TSDuck

Anatomy of a single-person open-source project

Topics

- Genesis of an open-source project
- Resource constraints
- Coding
- Maintenance
- Tests
- Documentation
- Delivery
- Support
- Infrastructure

Genesis of an open-source project

- It all started from a personal need
  - advanced research project on transport stream security
  - need to increase knowledge in TS structure
  - in-depth TS analysis
  - real-time transformation of TS using Dektec ASI devices
  - => needed flexible manipulations of TS for experimentations
- Then some colleagues used it for different purposes
  - unexpected usages
  - proved the usefulness of the toolbox
- And finally could be useful to any DTV engineer
  - we all receive a lot from open-source tools
  - sometimes, it's time to give back in return
  - => open-source your work

TSDuck timeline

- 2005-2006 : V1
  - written in C (a mistake!)
  - Linux only
- 2007-2011 : V2
  - scrapped and re-written in C++
  - multi-platform architecture, including Windows native support
- 2012-2015 : hibernation…
  - no longer needed to work on transport streams
- 2016-2021 : V3
  - moved to open-source
  - renamed as TSDuck
  - many improvements and new features
  - started a community of users

Resource constraints

- Personal project
  - less linked to my professional activities over time
  - on spare time only
  - on personal expenses (hardware, web hosting)
- Limited resources
  - no fully equipped lab
  - reduced time availability, no continuity
- Resource-driven project
  - scarce resources is the main driver for the project organization

Productization

- An open-source product is still a product
  - but many wonderful open-source tools
    - have zero doc (ffmpeg, openssl)
    - are a pain to build (dependency issues, poor Windows integration, exotic build tools)
- Essential qualities of a product
  - reliability (no bug, no crash…)
  - stability (no memory leak…)
  - documentation
  - packaging and installation
  - support (assistance, bug fix)
  - communication (web)
- All of this with limited time and resource…
  - self-discipline and automation are essential

Coding principles

- Efficiency-driven coding
  - write code rapidly
  - invest time in coding, don’t lose time in debugging
  - anticipate instead of debug
    - full compile-time code checking
    - use all language features to enforce defensive coding techniques
- Integrated « quality by design »
  - explained in a *TSDuck coding guidelines* document
    - a generic programming manifesto, not limited to TSDuck
  - based on past professional experience
    - large projects in Ada, Java and C++
    - stringent software engineering rules and methodologies

Coding techniques

- Use proven object design patterns
- Robustness enforcement
  - resist to incorrect or malformed input data
  - defensive coding, cross-checking, assertions, bug self-detection, etc.
- Avoiding resource leaks is easy
  - don’t spend time on new/delete or lock/unlock
  - implement « safe pointer » and « guard » classes
  - the C++ concept of « destructor » is invaluable !

    you can’t even count on it in Java or Python (not to mention C of course)

    properly using it saves hours of debug
- Refactoring
  - never let the quality of the code degrade, refactor properly
  - too many projects accumulate quick & dirty fixes or copy/paste

    and finally collapse over time because of an inconsistent code base

Coding cadence

- Extreme Agility
  - coding on spare time only
    - no time to enter long coding tunnels
  - small iterations
    - consistent and clean, commit on master branch
    - successfully compile and pass tests
  - avoid divergent branches
    - merge & rebase takes time, I haven't any
- Make short term a long term investment
  - plan evolutions on the long term
  - code step by step on the short term
    - dormant code for future features
    - => if you don't understand the latest commits, they will make sense later...

Maintenance

- Maintain stability
  - automated full non-regression tests after each commit
  - each build is as stable as a release
- Releases
  - the concept of « official release » is purely editorial

    same automated QA as any build

    => you may safely use nighlty builds
  - build of a “release” is fully automated from a macOS host

    boot, build and shutdown Linux and Windows virtual machines

    remote build on Raspberry PI
  - releases are tagged in git, published on GitHub

Tests

- Time resource constraints
  - no time to debug or come back on earlier developments

    => avoid regressions
  - test-driven development
- Low-level unitary tests
  - JUnit-like dedicated framework (« TSUnit »)
  - when developing low-level feature, use it as test & debug environment
  - 560+ tests, 28000+ assertions
- High-level test suite
  - commands and plugins scenarios in a dedicated git repository
  - 90+ test suites, 1300+ tests

Tests automation

- Automation
  - using « GitHub Actions » continuous integration
  - all tests are run on all push and pull requests
  - on Linux, Windows and macOS
- Limitations
  - limited to fully automatable tests
  - no QA team => no manual tests

    especially on hardware features (tuners, Dektec and HiDes devices)
  - best testing effort within the resource limits of the project

Documentation

- Continuous documentation
  - code-test-document in each iteration
  - require self-discipline
- User's guide
  - Microsoft Word
  - PDF and details are automatically updated using a PowerShell script
  - ~500 pages
- Programmer's guide
  - doxygen from code
  - automatically generated and published every night on tsduck.io
  - ~3000 HTML files

Binary deliveries

- Proper binary packaging is essential for user experience
- System-specific delivery
  - Windows: executable installer (NSIS)
  - Linux: rpm and deb packages
    User's contribution: AUR on Arch
  - macOS: Homebrew
    User's contribution: MacPorts
- User-friendly build
  - one-liner (« make ») or one-click (« build-installer.ps1 »)
- Full automation
  - scripts
  - continuous integration using GitHub Actions
  - automated production and publication of nightly builds on tsduck.io

Support

- Building a community means providing support
- GitHub issue tracker
  - mostly used as a discussion forum
- Spend time for users
  - … but not too much time
  - explain, explain, explain
    - give investigation clues or suggestions, don't give solutions
    - the default answer always remains « RTFM ! »
- Fix bugs quickly
  - bugs are annoying for everyone, they ruin the reputation of a project
- Implement suggested new features
  - transform user's requests in generic new features for everyone
  - protect the architecture and principles of the project

Infrastructure

Home (development)

NAS

git

backups

macOS

native: git

VM:

Ubuntu git

Debian git

Fedora git

CentOS git

Alpine git

Arch git

Windows git

Gentoo git

Raspberry Pi

native: git

push

GitHub

git

releases

issues

on push

on pull request

GitHub Actions (CI/CD)

continuous integration

Ubuntu

macOS

Windows

nightly builds

Ubuntu

Windows

Doc

uploaded every night

tsduck.io (web hosting)

PHP / HTML

nightly bins

streams repo

docs

Limited investment

- Limited personal development environment
  - basically an iMac and a Synology NAS on the shelf above it
- Full usage of GitHub features
  - git repositories
    - tsduck, tsduck-test, dektec-dkms, hides-drives, srt-win-installers, homebrew-tsduck
  - releases management and publishing
  - issue tracker
    - used as a discussion forum in practice
  - GitHub Actions CI/CD environment
- Web hosting @OVH
  - basic presentation of the project
  - transport streams repository

Thank you

Any question?