Technical Debt Catalog

"Friendly additions" to the interfaces, (developer to developer)

"Release it now and finish it afterward" leads to unfinished/untested or not fully tested code

"To do" comments

"Whistler" screen development (average time to develop a screen is ridiculous)

50% of code is in one class (for a 500,000 LOC system)

A lot of "to do"s in our code

Absent documentation (new work has to be done without understanding, which would be supported by documentation)

Accelerated development schedules

Accumulation of debt that is never paid off, and there's a compounding effect of that

Adding third party vendor code or acquired company code into our code base

Attempted "reuse" of things that don't really work for what they're being reused for

Build system

Build time (executable takes 2+ hours to build on a really powerful machine)

Build vs. use third party library instead of improving own engine

Business needs to have a working solution rather than a perfect solution (cause of debt)

Buying bigger hardware instead of revising architecture (eventually have to make change, but with more users)

Caveats for releases (i.e., publish list of caveats but don't ever fix the underlying issues)

Changing requirements that force you to hack instead of do a good design

Code branching, per customer (with hundreds or thousands of customers)

Code duplication

Code is written focusing more write-time convenience than read-time convenience

Code where developers didn't follow coding standards

Code where developers didn't follow design

Comments, lack thereof

Complexity of test

Compromises to modularity for latency concerns

Continuing to build on a foundation of (less than perfect) legacy code (and not properly wrapping)

Copy and paste ("clone and hack")

Deadlines

Declining code due to lack of standards enforcement

Defect fixes postponed to next release

Deferred testing

Delayed adoption of upgrades (so no straightforward upgrade path)

Delayed implementation of logging functionality (reduced ability to figure out production issues if this functionality isn't present)

Delayed performance concerns

Delayed refactoring (copy and modify rather than refactoring)

Deliverable dependencies so a lot of tactical fixes / workarounds

Demo code becoming production

Deprioritizing paying off technical debt

Design in general is short-term focused; not enough long-term thinking

Design that wasn't validated or mature before moving into implementation

Documentation, lack thereof

Duplicated code

Duplicated effort -- two groups each implementing solution to the same problem, both of which need to be maintained

Duplication of code / functionality

Effort needed to merge branches later

Ever-changing requirements

Evolving coding standards

Expedient data model changes--shoehorn in something rather than remodel your data (e.g., edge cases)

Failure to refactor to combat software entropy

Features that are partially released

Free BSD

Future-proofed code

General "good enough" (sloppy) kept too long -- eventually fixing defects takes longer than it would have taken to write better originally

Huge, complex classes that you plan to fix later but that you never fix

Ignoring performance just to get features done then optimize later on

Implementation before requirements have been finalized

Implementation in advance of new technology that isn't ready yet / code around hardware

In house policy language too awkward to use (and it gets used without improving it)

Inactive code

Incomplete automated test coverage (run tests manually, which takes time, and that leaves less time to write automated tests)

Incomplete log files don't give enough debugging information

Increasing use of "bad patches," which increases number of other systems that must be changed in parallel

Increasing use of a hastily designed class interface

Intermittent results (test may pass and fail on same build)

Introducing multiple code paths for similar things

Lack of a good debugger -- still mostly using print statements

Lack of a working dev (dev/integration/test) environment

Lack of common framework to enable extensibility in the future

Lack of communication / communication mistakes

Lack of debugging tools

Lack of design

Lack of documentation or out of date documentation (e.g., design documents)

Lack of full system testing (including implications for bugs in the field)

Lack of HAL (each new hardware means more work)

Lack of testing on new code

Lack of understanding of legacy code

Lack of unit testing, only integration testing

Lack of unit tests

Lack of unit tests ("we'll create them later")

Lacking testers' mentality

Legacy code

Legacy code maintenance

Legacy systems (not migrating clients off of the legacy systems and then having to maintain them)

Limiting release by releasing only partial functionality, including infrastructure support

Low priority tickets (defects) not fixed

Maintaining code for multiple products across multiple code branches

Maintaining software for legacy hardware

Make assumptions in requirements; then later have more work to do when assumptions are explored better

Make system / build system

Making assumptions about requirements so that you can begin implementation (and some assumptions turn out to be wrong)

Making changes for customers on a purely short-term, temporary, one-off basis (not tracking insertion of deficiencies)

Missing or incomplete features

Missing unit tests (someone intended to write but never got around to it)

Need to do more design and code reviews (attack the unintentional tech debt)

Need to support multiple games (is kind of a repeat of what happened before when needed to support multiple platforms)

Network failures

No time to maintain documentation

No unit tests

Non modular code makes future code changes/additions harder

Non-standard implementation of DLNA

Not considering all audiences for product (including configuration management)

Not considering enough design options to find a good one

Not doing a code review on your intern's work

Not doing code reviews

Not enough abstraction (too much vendor specific stuff spread throughout, at different levels)

Not fixing bugs because new version of client is coming up soon

Not fixing something when we think of it because not enough time, and then later it takes so much more time that we don't do it

Not iterating on the design enough (as underlying cause of tech debt)

Not programming for "N" (i.e., programming for specific number of cases instead of variable number of cases)

Not pursuing perfect extensibility in terms of design (or any extensibility, for that matter)

Not sufficiently optimizing and code not scaling

Not supporting future standards

Not testing adequately (corner cases) before release

Old versions of any third party software

Outdated documentation or no documentation

Overloaded interfaces

Ownership of code quickly changes (transition from "new code" team to next team)

Parallel system and not retiring the old system

Performance hit due to inactive code

Personnel changes

Poor cohesion on legacy code (including multiple quick fixes that were introduced because we were still using that legacy code)

Poor comments

Poor network and power outages (e.g., one power outage lost a 20 TB DB and 3 weeks of work)

Poor requirements

Poorly understood, poorly documented legacy code -- hard to know where to make changes

Porting code from different platforms and hacking it to make it work

Power outages

Products that aren't end of life (after we thought they were going to be)

Profile generation (lack of convention and standardization)

Promoting code we know is going out and not going back to fix it (code rot)

Prototype gets turned into production

Prototype that turns into production code

Prototyping that evolves into the solution (this is a classic mistake!)

Quick fixes, where the specific issue is "fixed" but not all dependencies are fixed (leaves some other bugs unfixed)

Reliance on immature technology

Remove functionality in the short term to meet deadlines with plan to add it back in later

Requirements changes (continuously)

Revision control (Clearcase -- but opinions vary)

SCDP Package (it's become a dumping ground)

Scoping problem (tradeoff about scope of technical solution is not made well)

Seven years of legacy code before adding a unit testing structure

Severe lack of information hiding (every header file includes every other header file)

Shipping something that is not production quality (as beta) and planning to fix it later

Short term vision (designing and coding for what we need right now)

Shortcuts that are not ever removed or improved

Shortcuts to meet at deadline

Skill level of developers

Skipping improvements because it would cause a lot of regression tests

SOAK test support

Start testing the product after you release it ("calculated risk" of releasing with unit testing but not integration testing)

State machines are brittle / rigid

Test data does not have all cases needed in production

Testing code after it's been developed rather than during development

Third party code, including open source libraries

Tight coupling with legacy components (that make refactoring difficult)

Tightly coupled code

Too many temporary solutions (that didn't get converted to better solutions later)

Trying to solve every single problem (including problems that really should be supported by the client)

Tunnel vision (focus on stop gap solutions, which you have to untangle at some point down the road, for integration of acquisition purposes)

Undocumented or poorly documented code

Unimplemented todos

Unintentional debt by overcomplicating the design (e.g., designing for use cases that will never occur)

Unsupported third party interfaces

Untested forward compatibility

Unused code

Use of third party code with unstable APIs

Using a low quality third party library

Using prototype code for production

Using VB coding approach in C#

Version of TCL

Work done by contractors (or other people outside the group) who leave or become unavailable to the group

Writing code during launch

Writing design after coding (=not designing before coding)

Wrong technology choice (in hindsight, sometimes, or due to lack of personnel)