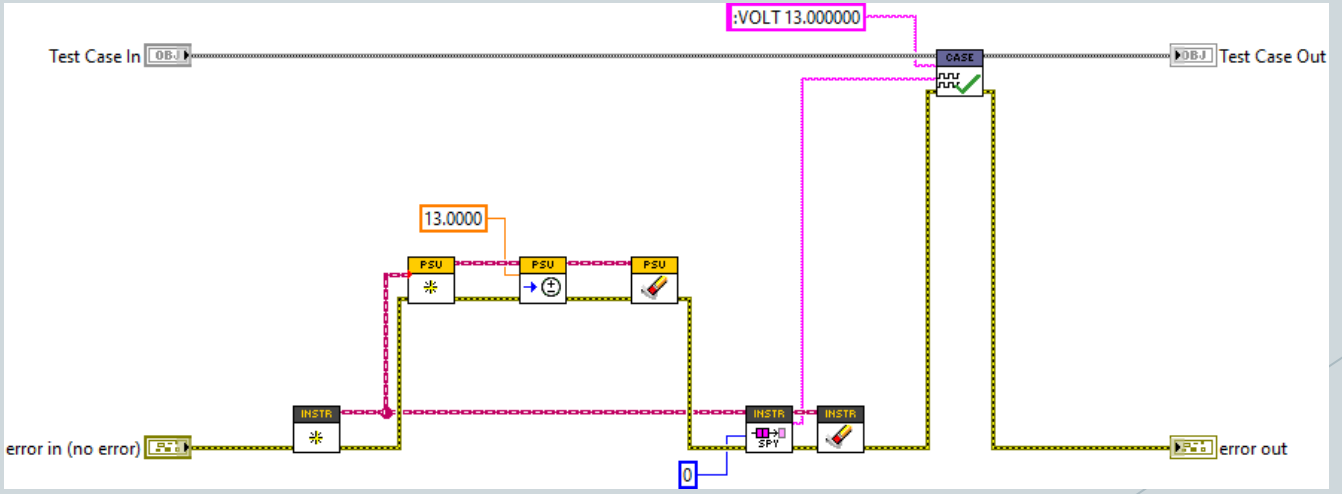




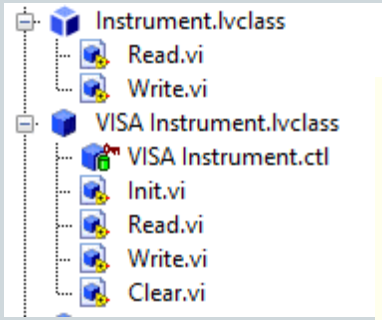
...faking it...

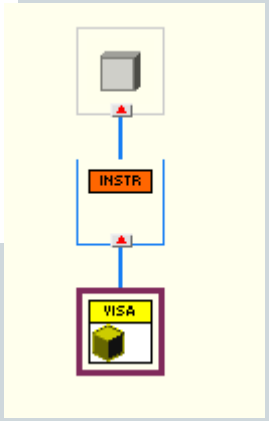

Instrument.lvclass
Read.vi
Write.vi
Mock Instrument.lvclass
Mock Instrument.ctl
Spy Write Data.vi
Init.vi
Read.vi
Write.vi
Clear.vi

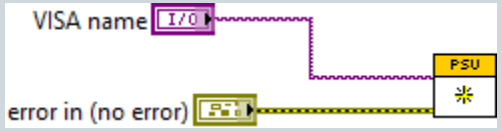

Mock Instrument Interface Out
Response
error in (no error)
Mock Instrument Interface out
Mock Instrument Interface in
Data
Mock Instrument Interface out
Mock Instrument Interface in
Data Written to Mock
Mock Instrument Interface out
Mock Instrument Interface in
Data Written to Mock
timeout in ms (-1)
Data
timed out?
error in (no error)
error out

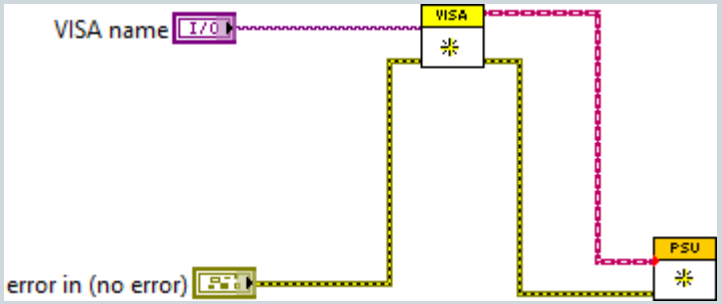

:VOLT 13.000000
Test Case In
Test Case Out
13.0000
PSU
PSU
PSU
INSTR
INSTR
INSTR
SPY
0
error in (no error)
error out

… and putting VISA back in

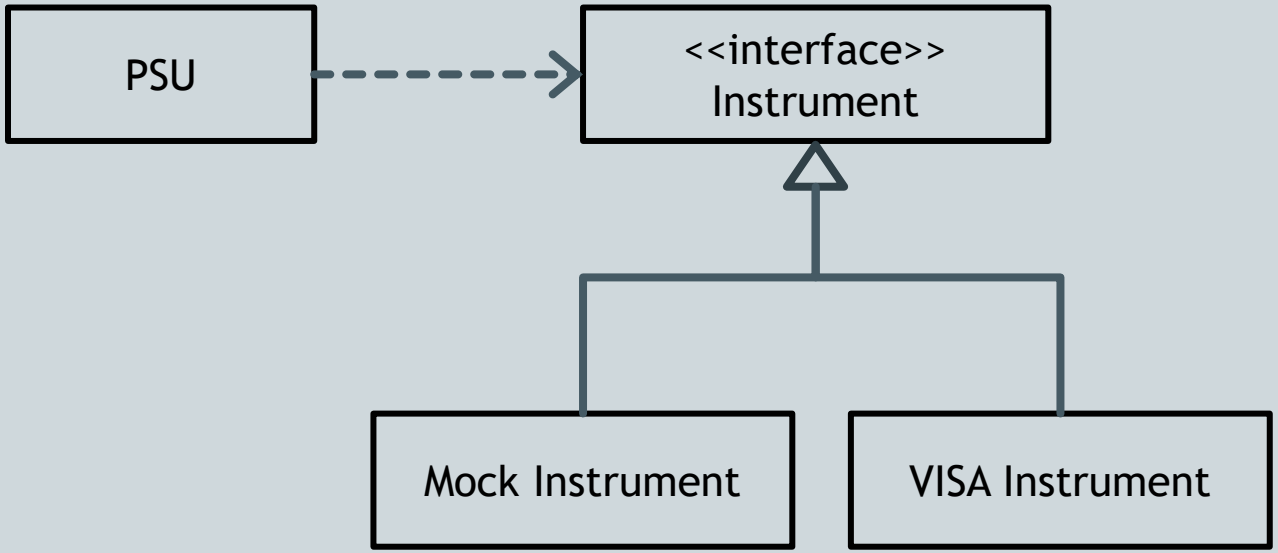




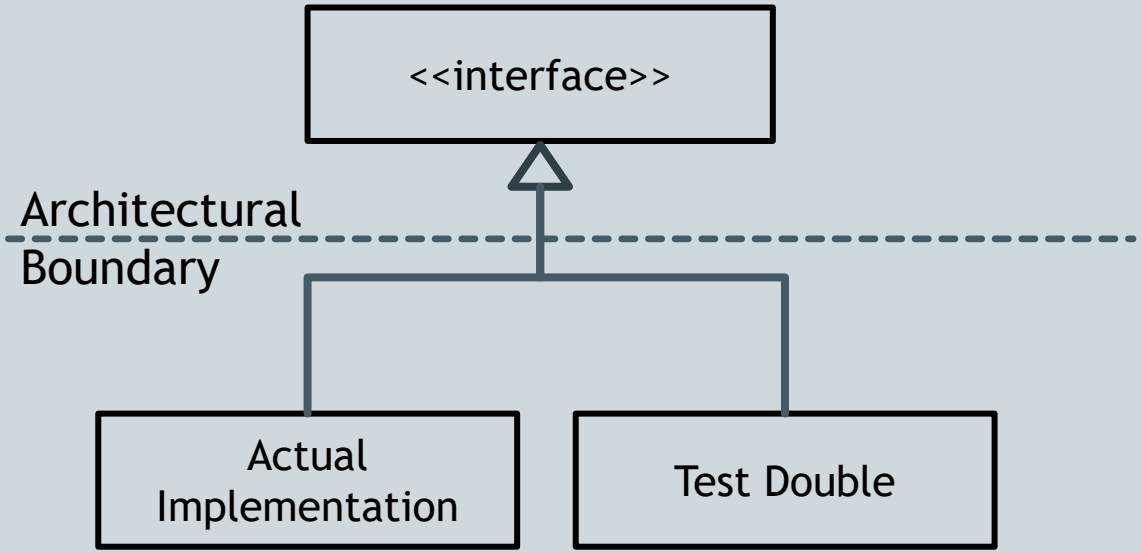




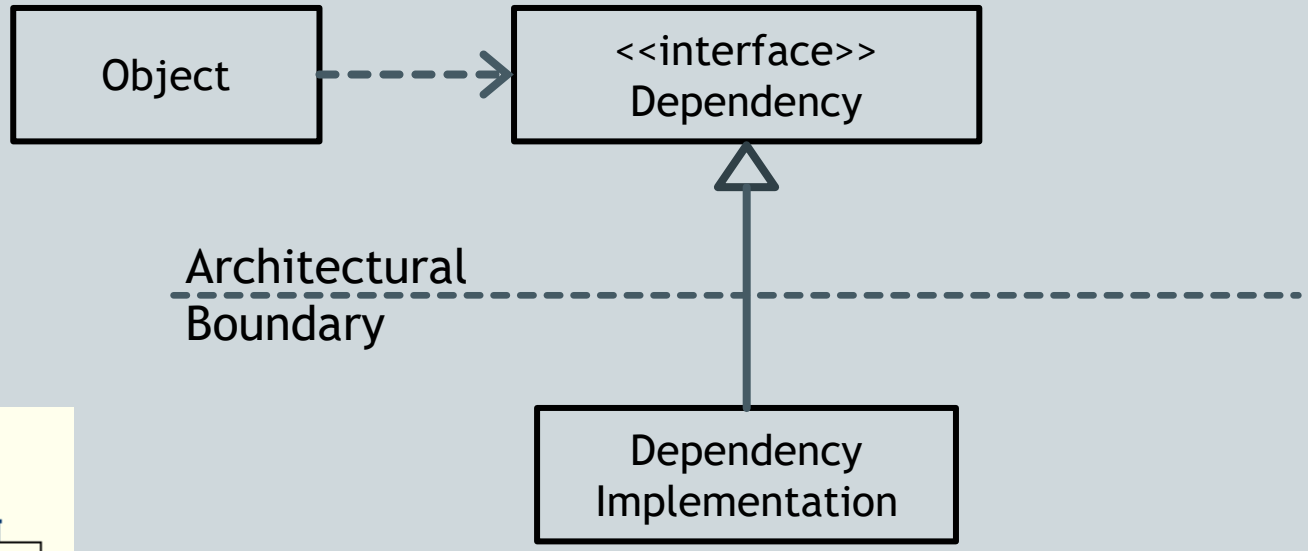
Example: Low Level Hardware

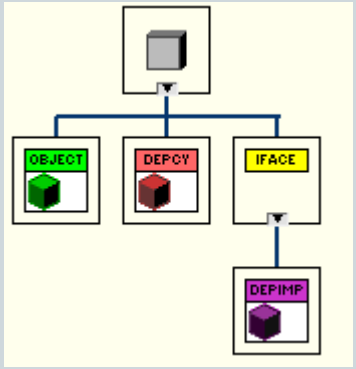


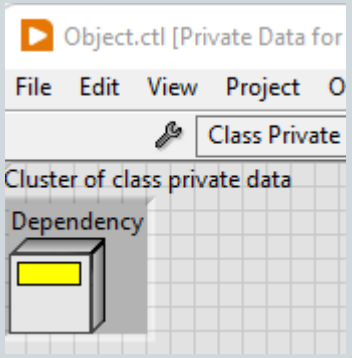
Test Doubles and Mock Objects

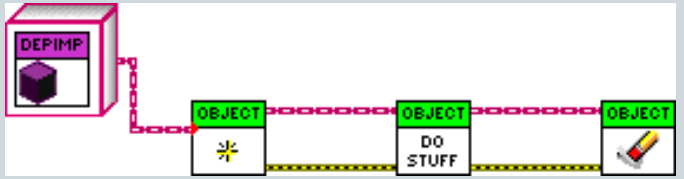


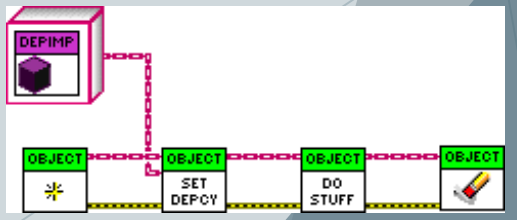
Dependency Injection



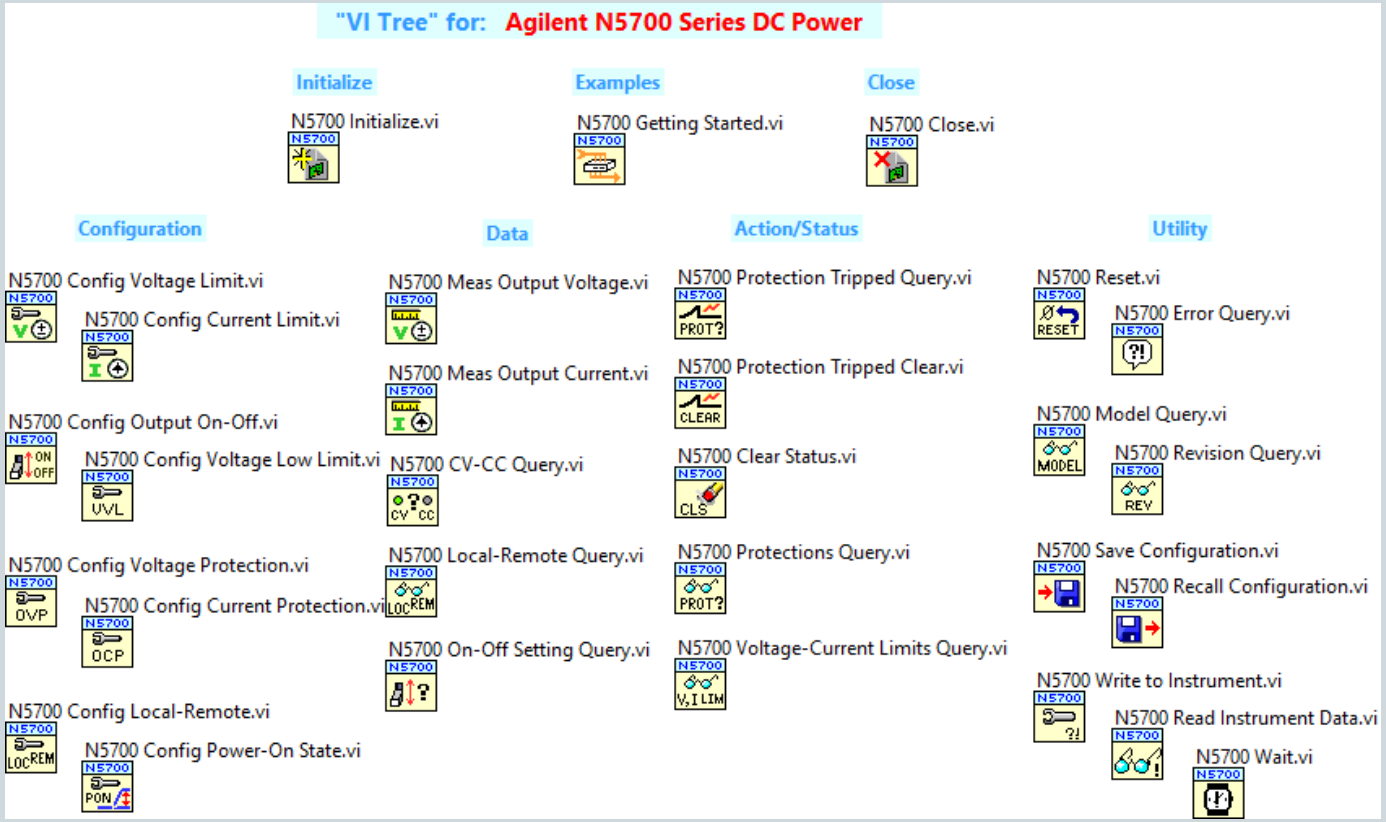




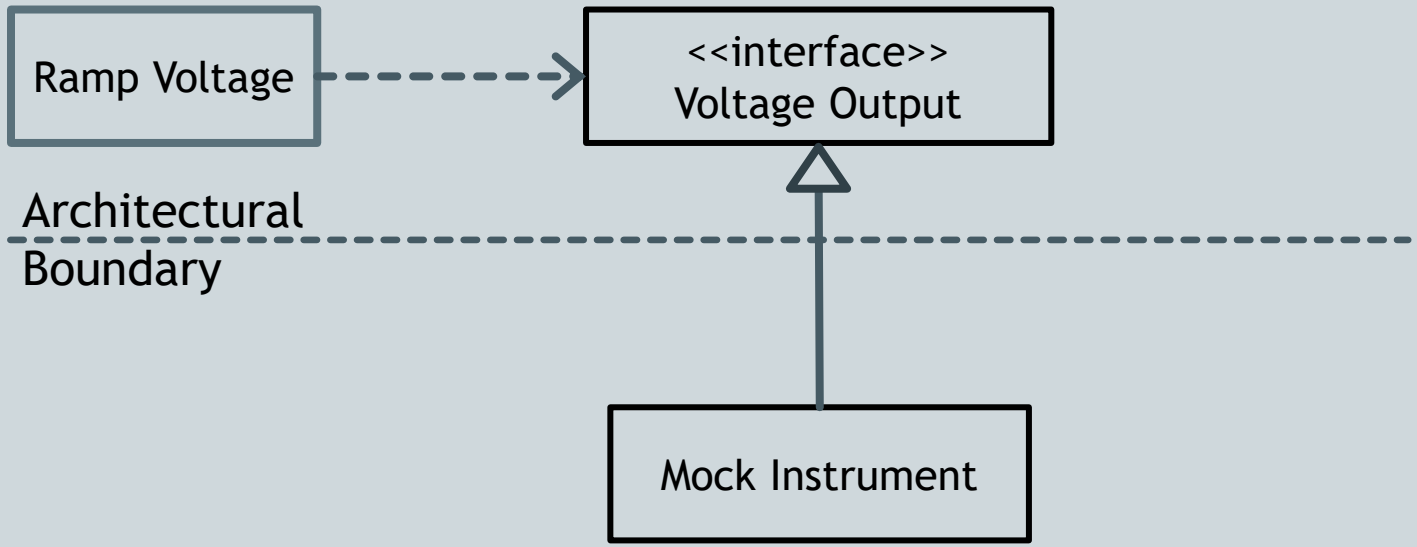




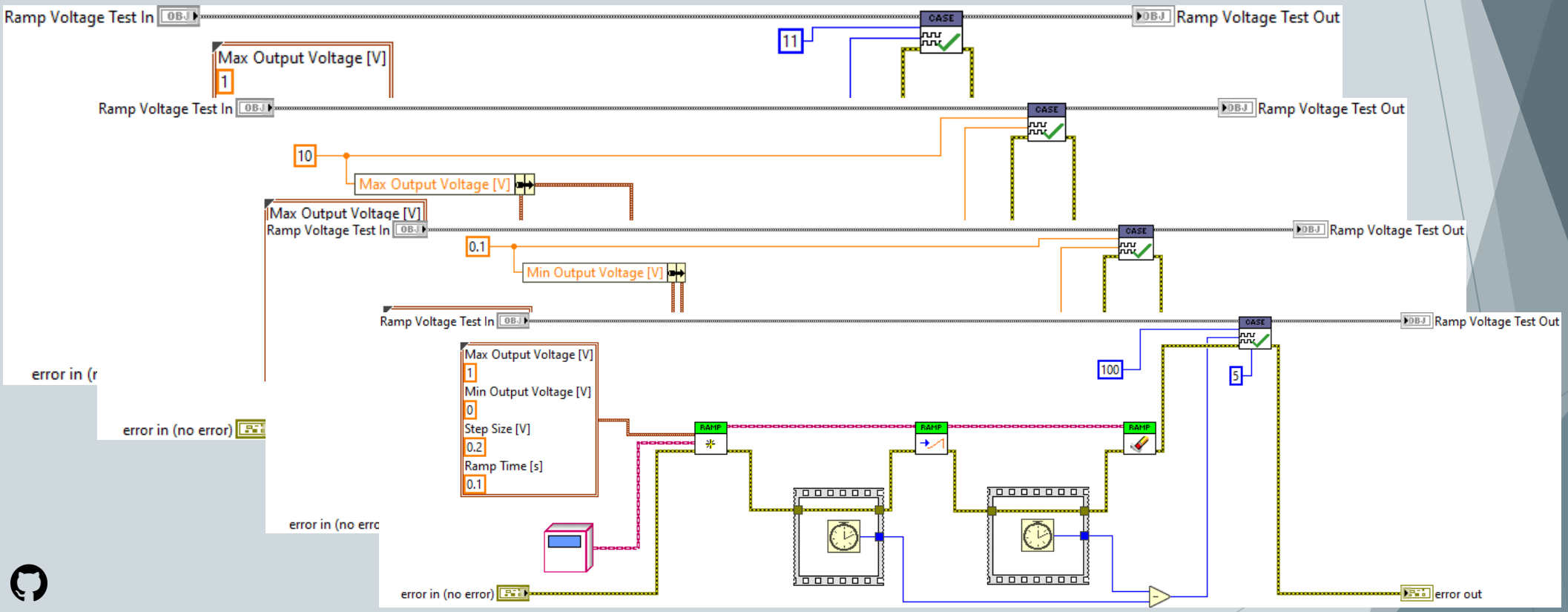
Example: High Level Hardware Drivers



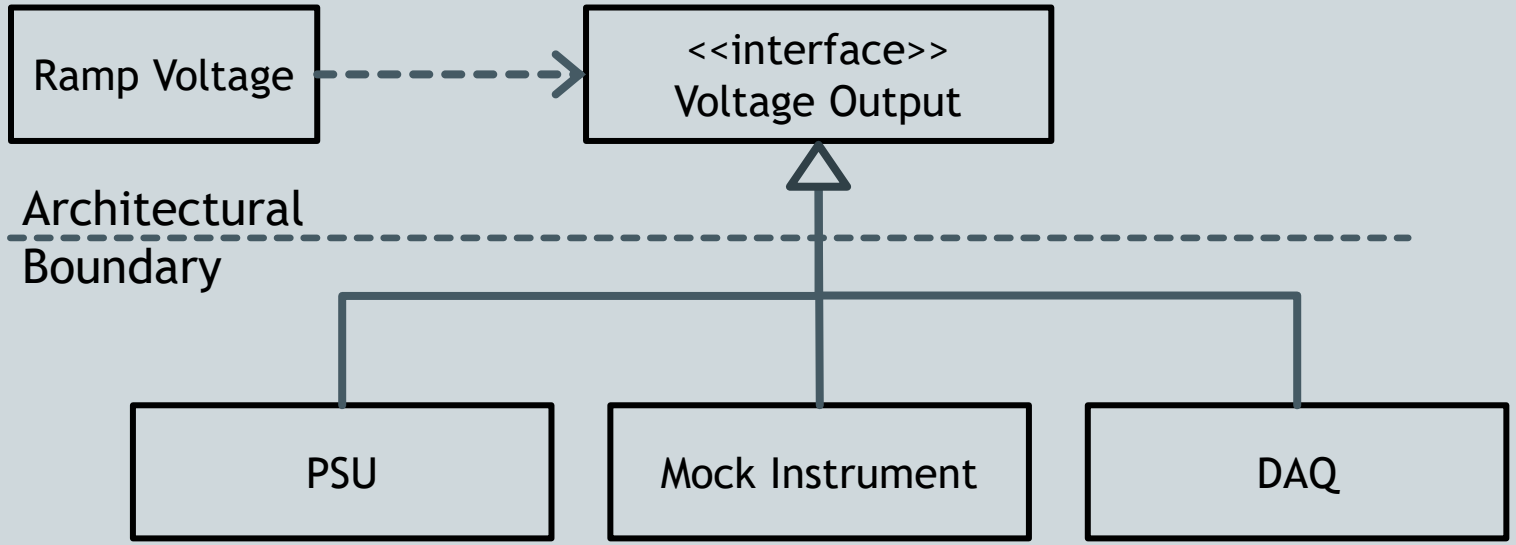
Higher Level Hardware



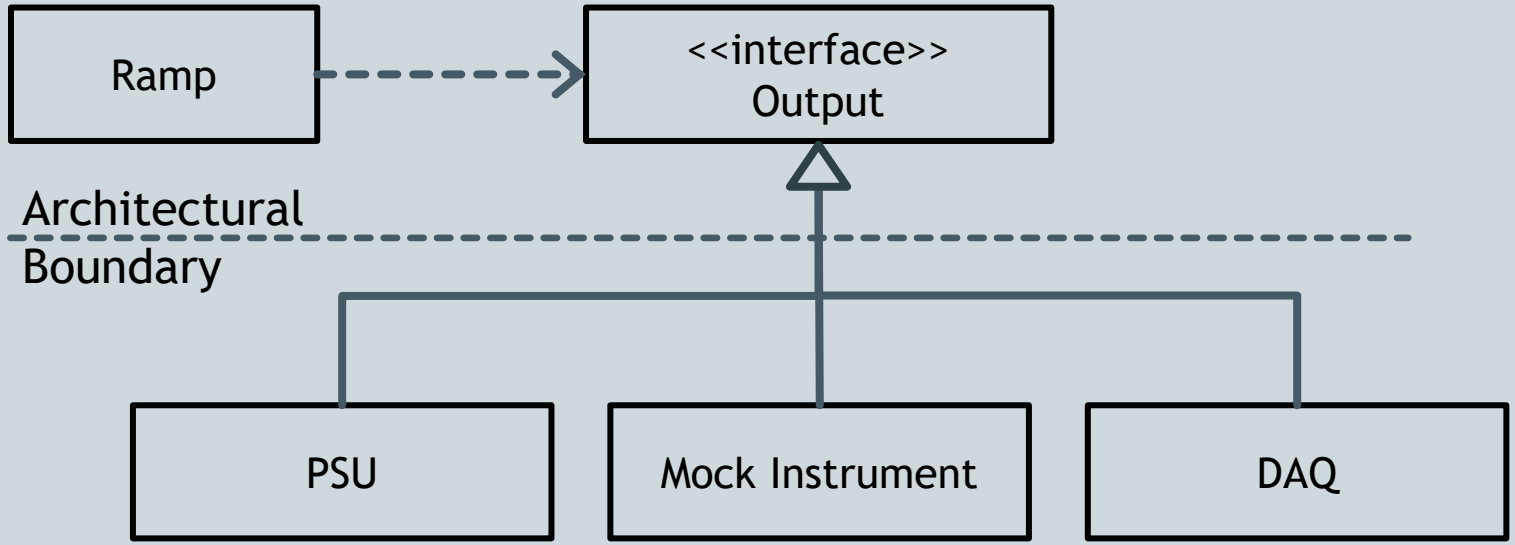
Testing Voltage Ramp



Higher Level Hardware



Level up!


Ramp
<<interface>>
Output
Architectural
Boundary
PSU
Mock Instrument
DAQ

Legacy Code

- “Legacy code is simply code without tests”
  - Michael C. Feathers, *Working Effectively with Legacy Code*
- “The Legacy Code Dilemma: When we change code, we should have tests in place. To put tests in place, we often have to change the code.”
- What have worked for me:
  - Always test new code
  - Gradually add tests to existing code
    - When refactoring
    - When changing
    - When fixing

Some useful practices

1. Keep user interfaces “dumb”
2. Don’t mary a framework
3. Only test private VIs through public interface
4. Use different compile and commit suites if test time gets painful
5. Test Behavior – not implementation
6. Always see test fail
7. “No production code before writing tests” – I often start with APIs
8. Avoid writing more than one test at a time
9. I rarely use manual Descriptions in assertions
10. Avoid testing more than one thing per test
11. Avoid complex setup/teardown
12. Running tests in parallel is useful
13. Using files for tests is ok – don’t prematurely optimize for speed