

Six Million Dollar Tester: Making QA Better/Stronger/Faster Through Automation

Presenting a framework for identifying and
evaluating tools and automation
opportunities.

Who is this guy, anyway?

- Leander Hasty <leander@1stplayable.com>
Programmer, 1st Playable Productions, LLC
- A brief history...
 - Taldren
 - Black 9 – Win32 / Xbox / PS2
 - Shadowbane – *NIX / MacOS / Win32
 - GoPets
 - Electronic Arts
 - LotR: BfME – Win32
 - QA / Central Technology
 - 1st Playable
 - CPK – GBA
 - ...various unannounced titles...

Why bother with automation?

- Facilitation of QA
 - Supplement / extend / replace tester work
 - Allow qualitative vs. quantitative feedback (“make better games”)
- Early Detection
 - Fix bugs pre-QA involvement
 - (Save embarrassment!)
- Iteration Time
 - Allow more iterations
 - Reduce bug turnaround time
- Metrics!
 - Help to “close the loop” on design, art, production, and engineering decisions.
- Look more professional!
- Wow your friends!
 - Be the envy of the industry!

The Framework

- Why a framework?
 - “Mapping the territory” - finding areas of coverage
 - “Prioritizing” - identifying easy opportunities (“low hanging fruit”)
 - “Sharing” - providing a common language for future discussions
- What's in our framework?
 - Prevention:
 - Stop problems before anyone notices. =)
 - “Don't even allow problems to be committed, if possible.”
 - Detection:
 - Force problems out of hiding.
 - Detect issues as early as possible.
 - Monitor the health of the project.
 - Analysis:
 - Gather as much data as possible.
 - Mine data in intelligent ways.
 - Use results to alter processes and practices.
 - These stages tie fairly easily into the bug and project lifecycle.

Prevention, Part 1

- “Prevention” tools live early in the feature cycle.
- Mostly lightweight auditing and sanity-checking tools.
- Examples:
 - Enforced coding standards!
 - (easy to moderate, implementation time varies)
 - “design by contract”
 - Code cleanliness / health tools:
 - (easy to moderate, low implementation time)
 - Parasoft CodeWizard
 - Gimpel PC-Lint
 - Unit testing suites / test harnesses:
 - (easy to hard, implementation time varies)
 - JUnit / NUnit / etc
 - In-house tools / add-ons:
 - (easy to moderate, implementation time varies)
 - Basic integrity checks (holes in the level, etc)
 - Art integrity checks (polycount, collision volume problems, etc)

Prevention, Part 2

- Integrating and automating preventative tools:
 - a) Deploy as a part of the “personal buildprocess”
 - Maya / Max / etc have “export” or “save” scripts
 - Most IDEs have pre-build and post-build scripts
 - Custom level editing tools can do anything!
 - b) Use source control “triggers” (see Perforce or SVN) to control submissions
 - There will be no failing material committed.
 - Can be troublesome to maintain.
 - c) Fire tools based on automated notifications (see Perforce “Reviews”)
 - This may be a fallback item.
 - Avoids burden on the client or server machines, though.
 - d) Poll committed material extremely regularly
 - This strategy may be useful for Detection and Analysis tools too; in that case it avoids some duplication of work.
 - Also avoids burden on the client or server machines.
- Give immediate feedback!

Detection, Part 1

- “Detection” tools live in parallel with testing.
- Heavyweight tools generally fall in this category.
- Some may be manually run...
- Examples:
 - Build validation tools
 - “smoke” tests (shallow and broad)
 - (usually easy; short implementation, med-short runtime)
 - Level load tests
 - “slice” tests (narrow and deep)
 - (difficulty varies; implementation varies; med-long runtime for a suite)
 - GUI traversal
 - Basic functionality verification (network connections, etc)
 - stress tests / soak tests
 - (easy; usually low implementation, long runtime)
 - Talkback tools
 - automated crash reporting
 - (usually easy, depending on platform)
 - Automatic escalation of frequent issues?

Detection, Part 2

- Examples, continued:
 - Random play:
 - Random pad input (a.k.a. “padmonkey”, at least some places I've been)
 - (usually really easy; often a manual tool, sometimes used with soak tests)
 - AI vs. AI games
 - (can be very easy; often used during specialized soak tests)
 - Special environments:
 - Boundschecker
 - KDE's Valgrind
 - Parasoft Insure++
 - ...even a normal debug build
 - All of these are useful in conjunction with most of the others here.
- A slight digression: built-in testing tools...
 - Cheat menu (Don't leave home without it.)
 - Quick load / jump to
 - Switch languages
 - Command-line switches

Detection, Part 3

- Implementing, integrating, and automating Detection tools:
 - Scripting capabilities in the engine can be of great help...
 - ...use them to drive AI, set up test scenerios, and the like.
 - Use a test harness or build environment (“hire a manager”):
 - Find a product with enough flexibility to handle all your test needs.
 - Use your buildsystem to run your tests.
 - Activate tests in the buildsystem with a build step.
 - (Make the tests a build step, if you like.)
 - NAnt, Jam, even Make are amenable to this sort of thing...
 - Use a “tiered” approach:
 - Per-build or hourly suite (rotating)?
 - Nightly suite (also rotating)?
 - Milestone verification suite?
 - Special “drill” suites for things like networking, AI, etc?
 - Get a dedicated test machine (or even a test farm!)
 - Don't burden your client machines (frustrating)
 - Don't overextend your build servers (suicidal)

Analysis, Part 1

- “Analysis” tools live late in the cycle.
- Generally more useful when they've been in use for a long time
- Examples:
 - Gameplay statistics
 - Data should be automatically gathered and databased
 - Save log messages, to start.
 - Extending what data is gathered should be easy!
 - Think in terms of “events” at “locations”, or adopt another paradigm.
 - Reports should be developed on an as-needed basis
 - e.g. plotting kills on an FPS level map to determine active vs. inactive areas
 - Replay (i.e. the flight recorder)
 - When possible, games should permit recording and deterministic playback!
 - If not easy, consider investing \$\$\$ into Identify AppSight BlackBox or similar
 - Save replays from all gameplay sessions and test runs!
 - As good as video for debugging...
 - Replays can be run through other test suites...

Analysis, Part 2

- Examples, continued:
 - Information garnered from previous tests:
 - PC-Lint error difference reports / trends
 - Memory trends over time (if you can harvest this from logs)
 - Metrics on draw calls, polycounts, CPU spikes, etc...

Analysis, Part 3

- Implementing, integrating, and automating “Analysis” tools:
 - Use a lightweight client/server if possible (think syslogd)
 - Send all log messages
 - Send special events notices with extra data
 - Send stackdumps and error data as appropriate
 - Send replay data!
 - Automatically track session start/end, machine name, player, etc.
 - It is possible to do this on consoles, too. Be creative.
 - Gather and batch data / logs to the server after execution
 - If impacting performance or behavior is a concern, do it before or after execution!
 - Tried-and-true for crash reporting.
 - Gather data through multiplayer services!
 - If you're using GameSpy or similar, you may already have a data conduit.

Analysis, Part 4

- Techniques and practices for data mining
 - Make frequent use of “diff” and “grep”
 - Graphs and other visualizations can often help; they're not as difficult as some might think...
 - Group data. Identify trends:
 - Over all games
 - For game periods (e.g. first five minutes, second hour, etc)
 - For certain levels / maps / areas
 - For individual players
 - For teams of developers
 - For individual developers
 - ...and so on.
 - Provide data in the format the consumer wants.
 - They'll be able to see things you won't, in most cases.
 - Make new queries/reports easy.
 - A well-structured DB backend can help here.
 - Automatically deliver the data regularly.

Overlap

- Significant overlap
 - Between prevention and detection
 - Code health tools can be run as either of these
 - Between detection and analysis
 - You can frequently mine data during normal gameplay or detection runs (e.g. logs, events, replays, etc); take advantage of this!
 - Crash reporting is a good example here...
 - Between analysis and prevention
 - The results of the analysis tools should feed back into the prevention and detection tools to make them better.
 - Do a periodic audit of all the data you've mined to identify new tool opportunities.
- It's a big circle!

Identifying Opportunities

- There are many ways to identify potential new tools...
 - Chronic problems
 - Physics, geometry, etc... Who hasn't had these problems?
 - Research
 - Read postmortems (Write them too! Praise your tools!)
 - Read software development journals.
 - Attend local IGDA meetings. Ask people!
 - Subscribe to on-line communities and mailinglists.
 - Listen & Watch
 - The “scream of despair” happens often.
 - Be a tester, an artist, a producer, or a designer for a day. Allow yourself to be frustrated. =)
 - Bonus: Think about future opportunities for the tools you develop.
 - (Mine data in the release build. Business people love stats.)

Implementing It All

- Start with a sky-high whiteboard. Put everything on it.
- Organize by ROI - “Return on Investment”
 - Identify targets of opportunity based on the tools and tech you already have.
 - Identify potential risks in your project and prioritize tools that mitigate that risk.
 - Address the “low hanging fruit” first. It gets buy-in from executives and consumers.
 - Make a sales pitch based on all the data you come up with, especially if commercial tools will help.
 - Keep pestering people.

Best Practices

- Use some sort of buildsystem / test harness.
- Allow interoperability between tools (the *NIX philosophy).
 - (Buildsystems and test harnesses really help here.)
- Use existing tools – don't reinvent the wheel!
 - VisualBoyAdvance
- Don't be afraid to write your own tools if necessary.
- Save every bit of data you can get your hands on.
 - Disk space is cheap.
 - It can be mined later.
- Show only the data that's needed.
 - Create smart diffs / graphs and target them; people don't need to be overwhelmed by the amount of data.
 - (Web interfaces really help here.)
- Each issue resolved should spawn a new tool or a new rule.

Important Things

- The organizational structure of your company: how do you interact with QA?
- Availability of tools and automation engineering resources: how many people, how much time?
- Engine architecture and support:
 - The availability of a scripting language. (Get one if you can.)
 - The ease of integrating unit tests. (Is it an old codebase? Form a plan of attack.)
- Non-standard clientelle:
 - Are you building an engine or a framework, or a library?
 - Are you serving someone who is doing this?

Summary

- ounceOfPrevention = poundOfCure
- Prevention, Detection, and Analysis tools should all work with one another.
 - Each should bolster the others.

“Your Turn”

- Successes and Failures?
- Common “low-hanging fruit”?
- Common heuristics / best practices?

Q&A