EUROPEAN
SPALLATION
SOURCE

Technical debt in the Mantid codebase

Simon Heybrock
simon.heybrock@esss.se

European Spallation Source

Outline

Goal

- Raise awareness for technical debt, quality, and resulting issues
- Use collective knowledge in the room:
    - Identify key problem areas in different parts of Mantid
    - Evaluate and prioritize
    - Decide on how to move forward?

Agenda

- Introduction with examples (Simon): Instrument code, code metrics, code duplication, tests.
- Group according to Mantid area (but include at least one outsider?)? Discuss (priorities, vague time estimate, benefit?).
- Regroup, presentation of group results. Plan?

Code metrics

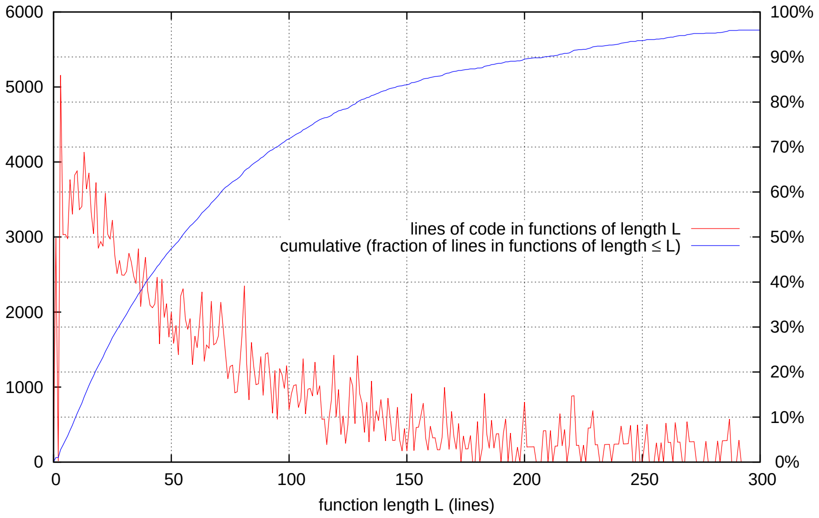
Why?

- We all have a feeling on how good or bad the code in certain areas of Mantid is. . .
- . . . but can we quantify it?

How?

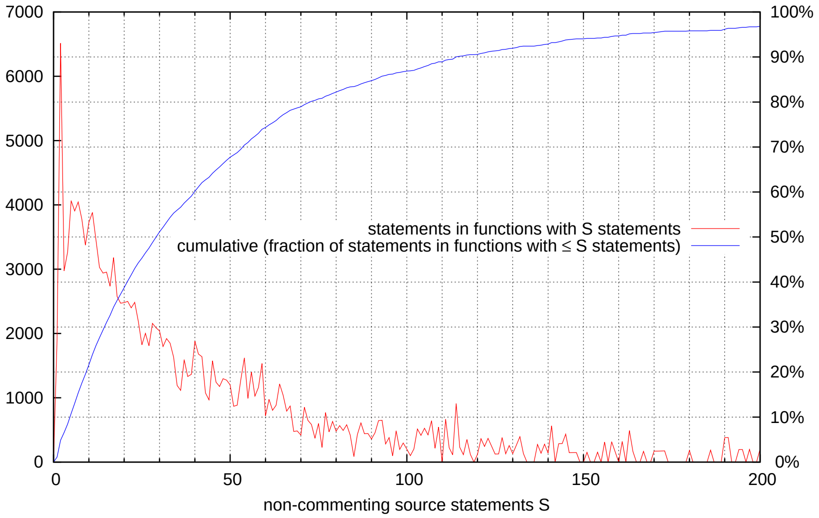
- Looked only at `Framework/`
- Used `OCLint` to gather some metrics

Lines of code in functions



- 52% of our code is in functions $\geq$ 50 lines

Number of statements in functions

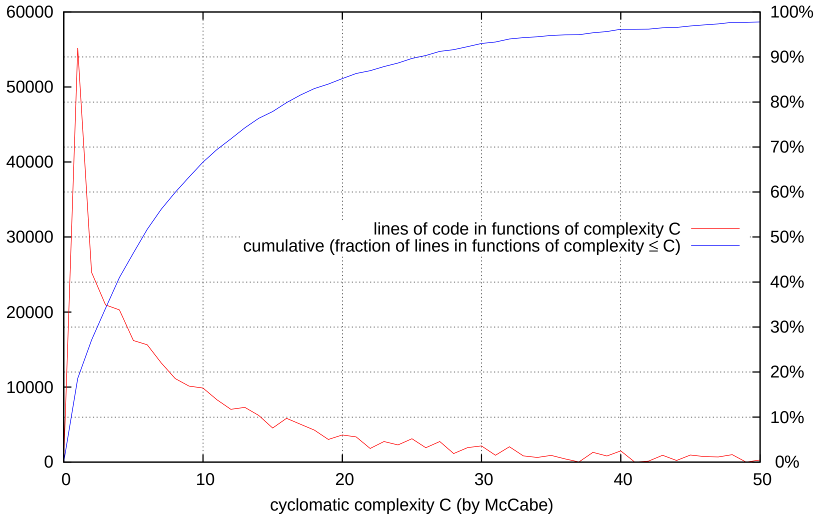


- 49% of our code is in functions $\geq$ 30 statements

Why complexity metrics

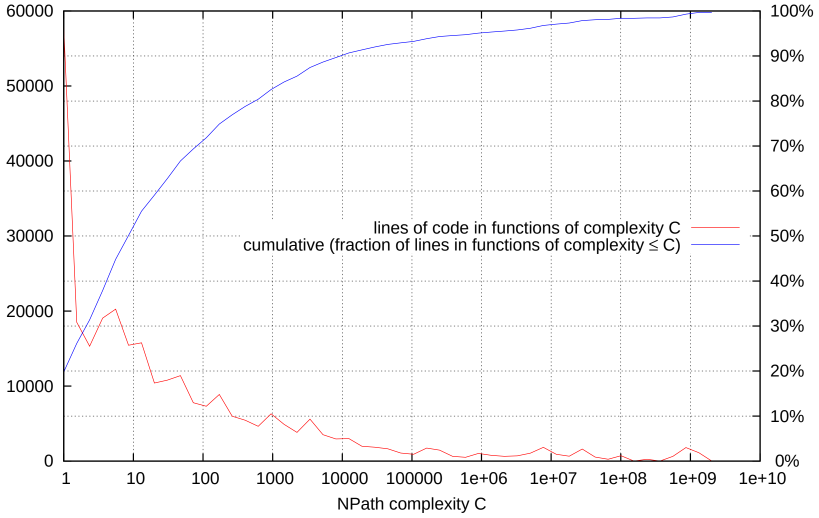
- Complex code obviously harder to understand
- Complex code harder or impossible to test:
  branch coverage $\leq$ cyclomatic complexity $\leq$ number of paths

Cyclomatic complexity — indicator for branch coverage



- 33% of our code has cyclomatic complexity ≥ 10

Complexity (2/2)



- 25% of our code has NPath complexity $\geq$ 200

Code duplication (1/3)

```
class SomeJacobian : public API::Jacobian {
public:
  SomeJacobian(size_t nData, size_t nParams)
      : m_nParams(nParams), m_data(nData * nParams) {}
  void set(size_t iY, size_t iP, double value) {
    m_data[iY * m_nParams + iP] = value;
  }
  double get(size_t iY, size_t iP) {
    return m_data[iY * m_nParams + iP];
  }
private:
  size_t m_nParams;
  std::vector<double> m_data;
};
```

- 22 implementations (9 in tests), all more or less similar
- Reason: Interface defined in different module?

Code duplication (2/3)

```cpp
// Make a brand new EventWorkspace
outputWS = boost::dynamic_pointer_cast<EventWorkspace>(
    API::WorkspaceFactory::Instance().create(
        "EventWorkspace", inputWS->getNumberHistograms(),
        2, 1));
// Copy geometry over.
API::WorkspaceFactory::Instance().initializeFromParent(
    inputWS, outputWS, false);
// You need to copy over the data as well.
outputWS->copyDataFrom((*inputWS));
```

- Same in plenty of algorithms ("Make a brand new EventWorkspace" shows up 28 times)
- Workspaces are difficult to deal with?
- Algorithms do too much, there is very little shared code
- boiler-plate code hides the important bits

Code duplication (3/3)

```
PARALLEL_FOR2(inputWS, outputWS)
for (size_t i = 0; i < size_t(numberOfSpectra); ++i) {
  PARALLEL_START_INTERUPT_REGION
  // CODE CODE CODE CODE CODE CODE CODE CODE
  MantidVec &yOut = outputWS->dataY(i);
  MantidVec &eOut = outputWS->dataE(i);
  const MantidVec &xIn = inputWS->readX(i);
  const MantidVec &yIn = inputWS->readY(i);
  const MantidVec &eIn = inputWS->readE(i);
  // CODE CODE CODE CODE CODE CODE CODE CODE
  prog.report();
  PARALLEL_END_INTERUPT_REGION
} // end for i
PARALLEL_CHECK_INTERUPT_REGION
```

- Highly duplicate: in almost every MatrixWorkspace algorithm
- Programmers need to know and think about threading
- We lose flexibility (fixed threading model)
- Every bit of code has to understand what a workspace is

How good are our tests?

- Coverage: 68%
- Some areas are well tested

Example 1 (Algorithms)

- algorithm `exec()` $+$ helper functions $\sim$ 400 lines of code
- unit tests total $\sim$ 400 lines of code
- 83% coverage
- modified only code with 100% coverage
- could insert 16 errors, unit tests and doc test still pass
- unit tests for some other algorithms fail

How good are our tests?

Example 2 (Crystal)

- algorithm `exec()` $\sim$ 200 lines of code
- unit tests total $\sim$ 100 lines of code
- 91% coverage
- modified only code with 100% coverage
- could insert 79 errors, unit tests and doc test still pass
- unit tests for some other algorithms fail

- swap `true` and `false`
- change loop ranges
- delete lines
- change operators
- misplace parenthesis

Instrument / Geometry

Problems

- Nearly impossible to add new features (e.g., moving instruments)
- Very slow (e.g., close to 30% of total time in a SANS reduction)
- Difficult to maintain, test, and debug

- Will be redesigned
- Currently: Gathering requirements

Instrument / Geometry

Trivial example for the issues:

- We have: Workspace
- We want: position of detector *i*

How?

```
Geometry::IDetector_const_sptr det = ws->getDetector(i);
Kernel::V3D pos = det->getPos()
```

“Get spectrum position” — internals

1. vector lookup: workspace-index → `ISpectrum *`
2. memory allocation: `Instrument`
    1. copy/allocate 10+ members
    2. memory allocation: `std::string` (copy)
    3. memory allocation: `ReferenceFrame`
3. find in (large) map: detector-ID to pointer-to-detector
4. memory allocation: `Detector`
5. find in (small) map: detector in `ParameterMap`?
    1. iterate over parameters for detector
    2. string comparisons
6. lock mutex
7. find in (large) map: cached location
8. lock mutex
9. find in (large) map: cached rotation
10. rotate and translate
11. memory deallocation: Detector
12. memory deallocation: Instrument

Too much responsibility

`MatrixWorkspace`

- neutron data (histogram, events)
- instrument
- logs
- index mapping
- masking
- monitors
- load/save Nexus
- MD geometry methods
- image methods

545 lines header + 2077 lines source

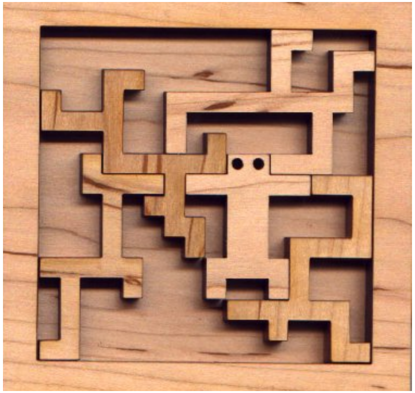
`MatrixWorkspace`

Consequences

- More bugs
- Tightly coupled code
- Is workspace the right abstraction for everything? Many algorithm do not need to know about all aspects of if.

Algorithms

What it was meant to be. . .



. . . and how it turned out.

Algorithms

- New algorithms similar but not identical to existing ones
  - Algorithms do too much ⇒ too specific and cannot be reused
- Lots of functionality duplicated in many algorithms
  - Generic bits of code are locked away inside algorithms
  - Duplicated efforts, wasted time, frustration, more bugs
  - Duplication on two fronts:
    - technical (create output workspace, loop over spectra, threading)
    - scientific (neutron science or instrument related, requires domain knowledge)
- Hard to maintain specialized code — algorithms end up unused, unmaintained, and forgotten.

Further topics

- Range of available property types too small
  - Adding types is very hard
  - Causes reliance on ADS (especially in connection with `GroupWorkspace`)
- Algorithm naming conventions (see Andrei's lightning talk)
- Singletons and dynamic factories
  - Global state! Use dependency injection instead?
  - See Owen's lightning talk
- "`ApplicationWindow` is a mess"
- `MantidQt/CustomInterfaces` hard to maintain
- `scripts/` tend to break, we do not know about it, fragile
- 3 different indices in many places of the code (detector ID, workspace index, spectrum number), should all of our code depend on spectrum number, which is more for displaying data to users?

Breakout Session

Split into groups

Framework, MantidPlot, Vates, . . .

Discuss

- Where do you see issues with technical debt in that area?
  - Within the area you are discussing?
  - In your area, but due to issues in other areas?
- Do you experience problems due to that? Which ones?
- Is fixing easy to get started or are there major issues?

Rejoin

Short presentation / discussion of results.