Software Development Practices in Project X

Todd A. Oliver
toliver@mit.edu
Aerospace Computational Design Lab
Massachusetts Institute of Technology

April 8, 2005
Overview

- Project X (PX) research and software goals
- Introduction to extreme programming
- Continuous integration
- PX build tools
- PX testing tools
- Conclusions and future work
Project X Introduction

Team Goal:
- To improve the aerothermal design process for complex 3D configurations by significantly reducing the time from geometry to solution at engineering-required accuracy using high-order adaptive methods

Students
- Garrett Barter (shock capturing)
- Tan Bui (unsteady aero/structures)
- Shannon Cheng (plasma physics)
- Krzysztof Fidkowski (hp adaptation)
- James Lu (optimization and adaptation)
- Todd Oliver (turbulence)
- Mike Park (meshing/adaptation)
- Peter Whitney (aeroacoustics)

Advisors
- David Darmofal
- Robert Haimes
- Jaime Peraire
- Karen Wilcox
Goals for Software Development Practices

■ Efficient code development
  ► Accomplish research goals as fast as possible

■ Flexible and lightweight — Little up front design required
  ► Difficult to generate specific software design and long-term plan in research setting

■ Readable code
  ► Readable code serves as its own documentation
  ► Easier to maintain

■ Test as much code as possible as often as possible
  ► Minimize debugging time

■ Integrate as often as possible
  ► Avoid code integration nightmares
Extreme Programming

■ Extreme programming (XP) developed by Beck, Cunningham, and Jeffries in mid-1990s

■ Agile software development methodology based on four values:
  ► Communication, simplicity, feedback, courage

■ Consists of twelve core practices:
  ► Sustainable pace, metaphor, coding standards
  ► Collective ownership, continuous integration, small releases
  ► Test-driven development, refactoring, simple design
  ► Pair programming, on-site customer, planning game
Wood and Kleb applied XP to development of advection-diffusion solver in Ruby


FAAST program at NASA Langley has incorporated XP methods into development approach

- http://fun3d.larc.nasa.gov/
XP in PX

- Metaphor: High-order DGFEM, CFD jargon — $p$, $q$, $\rho$, $\rho u$, etc.
- Coding standard: Virtually universal header comment conventions and some standard notations, but still lacking
- Collective ownership: No restrictions on who can modify what, but low truck number
- Test-driven development: Unit testing framework recently became available, but use not widespread yet
- Refactoring: Performed, but typically only to improve speed or when blatantly necessary
- Pair programming: Used infrequently
- **Continuous Integration**: All executables are built and entire test suite run after every CVS commit
Concurrent Versions System (CVS)

- Integration begins with software versioning system (in our case CVS)
- All source files stored on CVS repository
- All differences between versions of file also stored ⇒ can always revert to old version if necessary
- Developers checkout the code from the repository and commit changes
- CVS merges changes into code

**Bottom line:** CVS allows multiple developers to work on same code with very little chance of overwriting each other’s changes or making conflicting changes.
Continuous Integration Overview

Developer commits change

Wait 5 min without new commits

Pass

Fail
(reset clock)

Is the build already running?

Yes

Run the PX build and test procedure

No

Wait until current build and test finishes

Log fail on website and send email

Fail

Notify when finished

Pass

Log success on website

Wait until current build and test finishes
Build and Test

What happens inside the build and test?
- Executables built
- Unit tests run
- Acceptance tests run

Tools required:
- Build utilities (Autoconf and Automake)
- Unit testing framework (CuTest and PXUnit)
- Acceptance testing framework (runTests)
Autoconf

“Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems.”

– GNU Autoconf Manual

- Developer supplies `configure.ac` file
- `configure.ac` contains sequence of calls to Autoconf macros, for example,
  - `AC_PROG_CC` determines a C compiler to use
  - `AC_PATH_X` locates X header files and libraries
- Running `autoconf` produces `configure`
- Once created `configure` does not depend on Autoconf
Configure

- `configure` determines features of build environment, including
  - System type (linux, cygwin, etc)
  - Selects compilers (gcc, g77, mpicc, etc)
  - Probes for necessary system libraries (X11, mpich, etc)
  - Probes for necessary system headers (stdlib.h, string.h, etc)

- `configure` sets variables based on environment it finds
  - Allows creation of portable Makefiles

- Creates Makefile from Makefile.in

- Where does the Makefile.in come from?
  - You or ...
Automake

“Automake is a tool for automatically generating Makefile.ins from files called Makefile.am. Each Makefile.am is basically a series of make variable definitions, with rules being thrown in occasionally.”

– GNU Automake Manual

- Developer supplies Makefile.am files that are converted to Makefile.ins by running automake or make dist

- What is make dist?
  - Automake supplied target that creates tarball for distribution
  - Tarball contains configure and machine independent Makefile.ins

- Automake also supplies other “standard” targets
  - install, clean, check, ...
# -*- Makefile -*-

if HAVE_MPI
    bin_PROGRAMS = PXRunSolver2d PXRunParallel2d
else
    bin_PROGRAMS = PXRunSolver2d
endif

PXRunSolver2d_CFLAGS = -DDIM=2 -I$(top_srcdir)/include
PXRunSolver2d_SOURCES = PXRunSolver.c
PXRunSolver2d_LDADD = libPX2d.a libPX.a -lm @FLIBS@

if HAVE_MPI
    PXRunParallel2d_CFLAGS = -DDIM=2 -DPAR=1 -I$(top_srcdir)/include
    PXRunParallel2d_SOURCES = PXRunSolver.c
    PXRunParallel2d_LDADD = libPXPar2d.a libPX.a -lm @FLIBS@
endif
Unit Testing

- Unit tests exercise small pieces of code in isolation from each other and the application as a whole.
- Most easily applied to low level functions
  - Flux calculation, viscosity calculation
- Useful for medium level functions if low level functions adequately tested
  - Calculation of inviscid Galerkin residual
- Not applicable to highest level functions
  - Line solver
- To make unit testing practical, need a unit testing framework
- For list of unit testing frameworks see http://c2.com/cgi/wiki?TestingFramework
CuTest

- CuTest is a C unit testing framework written by Asim Jalis
- CuTest provides
  - Assert functions (e.g. CuAssertDblEquals, CuAssertStrEquals)
  - Functions for aggregating tests into suites and running test suites
  - Functions for recording and reporting test failures
  - Simplicity—only 2 files: CuTest.c and CuTest.h
- For PX purposes, CuTest drawbacks include
  - Tests are not added to suites automatically
  - CuTest defines structures that are required in testing code
PXUnit

- Set of C macros and 1 shell script
- Eliminates CuTest drawbacks for PX developers
  - Automates process of adding tests to suites and producing a main program
  - Eliminates need for PX developers to interact with CuTest data structures
- No knowledge of CuTest required to write unit tests in PX
PX_TEST( TestStaticTemperatureTrivial ){
    int ierr;
    PX_REAL params[6] = {1.4, 1.0, 1.0, 1.0, 1.0, 1.0};
    PX_REAL T;
#if( SA_TURB == 1 )
    PX_REAL U[5], T_U[5];
#else
    PX_REAL U[4], T_U[4];
#endif
    U[0] = 1.0; U[1] = 0.0; U[2] = 0.0; U[3] = 2.5;
#if( SA_TURB == 1 )
    U[4] = 1.0;
#endif
    #endif
    ierr = PXError( PXStaticTemperature(U, params, &T, T_U) );
    PXAssertIntEquals( PX_NO_ERROR, ierr );
    PXAssertDblEquals( 1.0, T );
}

Acceptance Testing

- Regression tests: Ensure code produces same answer as yesterday
- Verification tests: Ensure code produces expected order of accuracy
- Validation tests: Ensure code results match experimental data or analytic exact solution

- Only automated acceptance tests currently in PX are regression tests
- Regression tests controlled by two shell scripts
  - jobTest.csh: Runs single test and reports pass/fail
  - runTests.csh: Runs all tests and reports total number or errors
- Change in any output quantity (e.g. residual, force, adjoint residual, etc) of greater than 1e-13 causes failure
Build and Test

- `cvs co projectx` → Pass
- `configure` → Pass
- `make check (build and unit tests)` → Pass
- `make install` → Pass
- `runTests (acceptance tests)` → Pass

**Report failure**

- Fail
- Fail
- Fail
- Fail

**Report success**
Conclusions

- Shown that agile software development philosophy applicable in scientific computing environment
- Developed continuous integration procedure for use in Project X
- Developed build and test procedure to check code after every modification
- Features of the build and test procedure include:
  - Build of all executables and libraries in Project X
  - 214 unit tests
  - 17 acceptance (regression) tests
Possible Improvements

- Enforce more rigorous coding standard
- Expand unit test coverage and use of test-driven development
- Extend build and test to run automatically on multiple architectures
Acknowledgments

- Prof. David Darmofal
- Garrett Barter
- Chris Fidkowski
- Mike Park
- PX Team