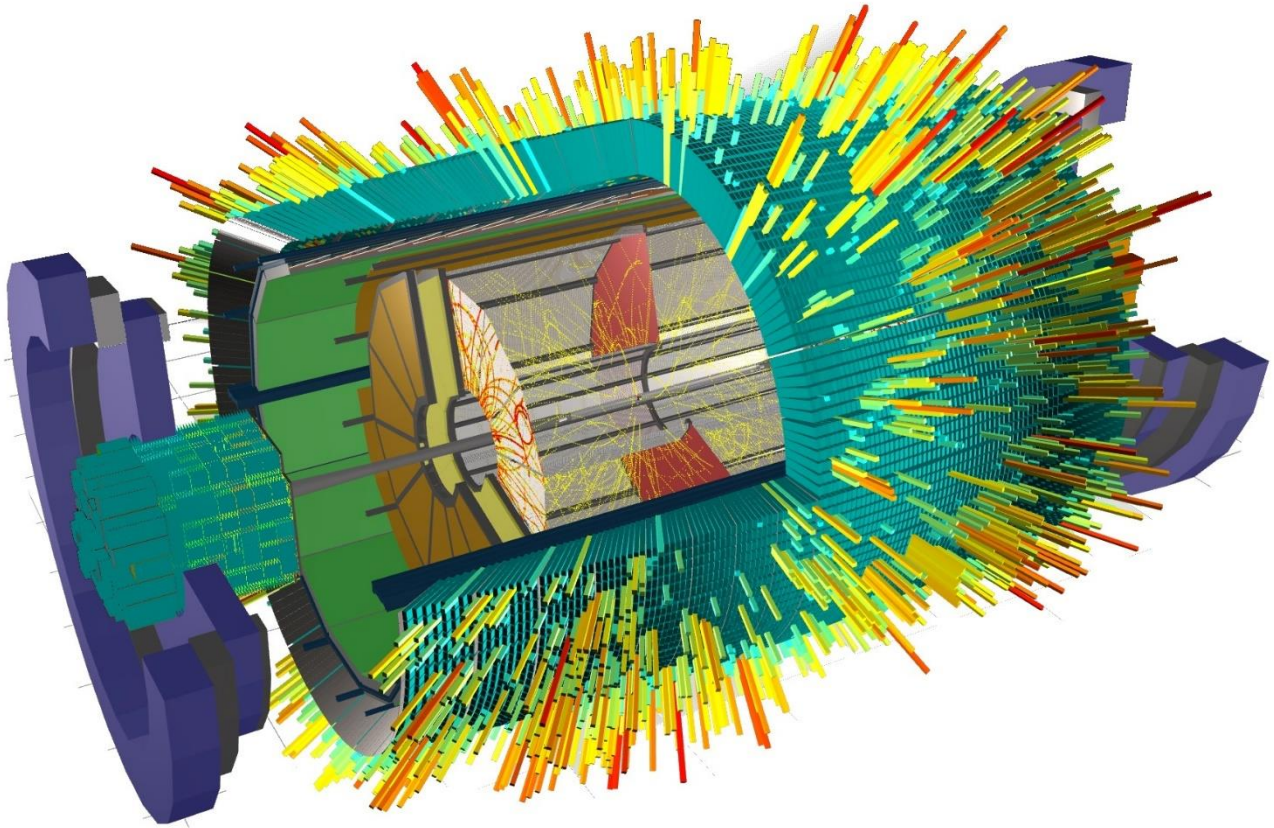
Performance Analysis and Optimization of MPDRoot

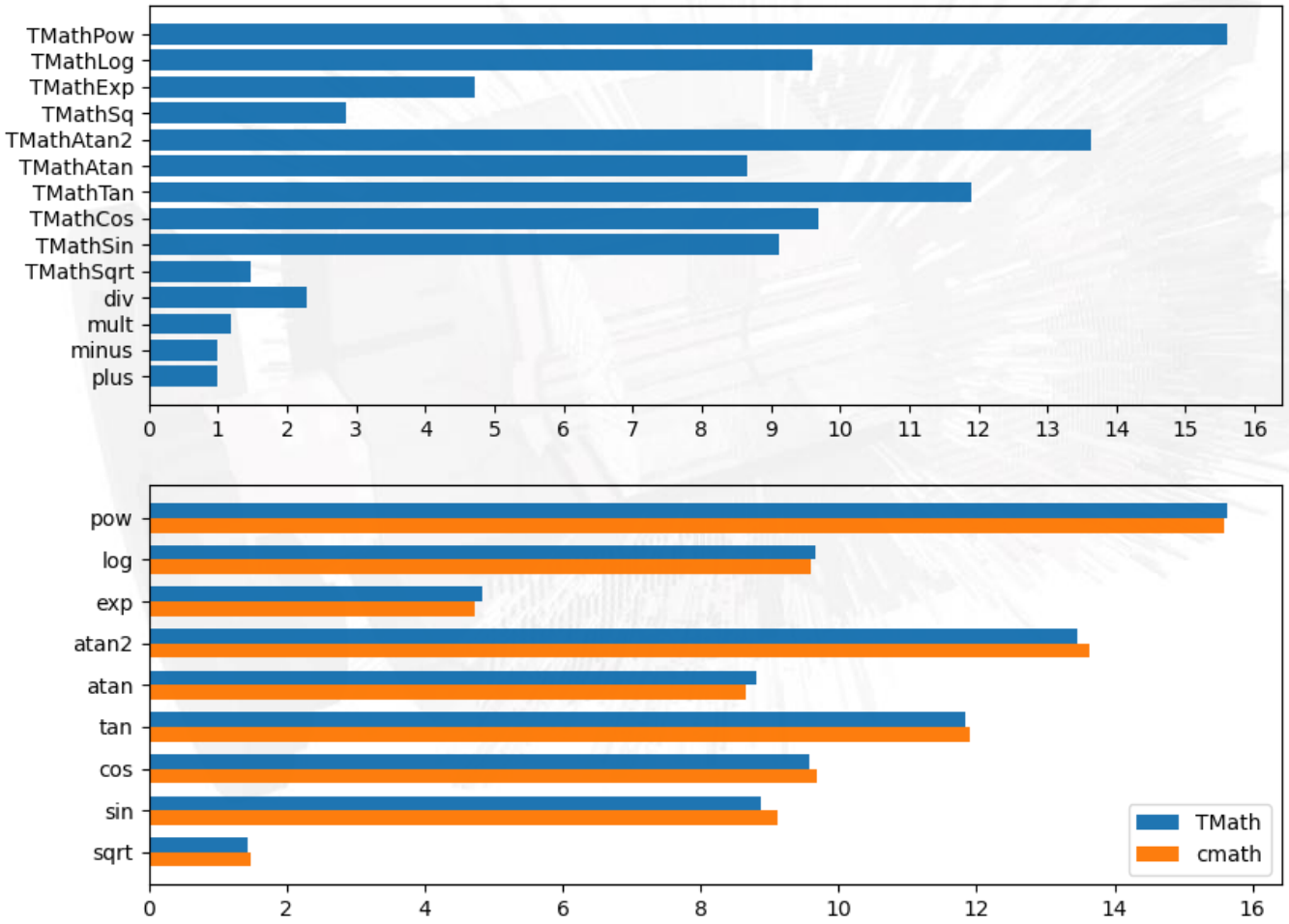
Buša J., Hnatič S., Rogachevsky O.
LIT JINR, LHEP JINR



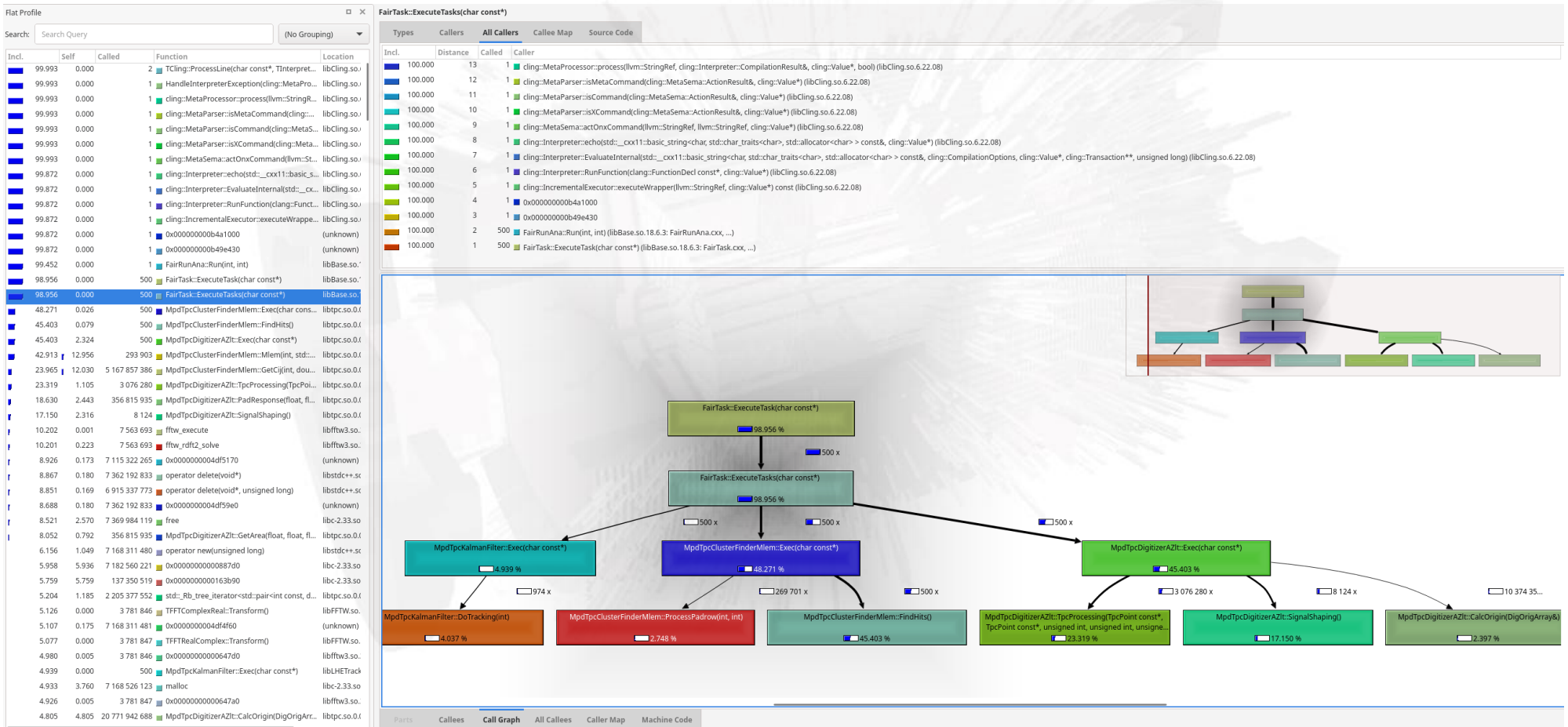
OUTLINE

- Benchmarking TMath (cmath) in MPDRoot
- Profiling MPDRoot – Instructions vs Timings
- TMath Optimization
- Reducing Calls
- Premature Optimization / When to optimize
- Getting the SD process under control
- Improving code quality
- Testing in development
- Testing environment

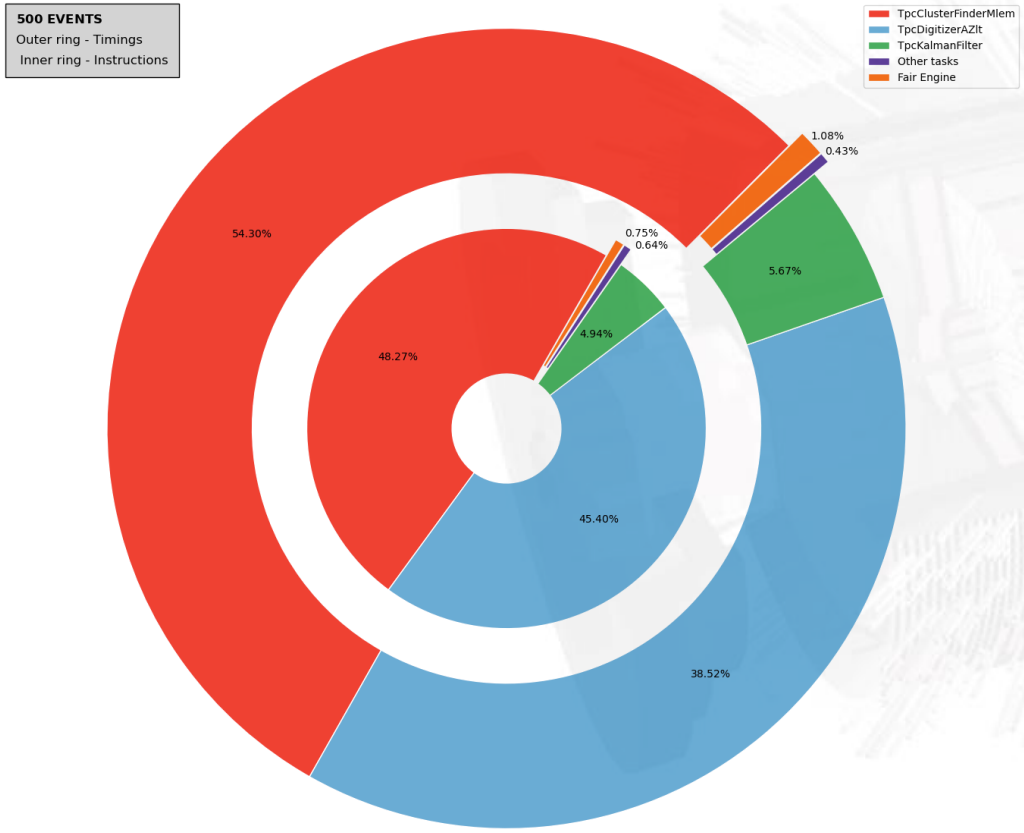
TMATH BENCHMARKS

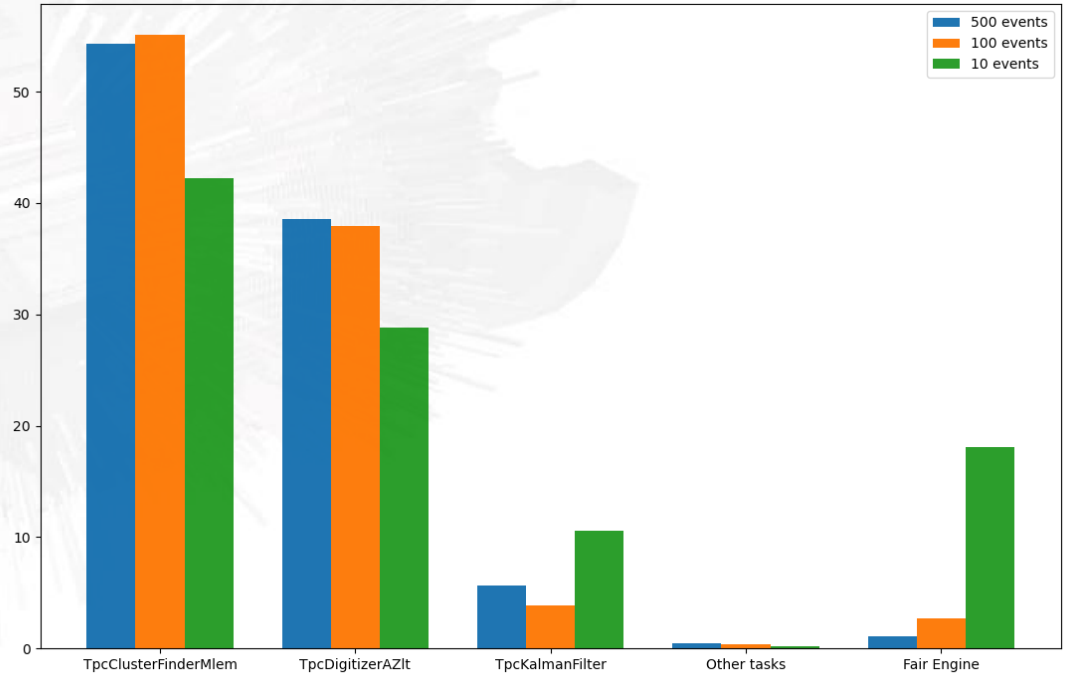


INSTRUCTION PROFILING

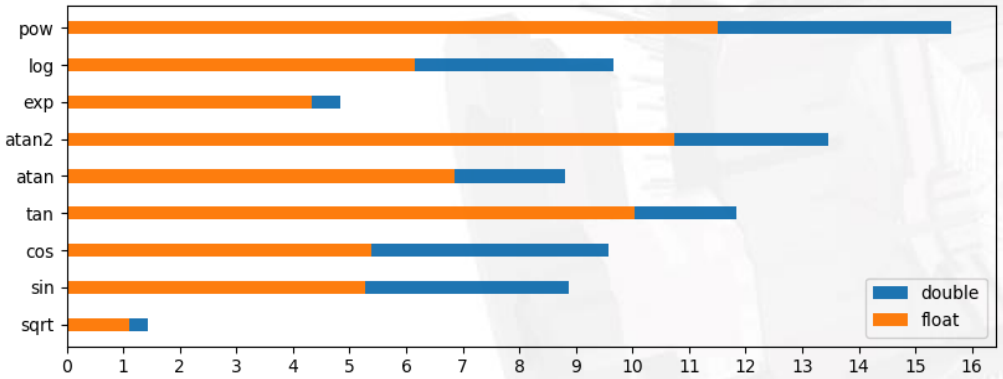


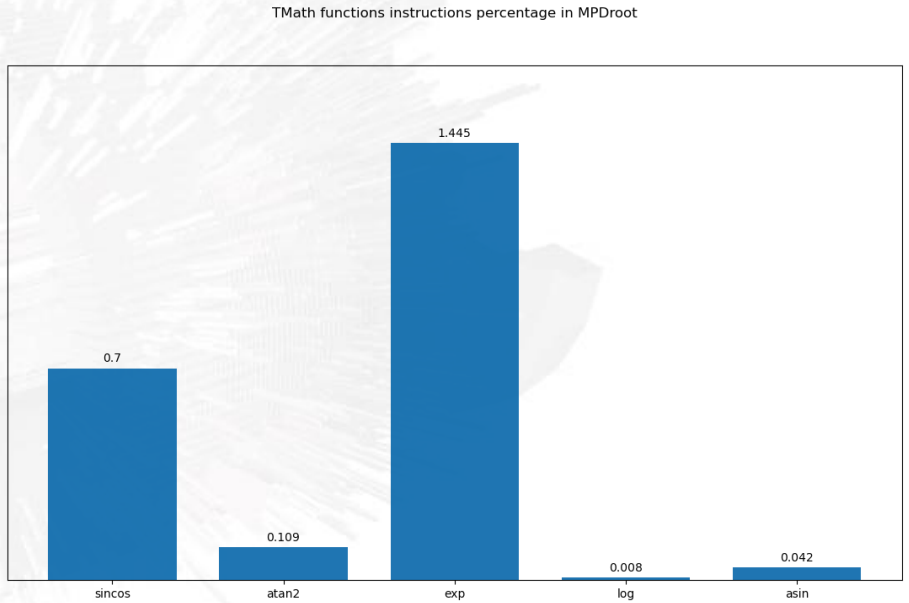
TIME PERFORMANCE





OPTIMIZATION TMATH





REDUCING CALLS

- Algorithm logic improvement
- Inlining (inline, flatten)

"Q: Do inline functions improve performance?
A: Yes and no. Sometimes. Maybe."
isocpp.org FAQ

Example from MPDRoot codebase

	% of Instructions out of total	Instructions per call	% of calls inlined	Task speedup	Total speedup
CalcOrigin (Digitizer Task)	4.8	18	100	4.2%	1.9%
GetCij (ClusterFinder Task)	12	380	90	-1.2%	-6.3%

PREMATURE OPTIMIZATION

"Premature optimization is the root of all evil"

Donald Knuth (TeX), Tony Hoare (quicksort)

Rules of optimization:
Rule 1: Don't do it.
Rule 2 (for experts only): Don't do it yet.

WHEN TO OPTIMIZE

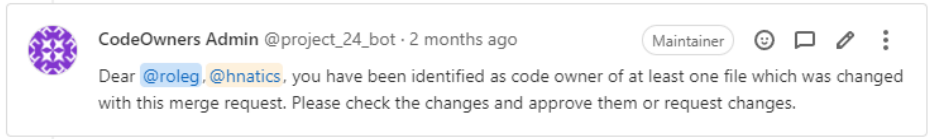
Single responsibility principle
Open/Closed principle
> **Software elements (modules, classes, functions etc) should be open for extension, but closed for modification**

Liskov Substitution principle
Interface Segregation principle
Dependency Inversion principle

GETTING THE SD PROCESS UNDER CONTROL

"The art of programming (software development) is the art of organizing complexity,
of mastering multitude and avoiding its bastard chaos as effectively as possible."
E. Dijkstra

- Code Ownership within GitLab - already developed



CODEOWNERS FILE - detector owner(s)
- reconstruction owners
- system scripts owners

Ownership of other directories to be assigned in the future – generators, geometry, physics analysis, QA, macros, system config

Benefits - code review by competent developers
- no arbitrary merges
- less trash code

IMPROVING CODE QUALITY

“We rarely put defects directly, instead we set up conditions that trick us into putting in defects later”

How to prevent this?

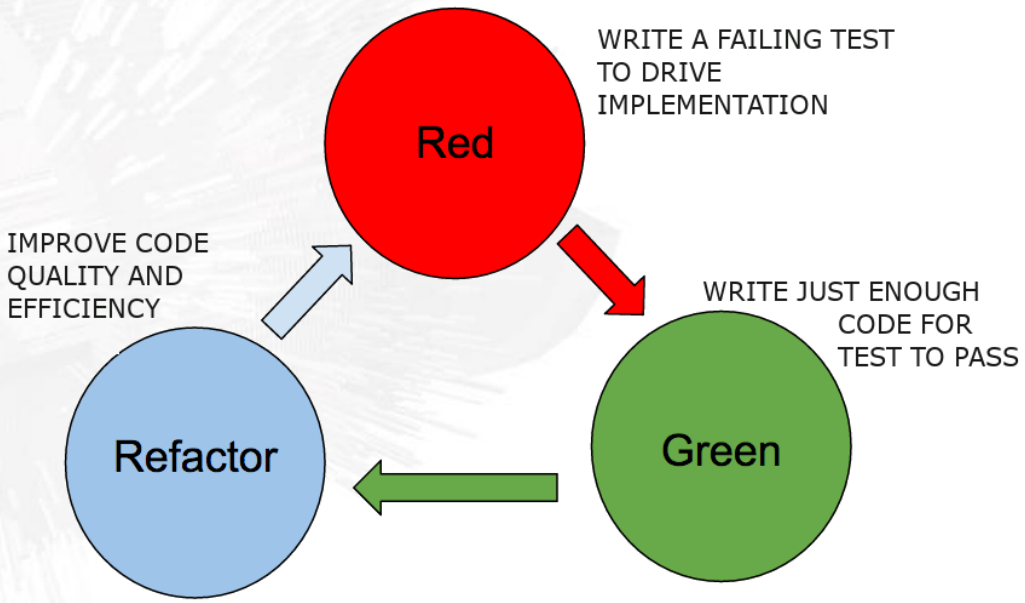
- Testing environment – in progress (critical, the very first thing to do)
- Removal of unused detectors – nearly finished (one left)
- Directory restructuring to reflect multilevel hierarchy - in progress
- Dead code identification & cleanup – to be done
- Automatic code formatting – to be done
- Documentation – in progress

TESTING IN DEVELOPMENT (NEW CODE)

EFFICIENT DEVELOPMENT CONSISTS OF

1. Define module's (algorithm) external behavior
2. Develop working prototype
3. Refactor

- improve code structure
  without changing its external behavior



Achieved by WRITING TESTS TO

- Force proper modularization
- Force proper abstraction
- Force clear separation into OO and procedural parts

TESTING ENVIRONMENT

CONSTRUCTION TESTING ACTIVITIES

Single step testing
- Stepping through the code with the debugger

Unit testing
- Smallest thing that can be automatically tested, like method (subroutine)

Component testing
- Testing of aggregate of units
- Automated, often with use of mocks, stubs, fixtures

Bench testing
- Testing of the component in the entire system

TESTING ENVIRONMENT (LEGACY CODE)

COMMON STRATEGIES TO ADD TESTS

- Error-prone areas of the system
- High-fan parts, afferent coupling metrics
- Critical modules
- Often changed areas
- Any time code is modified

TECHNICAL REALIZATION

- Toolset – mocks, stubs, fixtures – dependency injection
- Integrated with build system / executed after each build
- Ability to run on demand with custom parameters
- Incremental refactoring to increase testability (decoupling, interfaces)
- Root cause analysis

Thank You !

Q & A