My Favorite Practices and Tips

Emmanuel CHENU
emmanuel.chenu@fr.thalesgroup.com
http://emmanuelchenu.blogspot.com

This is a synthesis of my favorite practices and tips, read in various books.

CONTINUOUS_INTEGRATION

Early and at all times, the team ensures that the project is fully integrated, compilable, runnable, tested, ready to deliver and deploy.
All these instructions are automated by a one-action build script with two modes: full and incremental.

TEST_DRIVEN_DEVELOPMENT

"Tests are to prevent bugs, not find them."
The programmers work in very short cycles, adding a failing test, then making it work. When they get a bug report, they start by writing a unit tests that exposes the bug.
Writing tests first is a design tool. It will lead you to a more pragmatic, simpler and less coupled design.
Make sure all tests are fully automated, that they check their own results and are run on each supported platform and environment combination by a continuous integration tool.

CUSTOMER_TESTS

As part of selecting each desired features, the customers define automated acceptance tests to show that the feature is working.
These tests are run as part of the continuous integration build. They do the most with the least.

SIMPLICITY_AND_CLARITY

Correctness, simplicity and clarity comes first.
The team keeps the design exactly suited for the current functionality of the system. It passes all the tests, contains no duplication, expresses everything the authors want expressed, and contains as little code as possible.
When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

CODE_REVIEWS

Code reviews are invaluable in improving the quality of the code and keeping the error rate low. Review code after each task, using different developers.
Pairing is continuous reviews: All production software is built by two programmers, sitting side by side, at the same machine.
Moreover, code reviews are a great way to enhance collective code ownership.

COLLECTIVE_CODE_OWNERSHIP

Rotate developers across different modules and tasks in different areas of the system. Any programmer can improve any code at any time.

REFACTORING

Fix bad designs, wrong decisions, and poor code when you see them.
When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.

STANDARDS

All the code in the system looks as if it was written by a single - very competent - individual.

Embody the current best known practices in standards that are always followed while actively encouraging evenryone to challenge and change the standards.

TEAMS

All the contributors to a project - developers, business analysts, testers, etc. - work together in a single open space, members of one team. The walls of this space are littered with big visible charts and other evidences of their progress.
Stand-up meetings keep the team on the same page. Keep the meeting short, focused, and intense.
Innovate from the bottom up.
Real insight comes from active coding. Don't use architects who don't code - they can't design without knowing the realities of your system.
Publish your status , your ideas and the neat things you're looking at. Don't wait for others to ask you the status of your work.

TOOLS

Costly tools don't produce better designs. Keep critical path technologies familiar.

VERSION_CONTROL

Always use version control system. If you need an element, check it in.
Never check in code that's not ready for others. Never keep files checked out for long periods.
Deliberately checking in code that doesn't compile or pass its unit tests should be considered an act of criminal project negligence.

BREAK_DEPENDENCIES

Design components that are self-contained, independant, and have a single, well defined purpose.
Tell, don't ask: Don't take on another object's or component's job. Tell it what to do, and stick to your own job.
Minimize global and shared data. Sharing causes contention: Avoid shared data, especially global data. shared data increases coupling, which reduces maintainability and often performance.
Prefer composition to inheritance: Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know thet the latter benefits your design.

INCREMENTS

Write code in short edit/build/test cycles.

ROOT_CAUSE

Don't fall for the quick hack: "Quick fixes become quicksand."
Keep asking why: Keep questioning until you understand the root of the issue.

MISTAKE_PROOF_CODE

Design with contracts: Use contracts to document and verify that the code does no more and no less than it claims to do.
Use assertions to validate your assumptions and contracts.
The broken assertions enable to crash early and point to the root cause.
Compile cleanly at high level warning levels and take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

TRACK_ISSUES

Maintain a log of problems and their solutions: Part of fixing a problem is retaining details of the solution so you can find and apply it later.

DESIGN_STYLE

Extend systems by substituting code: Add and enhance features by substituting classes that honor the interface contract. Delegation is almost always preferable to inheritance.
Public inheritance is substitutability.

Consider making virtual functions nonpublic, and public functions nonvirtual. Prefer writing nonmember nonfriend functions.

DRY - Don't Repeat Yourself: Every piece of knowledge must have a single, unamiguous, authoritative representation within a system.

Program close to the problem domain. Design and code in your user's language.
Use a project glossary. Create and mainain a single source for all the specific terms and vocabulary for a project.

BIBLIOGRAPHY

[PAD] **Practices of an Agile Developer**, Venkat Subramaniam and Andy Hunt
[XP] **eXtreme Programming**
[PP] **The Pragmatic Programer**, Andy Hunt and Dave Thomas
[ILSD] **Implementing Lean Software Development**, Mary and Tom Poppendieck.
[SI] **Ship It!**, Jared Richardson and William Gwaltney Jr.
[CCS] **C++ Coding Standards**, Herb Sutter and Andrei Alexandrescu.
[R] **Refactoring**, Martin Fowler.