Sane software manifesto

v0.10+5 (826fe3e0bae6+1)

In respect to user freedoms, privacy, liberty and software quality we create software according to the following guidelines. Developing Sane software is not easy, however we believe that this is the right way because this software is written once but used many times and maintained for years or decades.

Free software

- Every piece of Sane software is also Free software (as originally defined by Richard Stallman). Which means that the user has freedom to run the program for any purpose, to study and change it (i.e. has access to the source code under a free software license) and to distribute modified or unmodified copies.
- The user controls his computer and software and owns the data. Not the author of the software or anyone else without user's explicit consent.
- Must be buildable using free software toolchain (like GNU/Linux + GCC or OpenJDK etc.).
- Must not promote non-free (proprietary) software or services.
- Copyleft licenses (like GNU GPL or GNU Affero GPL) are strongly recommended because they guarantee software freedoms to every single end-user and prevent possibility that freedom vanishes somewhere in the distribution chain and the user can not benefit from the free software albeit the software is build on originally free source code.
- The license must be compatible with GNU GPLv3 in order to allow mixing with the GPL code. The only exception is older software (created before this manifesto i.e. 2019) which is unable to change the license due to the copyright owned by many authors who can not be reached anymore and who can not provide approval with the license upgrade. Such software is called „Sane with exception“.
- If the software is distributed with a hardware, the hardware must support installation of independently built software without any restrictions or requirements (e.g. digital signature from the original author).

Documented

- At least basic documentation must be released under a free license (GNU FDL is recommended).
- Every advertised feature must be properly documented. Undocumented features can not be considered as features from the user/customer point-of-view.
- There might be also other documentation/books released under any license and price.

- But average software engineer must be able to build and operate the software with just the free (basic) documentation.
- There must be a free documentation with description of building and running the software on a fresh operating system installation including description of all dependencies.

Semantic versioning and upgrades

- Semantic versioning is required. The version number consists of three numbers: major.minor.patch. Major version is incremented if there is an incompatible change. Minor version is incremented if a feature is added in a compatible way. Patch version is incremented if a bug is fixed in a compatible way.
- Once publicly released, the package must not be changed anymore – if a change (even a small fix) is needed, new version number must be assigned.
- APIs, file formats and protocols might (and usually should) be semantically versioned independently from the implementation.
- The branching model in the version control system should reflect the semantic versioning. The released version e.g. 2.3.1 should be tagged as v2.3.1 and be placed in the v_2.3 branch. Where the v_2.3 branch was forked from the v_2 branch – from the v2.3 tag.
- Do not remove features unless they are really obsolete, unused or irreparably broken.
- The user interface might be simplified or redesigned while preserving the features under the hood.
- Incompatible changes must be planned and announced in advance.
- Upgrade scripts and upgrade documentation must be provided.

Names and identifiers

- The name of the product and its significant parts should be reasonably unique, so it can be found using full-text search or other standard methods.
  - Avoid generic words like common verbs, nouns or adjectives.
  - Avoid name collisions with well known and used software.
  - Use company or organization name as part of the name if the name itself would be too generic (e.g. „Speed“ is wrong while „SaneCorp Speed“ is right).
- For globally unique identifiers, the URI format is recommended.
  - Besides the uniqueness, the most important feature of it is immutability and stability.
  - Identifiers can be derived from an internet domain, an OID or PEN number etc.
  - When deriving from internet domains use the tag URI scheme to create timeless identifiers that are not affected by the changes in domain ownership.
  - Use randomly generated identifiers when full decentralization is desired. These identifiers might be just random (e.g. UUID version 4) or derived from a public key (e.g. an SSH key or a Tor address) or a hash of some (secret) data.

Interfaces, formats and protocols

- Open standards (protocols, formats) should be used if they exist.
- Already existing open protocol/format must not be modified or extended in a way which effectively creates a proprietary protocol/format.
- New open standards (specifications) should be defined and published if needed. Such standards must be semantically versioned.
- And they should be described using a machine-readable format (e.g. WSDL, ASN.1, XML Schema, Diameter dictionary, D-Bus) or at least formal language (Backus–Naur Form, EBNF etc.)
- Also configuration should have machine-readable description and the user should be able to test it by executing a command (validator).

Modular architecture and extensibility

- Larger and multi-purpose software must be divided into smaller modules.
- The modules must have defined dependencies (less = better).
- Particular modules should be compilable and executable (installable) independently. It should not be necessary to recompile the core and other modules if only one module is changed.
- Another good ways to extend and customize the software are: configuration (XML, RDF/Turtle, INI, RegExp, SQL, XSLT, XPath etc.) and scripting (Scheme, Bash, Python, Lua, ECMA Script etc.)

Testable

- Tests verify the compliance of the implementation with the documentation or specification.
- There should be automated build-time complex tests for the package. The test feeds the program with sample input and verifies expected output.
- There should be also automated runtime/postinstall tests – in order to verify that software was installed properly, all required dependencies are met and basic function is guaranteed – the program should report problem during its start (as a warning if it is not fatal), instead of unexpected failures during operation.
- Unit tests are recommended for code parts that are internally complex (algorithms, important business logic) and have simple interfaces.
- Each external interface should contain procedure/function that does nothing important or heavy, is idempotent and returns simple response which proves that the interface (connection) is working (e.g. echo, print version, status or current time). If authentication and authorization mechanisms are present, there should be one procedure/function callable anonymously and one that requires authorization.

Safe code and sustainability

- Correctness, safety and readability is preferred to performance.
- Strong data typing should be used, preconditions and possible exceptions should be declared in the interface and they are part of the contract.
- Data structures must be known and well documented. Do not use e.g. undocumented map keys or properties.
- Code, comments and specification should be written in the same natural language.
- There should be a dictionary of used terms, so whole team and also users and customers will speak the same language.
- Fail fast – errors in the code should be reported during build time or at least on first execution – do not silently continue if given error would lead to failure later in another part of the code – bad weak coupling leads to difficult debugging.

Small code footprint

- Less LOC (or cyclomatic complexity) = better.
- Boilerplate and unused code should be reduced. Adequately high-level programming language or framework should be used.
- Code generators should be used (during the build process, not to generate code to be manually edited and versioned).

Sane dependencies

- Avoid NIH syndrome and reuse code but also avoid dependency hell. Complexity is defined by the complexity of the program itself + complexity of its dependencies.
- Know your dependencies, know why they are required.
- Depend on small and useful libraries – not on bulky application packages or libraries with large transitive dependencies.
- If dependency on bulky application package is inevitable, add a layer of abstraction – create a generic interface and connector and allow others to replace the bulky package with their own sane implementation.
- Complexity caused by helper tools should be also reduced:
  - If you e.g. use Bash and Perl during the build process, do not add also Python dependency, write it in Perl – or use Python instead of Perl.
  - Or if you use Java as your main language, consider not using Python/Perl for scripting and use some JVM language for it.
- If possible, always depend on abstract interfaces, not on particular implementations.
- From the whole system point-of-view, Bootstrappable builds should be taken into account.

Easily auditable

- Small code footprint and sane dependencies make it easier to do security audit.

- Ungrounded refactoring and reformatting should be avoided – they make mess and noise in the version control system and impede the audit.
- Refactoring/reformatting changesets should be separated from substantive changes.

Reproducible builds

- Builds should be reproducible i.e. same code/version should lead to the same binary package.
- If they are not reproducible, it should be documented, why and how build products might differ, and there should be plan/task to make builds reproducible.
- However, there should be also documented way, how to build the software with different (similar) version of the toolchain (not reproducible) that is available on given machine.

Trustworthy packages and sources

- Every released version (binary or source) must be cryptographically signed by the authors (GnuPG/OpenPGP is strongly recommended).
- There should be also checksums/hashes for every released package.
- If HTTP is supported, HTTPS should also be – the attacker/eavesdropper should not even know what software/package/update is downloaded by the user.
- The attacker must not be able to suppress updates – the program (usually a package management system) must not be silent in such case and must warn the user that something possibly nasty and dangerous is happening.
- Releases should be downloadable also (or exclusively) over BitTorrent or other P2P network.
- Source code repository must be accessible through a secure and encrypted connection.

Network interactions

- Network connectivity must not be required during build – the build must be possible completely offline. All dependencies must be downloadable and documented including secure hashes or preferably cryptographic signatures.
- If dependencies are optionally automatically downloaded during or before build, the packaging system must cryptographically verify that they are undamaged.
- Avoid unwanted network interactions during runtime. There must be no „call home“ or update-checks without user's explicit consent.
- If any network connection is used, it must be by default cryptographically secured against MITM attacks.

Internationalization and localization

- Any software with nontrivial user interface must be internationalized which means that it allows localization (translation of the UI to national languages and other customization to national conventions).

- It should be possible to localize the user interface independently from the original author by creating an additionally installable language pack.
- GNU Gettext or other standard framework (like Java resource bundles) should be used.
- Error messages should have assigned unique error codes, so it is possible to find relevant information regardless the current locale.
- Data formats and protocols must be language/locale independent.
  - e.g. use decimal point instead of comma and no thousand separators for numbers, use standardized date formats
  - in general: everything that is expected to be machine-readable or machine-generated must be independent from the current locale
- Character encoding:
  - The software should always be aware of it and do not just blindly use current platform's default.
  - If given software/format/protocol has some default encoding, it must be clearly defined in its specification and this default must not be changed without changing the major version number.
  - If there is no default, the encoding must be specified in the metadata attached (e.g. protocol headers, extended attributes on filesystem) to the actual data or at least at the beginning of the data (like declaration in the XML format).
- The metric system should be used as default.

Communication with users and developers

- Following information should be provided in RSS/Atom or other machine-readable format: announcements (security, new versions, infrastructure outage), blog posts, tutorials and AFK events (e.g. conferences, meetings or hackatons).
- A mailing list (e-mail conference) or other equivalently open and decentralized technology should be used for the many-to-many communication.
- Users must not be pushed to register at proprietary social networks (at particular providers of such services). Users without such account must not be disadvantaged – use open and decentralized networks/protocols instead.
- There should be a second-level internet domain for the project or its team.
- URLs should be as stable as possible – accessible in next decade.
- The website must be independent and must contain everything needed – any content (JavaScripts, CSS, fonts, images etc.) downloaded from other domains must not be required to browse/use the website.
- JavaScript or other code executed on client computers must be also free software with properly declared license.
- The website should not require a modern complex browser for basic tasks like reading the documentation, downloading a release or submitting a bug report. Such tasks should be feasible even with simple text browsers (e.g. Lynx or Links2).
- There must be a crpyptographically secured (GnuPG/OpenPGP or X.509) e-mail address or a secure web form for receiving security vulnerabilities reports.

- Source code repository (versioning system) must be public. Do not publish just source code snapshots of released versions.

Accepting contributions

- Good quality code contributions with appropriate copyright and patent licenses or assignments should be accepted from anyone.
- The „good quality code“ is defined by the project and might involve code style, idioms, design patterns, software architecture, required tests, documentation etc.
- Such requirements and rules should be available to the contributor before he begins. However (especially smaller) projects might communicate such code quality requirements and provide consultations and guidance during the contribution.
- In order to contribute, it must not be required:
  - to have an account on any particular third party service like particular e-mail or hosting provider,
  - to sign a contract (which includes accepting „Terms and conditions“) with any particular third party (e.g. source code hosting provider),
  - to sign any political, religious or other proclamation or agree with it.
- In order to contribute, it might be required:
  - to have an e-mail address (but not at particular domain),
  - or use similar decentralized technology which has open standard and free software implementations,
  - to assign the copyright to the project and grant a free license for all patents relevant to the contribution.
- The project should record all accepted contributions and maintain a public list of all authors/contributors.
- The contributor must not lose the right to use or distribute the contributed code under any license (of his choice).