Methodical Accelerator Design

Project Overview

Laurent Deniau

CERN-BE/ABP

31th May 2011

Preliminaries

- Education (not a physicist)
  - engineer in computer science (≠ IT) and signal processing (1992)
  - master (DEA) in signal processing and sensors (1992)
  - PhD in signal and image processing (1997)
- Joined LHC-MTA in Feb 1997 (➪ AT-MTM ➪ AT-MEI ➪ TE-MSC)
  - magnetic measurements
  - software for analyzing measurements of LHC magnets (servers, web)
  - modeling superconducting magnets behaviors for LHC operation
  - Field Description for LHC (FiDeL, scientific secretary 2004-2011)
- Joined BE-ABP in May 2011
  - new custodian of MAD-X (50%)
    - support and evolution (midterm & longterm) of MAD-X
  - beam dynamic physics (50%)
    - must learn first, improve MAD-X physics, participate to accelerator design
    - no knowledge of beam dynamics, Hamiltonian mechanics or symplectic geometry!

Part I
The Present

Project Overview

- MAD-8 (80’s - 90’s)
  - ➡ world-wide tool for accelerator design in the 90’s (became a “standard”)
  - ➡ 4D only, but very flexible and very fast for users
  - ➡ old style of programming (very low level, no abstraction)
  - ➡ somehow limited physics (e.g. no advanced topology)
- MAD-9 (mid of 90’s - 00’s)
  - ➡ complete redesign and rewrite from scratch (organized as a library)
  - ➡ flexible and consistent (new scripting language), extensible, well structured, but slow
  - ➡ problem with the Lie Algebra approach (difficulty to compose maps)
- MAD-X (mid of 90’s - today)
  - ➡ includes a subset of MAD-8
    - same advantages (fast, flexible) but not extensible (~frozen) and not robust (buggy)
  - ➡ includes PTC-FPP
    - handle 6D and non-linearities correctly (large δp & β), compose maps ~arbitrarily
    - needed by most recent studies & future projects (e.g. PS, CLIC, LHC-HL)
  - ➡ mix of technologies (painful to maintain)

MAD-X Typical Uses

- **Batch mode** (machine simulation)
  - ➡ send a job, wait for results
  - ➡ layout, optics, twiss, matching, response matrix, ...
- **Direct mode** (physics on top of machine simulation)
  - ➡ use MAD-X as part of a more complex framework (e.g. Python or Mathematica scripts)
  - ➡ run MAD-X directly for simulations (as for batch mode)
  - ➡ iterative process for complex analysis (e.g. optimization, inverse problem)
- **Interactive mode** (machine design)
  - ➡ use MAD-X as a shell, load data & scripts, typewrite commands, etc...
  - ➡ exploration of parameters space, manual optimization, advanced uses
    - takes time to setup workspace & studies
    - unexpected *quit* results in users waste of time and frustration

The presence of many bugs weakens the users confidence
Syndrome of “the hidden part of the iceberg”

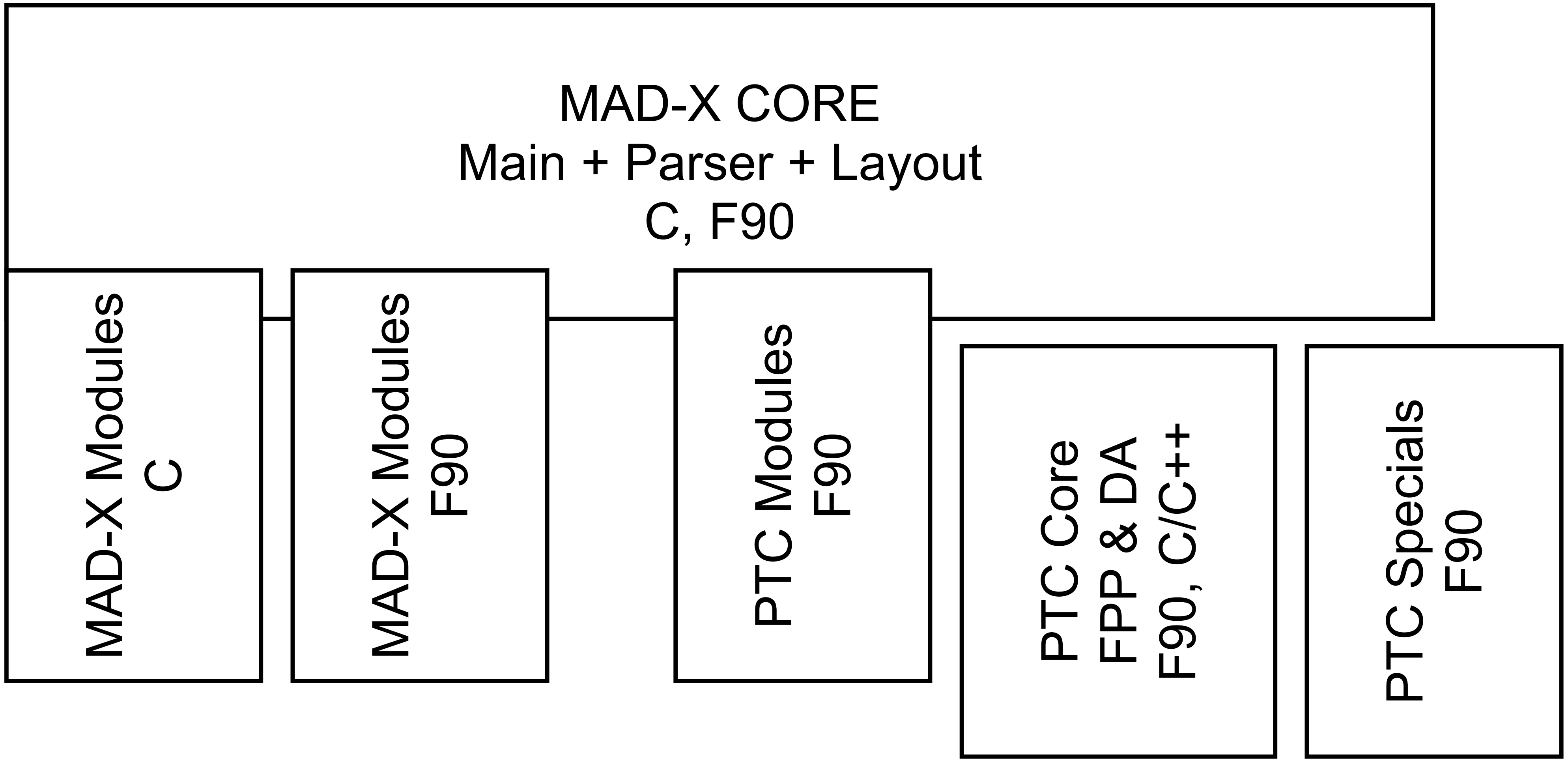
Known Pending Bugs

- This list exclude parser bugs!
- Memory corruption in Aperture module
  - ➡ well located, should be solvable
- Aperture followed by Survey command
  - ➡ observable: side effects on the layout, introduce deflections at some magnets
  - ➡ location: probably in the Aperture module
  - ➡ MAD-X 5 only
- Portability problem leading to crash or incomplete output
  - ➡ observable: DA becomes unstable, MAD stops with Fortran severe error, invalid results
  - ➡ location: PTC_TWISS (?), not identified, no idea where to start
  - ➡ differences between MAD-X version 4 and 5 and Windows vs Unix
    - maybe related to Fortran compiler, but certainly not only

MAD-X is not equipped for
tracking bugs and code validation
Syndrome of "valid output = validated code = no bug"

MAD-X Schematic Components

- Medium size project (lines of code: total 167k, 36k in C, 123k in F90)
- MAD-X core is overly complicated (very high coupling, very low cohesion)
- PTC represents 3/4 of the code, it is limited by MAD-X core (information transfer)
- Some MAD-X modules are not yet in PTC modules
- Collaboration between C and F90 is fragile (e.g. I/O)



MAD-X CORE
Main + Parser + Layout
C, F90

MAD-X Modules
C

MAD-X Modules
F90

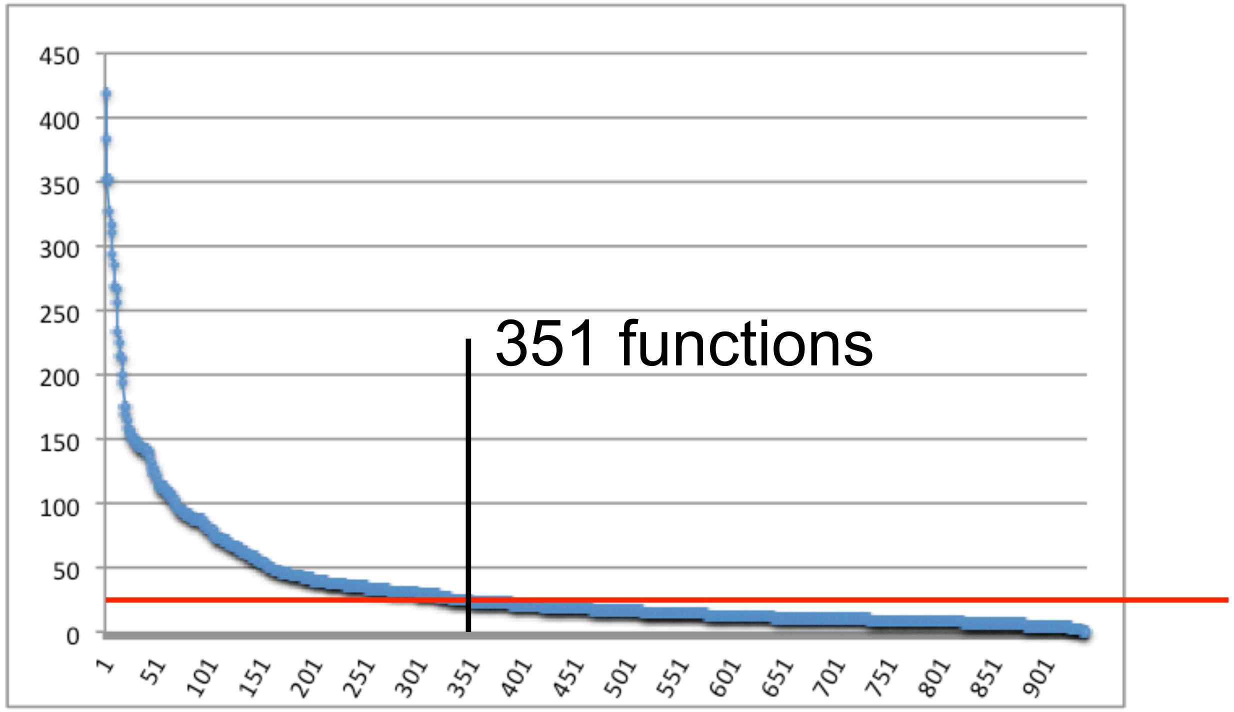
PTC Modules
F90

PTC Core
FPP & DA
F90, C/C++

PTC Specials
F90

Code Static Analysis

- C functions by number of effective lines of code



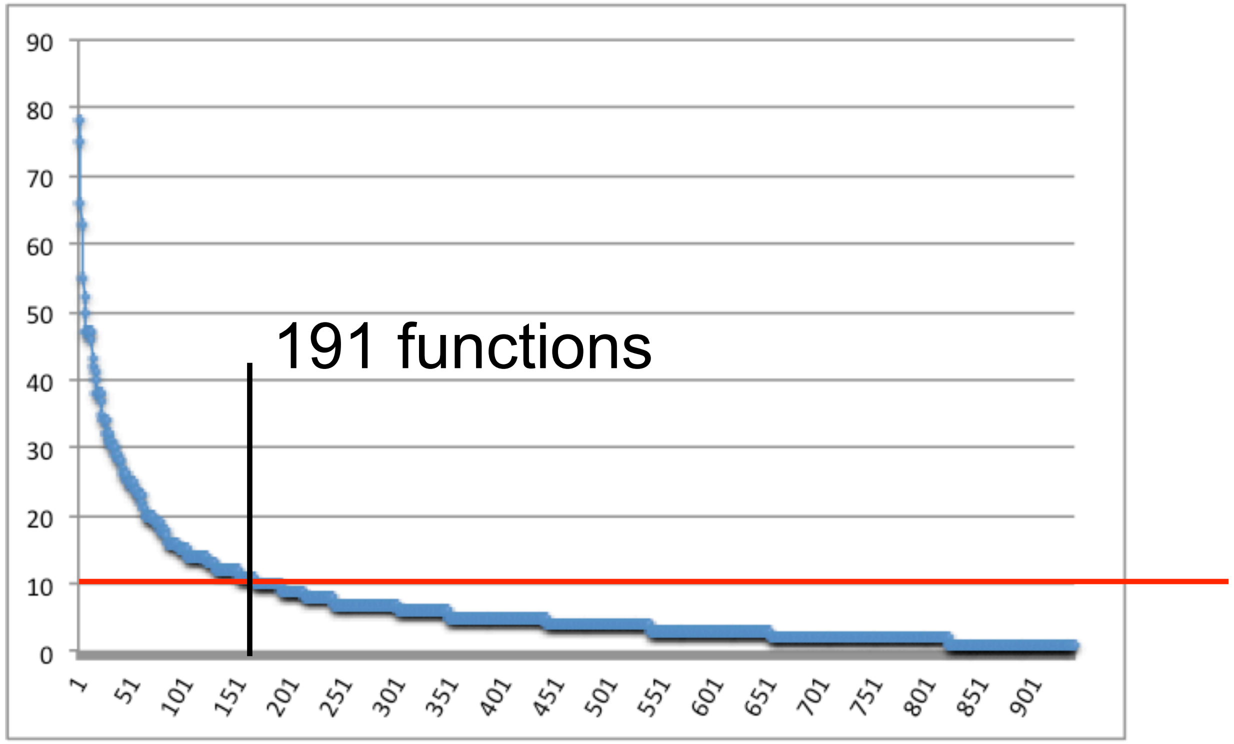
351 functions

25 lines (max recommended)

Extremely powerful tools available at CERN for C/C++ (e.g. Coverity)

80000+ violations of "coding rules"

- C functions by cyclomatic complexity (control flow)



191 functions

complexity 10 (max recommended)

Lack of tools to analyze Fortran

Not necessarily needed
as soon as design and abstraction are good

Poor Designs Symptoms

- *Rigidity*
  - ➡ The software is difficult to change, even in simple ways
- *Fragility*
  - ➡ The software breaks in many places every time it is changed
- *Immobility*
  - ➡ The software is hard to extend and requires hacks to evolve
- *Redundancy*
  - ➡ The software fails to reuse/be reused by others, leading to duplications
- *Viscosity*
  - ➡ The development environment fails to build and test the software efficiently

Do these terms make sense to you?

MAD-X core suffers from the lack of design and refactoring
Syndrome of "it works, don't touch it!"

Coding Design Rules *(very simplified)*

- Keep Coupling Low *(Dependency of Knowledge)*
  - ➡ enhance readability and maintainability
  - ➡ **importance of interface**
  - ➡ very high in MAD-X core

- Keep Cohesion High *(Locality of Knowledge)*
  - ➡ enhance reliability, reusability and understandability
  - ➡ **importance of encapsulation**
  - ➡ very low in MAD-X core

Better in Fortran code

Show the importance of understanding
the programming languages

MAD-X like Projects & Parsers

- BMAD (stands for “Better MAD”) & TAO (Tool for Accelerator Optics)
  - ➡ much like MAD-X & PTC-FPP, very active, very complete (ahead of MAD-X?)
  - ➡ compatible with PTC (?)
- LEGO (library similar to PTC-FPP)
  - ➡ Symplectic integrators for integration of the Hamiltonian
  - ➡ Differential algebra for Taylor map up to arbitrary order
  - ➡ Optics analysis for the linear and non-linear cases
  - ➡ Monte Carlo simulation for synchrotron radiation
- GMAD parser (from BDSim, an extension of GEANT4)
  - ➡ clean & extensible parser, fully compatible (?)
- Universal Accelerator Parser/Accelerator-ML Project (ANTLR-based)
- COMPASS SciDAC-2
  - ➡ very large all-in-one project for multi-physics accelerator design
  - ➡ only on paper?
- MAPA (GUI), Merlin-Pt (library), Placet (linac), ...

I haven’t check the details...

Myths and Legends

- *Speed is the main concern, the code must work, etc...*
  - ➡ Do it **right**, then do it **simple**, then do it **fast**, is the only way to obtain good results
- *Project XXX failed because of C++*
  - ➡ failure of a project has to do with project management
  - ➡ failure of the physics has to do with the understanding of the physics
  - ➡ programming languages are **tools** with features and expressivity which should mainly help to abstract interfaces and implementations and save human resources globally
- *Fortran is fast, C++ is slow, C is error-prone*
  - ➡ C, C++ and Fortran have about the same speed and accuracy
    - often share the same optimizer and code generator (e.g. Intel & GCC compilers)
  - ➡ C is very flexible but requires more discipline to write clean & readable code
- *C++ is slow and not suitable for scientific computing*
  - ➡ most recent scientific libraries are written in C/C++, including in the HEP community
    - e.g. linear algebra, differential algebra, tensors algebra, symbolic manipulation
  - ➡ multi-paradigms and expressivity of C++ are world-wide appreciated for SC
  - ➡ C++ allows to code at higher level of abstraction without speed loss

Part II
The Future

Can We Do Something?

Motivation for Changes

- Reduce the size and complexity of the code
  - ➡ improve the robustness of the code (policy of zero pending bug)
  - ➡ make the physics readable and accessible (as much as possible)
  - ➡ attract motivated (young) physicists (“it works” syndrome)
  - ➡ restore the confidence of users (“iceberg” syndrome)
- Reduce drastically the resource consumed by maintenance and support (< 0.01 pmy)
  - ➡ presently debugging is non-deterministic (may be endless)
- Improve the extensibility for future
  - ➡ parser and layout limitations and complexity (the parser is not ... a parser)
  - ➡ new modules for new physics, not for new projects (e.g. PTC specific cases)
  - ➡ stem external MAD-like rewrite, propose collaboration on sound code
- Add new physics modules (consensus around “PTC does it right”)
  - ➡ PTC everywhere with optimized specializations for simpler cases
- Focus on the essential (better FPP)
  - ➡ FPP as a white box (**where I can help the most!**)

Motivation for Closing

- Stop the resource leakage consumed by maintenance and support
    - ➡ solve pending bugs (may take long time or introduce new bugs)
    - ➡ close the modules, freeze the code
    - ➡ use as-is, provide limited support

- Start a “new” project on top of PTC-FPP (aka MAD-P)
    - ➡ not starting from scratch!
    - ➡ dealing only with working physics (PTC approach)
        - with optimized specializations for specific cases
    - ➡ quickly extensible to other project (≠ adapt the existing)
        - PTC special cases
    - ➡ focus on the mathematics (FPP) and the physics (PTC)
        - bottom-top approach
    - ➡ save resources, simplify the management, motivating
    - ➡ open new possibilities

Orientation & Strategy

Improvement

Concept Evolution

~~R~~evolution Strategy

- ~~Rewrite from scratch~~
  - fun (matter!), simpler, risky (i.e. risk to never converge)
  - long period without improvement for the end user
  - hard to combine with support of previous releases with limited resources
- Improve the situation
  - modify smoothly the existing and ensure working steps (i.e. regression tests)
  - must be transparent to the end user (e.g. only fully validated substitution)
  - must stay backward compatible (e.g. scripts)
  - much less risky for the current project
    - the existing remains the main project
    - we can move backward at any moment
    - moves depend on the available resources
    - **danger of priority inversion** (i.e. endless support and no evolution)
  - much more complicated
    - requires a strategy (e.g. substitution principle)

Proposed Strategy *(works for changes and closing)*

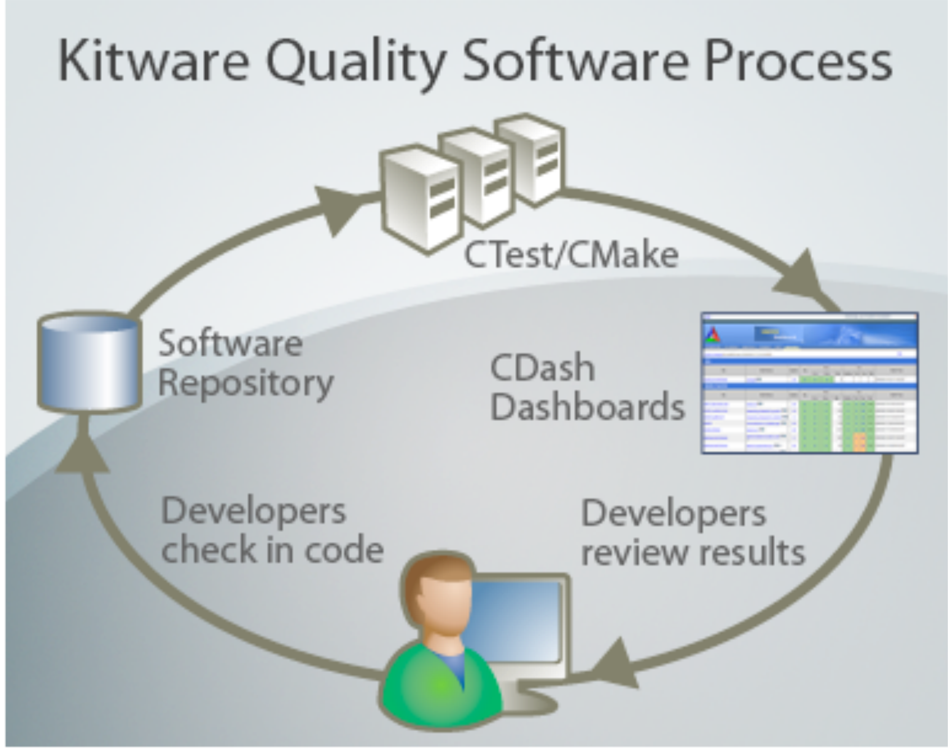
- Improve the **communication** through open mailing lists (e-groups)
  - ➡ better information sharing and dispatch, history of ideas and actions
- Improve the **build system**
  - ➡ protect developers and users against regressions
  - ➡ save resource for future maintenance
- Improve the **separation of concern**
  - ➡ close the modules (i.e. enforce encapsulation and interface)
- Provide an **I/O module**
  - ➡ ensure clean and robust coordination of C and Fortran I/O
  - ➡ allow better logger and pipe usages (i.e. direct mode users)
- Provide an **extensible parser and layout** (as a new module, see later example)
  - ➡ better separation between physical objects (layout) and math objects (properties, maps)
  - ➡ safe layout, invariants constraint, state machine for updates
- Make **FPP** comprehensible
  - ➡ better reuse of mathematical structures and algorithms
  - ➡ improve expressivity and knowledge of FPP

Proposed e-groups

- Public mailing lists (e-group, all archived)
  - ➡ hep-project-madx
    - very low bandwidth, only for important information (e.g. releases, reports)
  - ➡ hep-project-madx-usr
    - low bandwidth, only for users related topics (e.g. bugs, features)
  - ➡ hep-project-madx-dev
    - medium bandwidth, only for developers related topics (e.g. bugs, features)
  - ➡ hep-project-madx-src
    - medium bandwidth, source code modifications (automatic emails from svn commit)

Build System

- Build process
  - ➡ cross-platform (Linux, MacOS X, Windows), automatic and simple to use
  - ➡ The CMAKE suite (~standard @CERN)
    - build & regression tests & dashboard (nightly)
    - user tests (weekly)
    - official releases (when significant)
- Bug tracking system
  - ➡ could be replaced temporally by e-group
  - ➡ could be of interest in some future
    - bugs classification
    - bugs correlations
    - bugs assignations
    - bugs history
    - widely used at CERN by large team



Parser & Layout Substitution Example *(changes only)*

1. Retrieve an existing parser compliant with MAD-X scripting language (or write one)
2. Transform it to a new “hidden” MAD-X module with activation option
3. Tests compliance with the existing parser and layout (e.g. parse twice) *(state compatibility)*
4. Provide common interfaces to all modules for old and new layout
5. Provide synchronization with the existing layout (e.g. update twice)
6. Update modules to common interface (one by one) *(interface compatibility)*
7. Check modules actions and invariants (one by one)
8. Make the old layout read-only and activate the new layout
9. Check for invariants and remaining variants
10. Disable the old layout and check for consistency *(functional compatibility)*
11. Test deeply on many many cases!
12. Make the new parser and new layout the default (long after last step)
13. Call for a drink

*Questions, proposals, help
are welcome*