MousePaw Media Standards

Release 1.2.1

MousePaw Media

Jun 17, 2022

CONTENTS

CHAPTER

ONE

ABOUT THE STANDARDS

MousePaw Media has authored a number of standards to guide their ongoing software development efforts. These standards are made freely available for the benefit of the open source community, and the programming community at large.

You can access the source files for these standards from the GitHub mirror. For more information about contributing, see *Feedback*.

CHAPTER

TWO

PUBLIC STANDARDS

These standards are intended for general use.

2.1 Commenting Showing Intent [CSI]

Version: 1.2.1

Last Updated: 2019-08-26

The CSI (Commenting Showing Intent) Commenting Standards refers to a style of code commenting which allows for the complete rewriting of a program in any language, given only the comments. It is also a backhanded reference to Crime Scene Investigation: putting the clues together to recreate the whole event.

2.1.1 Purpose and Theory

In many languages, it is considered "bad practice" to not comment code, yet so many programmers fail to do it. Many more write comments to themselves, but these comments are often cryptic at best. Still others write comments that restate the functionality of the code.

Because of these problems, several practices exist which condemn comments in most, or all, situations. This leads to code which becomes inexorably separated from its specification.

The CSI Commenting Standard offers a non-language-specific standard for writing comments.

Note: Intent-Commenting is discussed in detail in an article by Lead Developer Jason C. McDonald entitled To Comment Or Not To Comment

---

Advantages

We're asking you to type twice, if not three times, as much as you do now. What's the advantage to that?

1. CSI comments become a **living specification**, wherein the expected behavior of the program and the actual functionality live in close proximity to one another, and can be more easily kept in sync and up-to-date.

2. When you come back to your code after a long time away, you will be able to find your footing much faster.

3. When other developers (or non-programmers) read your code or API documentation, they will be able to understand it more thoroughly, much quicker. This is vital to efficient code reviews and third-party debugging.

4. You significantly reduce the entry learning curve for new contributors. Because learning the code is much easier, individuals are able to start making meaningful contributions more quickly.

5. You and your code reviewers will be able to desk-check, debug, and track complicated logical thought patterns much quicker. Mismatches between intent and actual behavior become evident.

6. When you are dealing with a complex logical process that you are trying to write code for, you can start by writing your entire function in CSI-compliant comments. Then, using these comments as a sort of pseudocode, you can write the code under each comment.

7. You can translate your program into another computer language much quicker. Again, by using the CSI-compliant comments, you can rewrite each line of code to work in your programming language.

8. The code becomes collaterally useful for demonstrating the language syntax and features themselves. Armed with an understanding of the goal, the less experienced developer can more quickly surmise how the code works. (See #4.)

CSI vs. Self-Commenting Code

"Self-Commenting Code" is a practice wherein a code's functionality is self-evident. Through naming, structure, and various other techniques, the immediate purpose becomes obvious to the reader. This is beneficial in any language.

However, "Self-Commenting Code" is seldom capable of expressing the entire intent, or "why", of the code.

- It is nearly impossible to express the *intended* behavior of the code;

only the *actual* behavior is evident, which can conceal logic errors.

- Code cannot imply the reason the current approach was taken over another.
- Code can seldom self-express its purpose in its larger context. Even attempting

to do so can lead to impractically long function and class names.

The CSI Standard should exist *alongside* Self-Commenting Code practices, not *instead of*.

	CSI	Self-Commenting
Topic	Intended behavior.	Actual behavior.
Question	**WHY** did we write this code?	**WHAT** does the code do?
Expresses	Language-agnostic specification.	Language-specific functionality.

CSI vs. Documentation

The CSI Commenting Standard **should not** be confused with documentation.

	CSI	Documentation
Audience	Maintainers and Developers	End-Users and End-Developers
Topic	Intent and design of code.	How to use the software/API.
Question	**WHY** did we write this code?	**WHAT** does the code do and **HOW** do we use it?

However, these standards *may* be merged with API documentation standards, to help produce better autodocs. The important distinction is that **CSI comments state WHY**, and **doc comments state WHAT and HOW**.

Keeping Up-To-Date

A common argument against comments is that *"comments become outdated too quickly"*, and *"maintaining comments takes extra work"*. However, proper application of the CSI Commenting Standard avoids both of these problems.

1. When developers are in the habit of using this standard, the first step in modifying code is to update the intent-comment.
2. Developers spend considerably less time trying to recreate the intent of the previous developer(s), including their past selves. Only a portion of this time must be used to update the comments.

2.1.2 Format

C-Based Languages

For readability, CSI comments should use single-line comments only for single-line statements. If more than one line is required, multi-line comments should be used (if available in the language.)

It is recommended that each line in a multi-line comment start with an aligned asterisk, as this improves the readability of the comment.

```
/* This is a multi-line
 * comment with the
 * preceding asterisk.
 */
```

In any language, we strongly recommend leaving an extra space between the comment token and the comment text, to aid in readability.

Python

CSI comments should not be confused with docstrings (see CSI vs. Documentation). Line comments should be used for CSI. Placing the comment above the code in question is recommended. Inline comments are prone to causing an overrun of PEP 8's line length limits.

```
# This is a CSI comment, describing intent.
doSomething()
```

Commenting Style

Again, many of these principles can be applied to documentation comments as well. The distinction is that CSI comments state **WHY**.

Note: I have intentionally oversimplified the code examples to make them easy to quickly understand. Most real code is far less obvious in its intention at first glance.

Tone

Comments should be written in a conversational tone, in the same manner that the code might be explained to a newcomer. It should be free of language-specific syntax as much as practical. This enables non-programmers (and programmers from other languages) to understand the code more easily.

BAD

```
// set box_width to equal the floor of items and 17
int items_per_box = floor(items/17)
```

This merely restates the code in a generic way, and it entirely redundant when paired with self-commented code. It also depends on the language term "floor" - if a reader is unfamiliar with this term, they will have to look it up just to understand the comment - a situation that we should avoid as much as possible.

BAD

```
// Find how many times 17 goes into y, without a remainder.
int items_per_box = floor(items/17);
```

Now we know what the code is doing, in a language-agnostic fashion. As a side benefit, the reader can also surmise what "floor" does, if he or she were unfamiliar with the term.

However, this comment is still not true CSI, as it is only stating *WHAT*, and not *WHY*. Furthermore, the self-commented code makes this redundant to an average C++ developer.

BEST

```
/* Divide our items among 17 boxes.
 * We'll deal with the leftovers later. */
int items_per_box = floor(items/17);
```

Now we know *WHY* the code is here - we're dividing our items among the boxes. We also know that this line isn't intended to handle the extra items (thus why we are using `floor()`).

If you imagine a lone maintainer looking to change this code to divide the items among any number of boxes, the comment would make his change obvious, even with a minimal understanding of the code…

```
/* Divide our items among the specified number of boxes.
 * We'll deal with the leftovers later. */
int items_per_box = floor(items/boxes);
```

Avoiding Vagueness

CSI comments should specifically outline the programmer's logic and reasoning. The more left unsaid and undefined, the less effective the comment.

BAD

```
// This tells us how much we can handle.
int maximum_range = 27;
```

This is too vague, and redundant given the variable name. (I'm assuming this isn't being clarified by immediately prior comments.)

BETTER

```
// This tells us the maximum workable integer
int maximum_range = 27;
```

This is still vague. If we didn't know exactly what "maximum workable integer" meant in this context, we'd still be confused. (Again, assuming no context.)

BEST

```
// Anything larger than this integer causes the algorithm to return 0.
int maximum_range = 27;
```

Ahh, so the *algorithm* has a specific limitation! All becomes clear…

Humor

Humor should not be suppressed, so long as it does not detract from clarity. It makes the documentation a lot easier to read, because who likes dry documentation?

The first rule of humor is applicable here, though: don't force it. If you try to be funny, you won't be. The only point is to not force yourself to be totally serious.

That said, don't be crass for crass' sake, as it may drive away others, detracting from the whole point of this standard.

ACCEPTABLE

```
/* We return -1 instead of 0 to avoid a
 * math error in the upcoming division. */
return -1;
```

BETTER

```
/* We return -1 instead of 0 to keep the
 * math gremlins happy in the upcoming division. */
return -1;
```

Context

Context is very useful in comments. Since we're aiming for a conversational tone, it is okay for one comment to help explain the comment immediately following. However, we do not want to become too reliant on context, as it is yet one more thing the reader must keep track of.

The following would be good in a short function.

EXAMPLE

```
/* count tracks the number of times the word "Bah"
 * appears in the given text. */

// We encountered a "Bah", increment the count.

// Return the count.
```

The following would be better in a very large function.

EXAMPLE

```
/* count tracks the number of times the word "Bah"
 * appears in the given text.
 */

// We encountered a "Bah", increment the count.

// Return the count of "Bah" instances.
```

Length

Obviously, the above practices will result in longer comments. This isn't a bad thing, as it seriously increases the code's readability, and speeds up debugging. Appropriate brevity comes with practice.

Bear this in mind: a single comment should state the **purpose** of a line or block of code in plain English.

ACCEPTABLE

```
/* Search through the list of integers we got from the user
 * and find the number of integers that are divisible by
 * both 5 and 7. Then, return the sum of those numbers. */
int sum = 0;
for(int i = 0; i < len; ++i)
{
    if( !(nums[i]%5) && !(nums[i]%7) )
    {
        sum += nums[i];
    }
}
return sum;
```

This attempts to pack entirely too much information into one comment, which slows us down. We now have to stop and determine what `sum += nums[i]` is doing, based on the big comment. It is also lengthier than it needs to be.

BEST

```
// Store the running sum.
int sum = 0;

// Search through the list of integers...
for(int i = 0; i < len; ++i)
{
    // If the number is divisible by both 5 and 7...
    if( !(nums[i]%5) && !(nums[i]%7) )
    {
            // Add it to our sum.
            sum += nums[i];
    }
}

// Return the final sum.
return sum;
```

By spreading out the comments, we can see the intention behind each piece of code. `sums += nums[i]` is obviously adding the number we found to our running sum.

Spreading out comments also helps to ensure they are kept up-to-date. One of the reasons programmers neglect to update comments is that they are not in the immediate vicinity of their other changes.

Frequency and Necessity

The core standard is this: **comment everything at first**. Each logical step should have an explanation. Yes, it doubles the size of your document, but you (and other people) will be able to better read the code and documentation later.

In a nutshell, aim to comment more lines of code, not to pack more into one comment.

There may be a rare occasion where a line of code is so entirely obvious and ordinary, a CSI comment would be redundant. However, before drawing this conclusion in a given instance, ask yourself whether someone entirely unfamiliar with the syntax and program would immediately know what the *intent* was.

OBVIOUS

```
# Greet the user.
print(welcome_message + username + ".")
```

This line of Python code is so obvious, we could choose to omit the comment and still be CSI-compliant.

MOSTLY-OBVIOUS

```
# Display the status or error code from the rendering engine.
print(get_status(render_engine))
```

This line is a little harder to parse, unless you know that our theoretical function `get_status()` queries the object's status, and returns it as a string. Even if we surmised that much, we might not know that error codes are returned here as well (perhaps we're looking for that line!)

NON-OBVIOUS

```
# Display the result of the final step of calculation.
print(str(foo%bar*baz))
```

We need the comment here to specify that we are actually completing the last step of a calculation within our print statement.

Trimming Contents

Commenting WHY instead of WHAT can be difficult, especially when you're familiar with the code. It may be tempting to write vague comments, or even remove them, as you work.

However, the purpose of the CSI standard is to inform the programmer who is *not* presently familiar with the code. Therefore, we recommend the following:

1. Comment every logical statement while working. No exceptions!
2. Have someone unfamiliar with the code review the comments and suggest improvements. You may be able to do this yourself, if you leave the comments AND code alone for a couple of weeks first.
3. Using the insight from Step 2, rewrite WHAT comments to WHY, and eliminate entirely unnecessary comments.

2.1.3 Types of Comments

Declarations

CSI-compliant source code should specify the purpose and intent of variables and functions. As previously mentioned this can be merged with documentation standards, especially because the resulting autodocs will be far more usable.

Note: If the name of a variable or function fully explains its intent, you may omit the comment as your documentation standard permits.

In these examples, we'll demonstrate combining CSI with a Doxygen-compatible doc comment. To that aim, the comments below contain the names of the items in question, in anticipation of the resultant autodocs.

VARIABLE/CONSTANT

```
/** The SILVER_INTEREST_RATE constant stores the
 * monthly interest rate for Silver savings accounts.
 */
const int SILVER_INTEREST_RATE = 1.06;
```

Preceding a variable or constant (especially the latter), we should state its intent - its purpose for existing. While a good variable or constant name tells us **what it is**, the comment should state **why it exists**.

FUNCTION

```
/** The countBah function determines how many times
 * "BAH" appears in a given string.
 * \param the string to count "bah" in.
 * \return the number of times "bah" appeared.
 */
int countBah(string inputText);
```

Immediately preceding a function declaration, its purpose should be stated, as well as the purpose of the input values, in plain English.

Special Comments

Using `TODO`, `NOTE`, and `FIXME` comments is common practice in many languages, and many tools exist that generate lists from these. The CSI standard recommends that these types of comments be used, and follow the same tone as other comments when possible.

```
// TODO: Expand the whatchamacallit to make whozits.
// NOTE: Is there a faster way to produce thingamabobs?
// FIXME: This math always seems to produce the result "2".
```

Entry Points

Major features should have entry points, which indicate where one should start reading the code if they want to follow the entire call stack for a particular function or feature. For example, if a game engine has a long process for generating an animated character on the screen, the beginning of this process - such as the function that initializes it - should have the comment…

```
// ENTRY: Generate Animated Character
```

From this comment, the reader can follow each class, object, and function through to the end to see the entire process.

In order for this to work, the call stack commenting should not have any "gaps" (such as a virtual function) that do not have some comment to indicate where the call stack continues in the code.

Entry points are not always practical, but where they are used, it will be much easier for a developer who is unfamiliar with the code to find "where to start".

Commenting Out Code

It can be very easy to confuse a regular comment and commented out code. There are two ways to clarify this action.

EXPLANATION METHOD

```
// It would seem that float is better for this task.
//int foo = 187;
float foo = 187;

// Just testing if we really need this function call at all.
//refreshEverything();
```

Here, we add a preceding comment to explain why the code was commented out. The benefit to this is that it helps you and other programmers recognize and follow changes in program logic.

This method is ideal in languages where double-commenting (below) is not possible.

DOUBLE COMMENT METHOD

```
////refreshEverything();
```

We can "double-comment" out the code. This is probably ideal in situations where the commenting-out is temporary, and you don't want to have to write an explanation.

COMBINATION METHOD

```
// Just testing if we really need this function call at all.
////refreshEverything();
```

By combining the two methods, you can see what code was commented out, while stating the reasons behind it.

This method is ideal in languages where double-commenting is possible.

In any case, you should ultimately aim to remove commented-out code as soon as possible.

Top of Document

On the top of the document, the programmer should ideally list the project name and version, module/class name and description, date last updated, and authors (optionally). This may be adjusted to comply with documentation needs and individual standards.

```
/* Dohickey Class [Some Epic Project]
 * Version: 1.0
 *
 * This performs jazz on input data to produce whatzit.
 *
 * Last Updated: November 25, 2014
 * Author: Bob D. Example
 */
```

Immediately following in a separate multi-line comment, include copyright and licensing terms. Because many licenses are extremely long, placing the license comment separate from the main top-of-document comment allows for the license to be collapsed in most code-folding-capable IDEs.

```
/* LICENSE
 * Copyright (C) My Really Cool Software Company.
 * Licensing yada yada goes here.
 */
```

2.2 Live-In Testing [LIT]

Version: 1.0.0

Last Updated: 2016-05-31

2.2.1 Purpose

The phrase "unit testing" is far too overreaching, and is linked heavily to the theory of Test-Driven Development. It is also intended for a very specific purpose, but is often mis-applied to all built-in tests.

Because we need a testing standard that has no relation to TDD, we have created the Live-In Testing Standard [LIT]. While it is our sole testing standard at MousePaw Media, other projects may choose to use this standard alongside TDD, as it is an unrelated standard.

2.2.2 Philosophy and Limitations

The main principle behind the Live-In Testing Standard is that all tests are built into the code being tested, and are compiled alongside of it. Tests are executed while the program is running.

There are several distinct advantages to this:

- Tests can be shipped with the software. Tech support can instruct users to run tests, in order to generate specific and useful debugging information. This information can in turn be used to fix the problem over the phone, or provide the necessary details to file a bug report.
- It is easier to extract useful information from a test output than from a log file.
- Benchmarking and (most) testing can be performed on all supported systems without the need to install special tools or applications.

2.2.3 Vocabulary

dependency
: a test that must be run before the current test.

lifetime
: a single test lifetime involves a single setup, any number of runs, and a single teardown.

repetition
: starting a single test one time generally has multiple repetitions. One consistency test may, for example, input the same scenario and validate the output 100 times in a single run of the test.

run
: a single time a test is executed by the developer or user.

scenario
: a complete and exact set of variables and conditions for running a test. Tests should control for all variables.

setup
: the actions taken to prepare a test to be run. Called once per lifetime.

teardown
: the actions taken once at the end of a test's lifetime, generally before the test object is deleted.

2.2.4 Implementation

MousePaw Media' **Golilocks** library (a component of PawLIB) is designed specifically for LIT in C++. **GoldilocksShell** (upcoming in PawLIB 1.1) can be used to quickly create a run-time interface for Goldilocks.

For other languages, the most important thing to remember about implementing an LIT testing framework is that all tests are executed *on-demand* by the user *during run-time*.

2.2.5 General Guidelines

- If a single test supports multiple scenarios, each scenario should have a unique code. For example, an Edge Test that randomly generates six integers for the scenario might also use those six integers as its scenario code. `3.13.9.15.39.779`
  - A given test may use any systems for generating the code, so long as the code is a) unique and b) can be used to duplicate the exact scenario in the test.
  - The test must be capable of re-running a scenario by an input scenario code. (Necessary for long-distance testing.)
  - Scenario codes must be made up of visible and recognizable characters (ideally Latin-alphanumeric with punctuation), and should be no more than 35 characters in length. (Again, necessary for long-distance testing.)
  - Particularly large scenarios may need to be randomly generated beforehand, hard-coded, and stored in a lookup table in order to make the scenario code practical.
- If a test relies on the proper functionality of a feature that is not being directly tested and validated, the test for that feature would be considered a dependency. For example, one must test file I/O before testing XML parsing, and XML parsing must be tested before using a test which writes a random XML file as part of scenario generation.

2.2.6 Test Types

Behavior Tests [B]

A Behavior Test tests a single common behavior or feature. These are generally most useful for comparative testing. A single letter may follow the ID to indicate which subversion of the test is being used.

Consistency Tests [C]

A Consistency Test repeats one scenario a large number of times within known limits, and ensures the output is valid and consistent. This is designed to test features and catch Heisenbugs.

Guidelines

- Whether a scenario is random or crafted, the test should repeat that scenario exactly multiple times.
- The output of each repetition should be identical to the output of the last repetition.
- Find repetition limits first, and stay well within them.

Developer Tests [D]

These are custom tests designed by a specific developer that don't fit in another category.

Edge Test [E]

Edge tests generate random scenarios for testing a given feature.

Guidelines

- Each scenario should be unique and random, as far as is possible and practical.
- Crafted scenarios may be integrated into the test, as long as the test scenario is still random enough to potential end-user situations. (The danger of crafting a scenario is our implicit and innate tendency to avoid triggering problems in the code in our tests.)
- An edge test may either explicitly target the valid creation of non-error response, or error response, but not both in the same test.
- Edge tests should be written to target vulnerable situations, that is, situations that have a higher probability of failing in an unusual scenario.

Fatality Tests [F]

Larger tests intended to trigger total program or system crashes. Intended to find the hard limits of the software and its environment. One example of this is to run a Consistency Test until the system's resources are totally consumed. The results of a Fatality test are generally useful in a) establishing checkpoints and failsafes that prevent the software from crashing or taking out the system, and b) validating comparability with a particular system.

Guidelines

- These should NEVER be run automatically! (Must be run manually.)
- The interface should display an error before running.
- A Fatality Test should not be totally dependent on the program being tested. (Logfile writing is vital.)
- System Fatality Tests should be monitored closely, and designed to crash the system in a manner which allows the test data to be collected.
- Fatality Tests should always generate a scream-and-die situation, so that the cause of the crash can be validated.

Integration Tests [I]

Smaller tests that are intended to ensure that connected classes are communicating with one another properly, and that constructors and initialization is functioning properly.

Guidelines

- Works primarily through ping/pong scenarios, which ensure that two classes, objects, or programming structures are able to access each other appropriately.
- Integration tests with OOP situations would generally need to use specialized dynamic allocation of objects, thus allowing the scenario to be fully controlled by the test.
- Integration tests should use the same constructors and initializers that are used in normal program execution. Thus, the necessary code for Integration tests would have to be hard-coded into the regular program structure, though most of the special code would only be triggered by the test itself.

Proposed Tests [P]

Any test that is written, but not yet adopted.

Stress Tests [S]

Larger tests intended to break fragile code, to make sure they're stable.

Guidelines

- Should find the breaking point of the targeted code or feature in all conceivable directions. Each direction may be split into a different test, depending on testing and project needs.
- Stress tests should never target a program or system crash. (See Fatality Tests).
- Should run as either an extreme Consistency test or Edge test, but not both in one test.
- Must write to an external log file, otherwise the information cannot be examined.

2.2.7 Test Suite Types

Regression Suite

A regression suite runs the minimum tests necessary to ensure that all basic program functionality works as expected. A single suite may target only the particular set of code or feature set affected by a change. (For example, we might have a separate Layers Regression Suite.)

Guidelines

- Made up of Consistency, Edge, and Integration tests only.
- A regression suite should be run on every Differential.
- Should only involve tests which can run quickly and automatically.

Use Suite

These are intended to simulate specific use cases. This allows us to ensure that simultaneous use of features isn't going to create problems.

Guidelines

- Made up of Consistency, Edge, and Integration tests only.
- Should ideally generate semi-crafted scenarios.
- May take longer to run than a Regression Suite, but should only involve tests which can run automatically.

2.2.8 Test Library Structure

Naming

A test should have a unique identifier in addition to a name. The unique identifier should start with a lowercase 't', and then it should indicate the type and a number. In the case of Developer and Proposed, initials are also required at the end. With Proposed tests, the intended category should precede the number.

Examples:

tD001-JCM
A developer test by Jason C. McDonald (with the number 001).

tB051
Behavior test 51.

tE105
Edge test 105.

tS068
Stress test 68.

tC971
Consistency test 971.

tC679-pJCM
A consistency test designed by Jason C. McDonald, but not yet officially adopted.

Namespaces

Each section of a project should be given its own "namespace" within test names, to prevent conflicts.

Major sections might be assigned an ID, which can be tacked on the beginning of the test name. For example, in the RATS Game Engine, the following IDs are used.

ID	Project/Subproject
A	Anari Graphics System
AP	Anari: Punchline
P	PawLIB
R	Ratscript
S	Stormsound
T	Trailcrest
TCE	Trailcrest: Content Engine
TUE	Trailcrest: User Engine
TWE	Trailcrest: World Engine
X	SIMPLEXpress

For example, `P-tB102` would be a behavior test for PawLIB.

Note: Depending on implementation, all tests for a particular project could be loaded into the test system on-demand.

It may frequently be necessary to further subdivide a project's tests. The first one or two digits of the test ID can be used to indicate the sector of the project. For example, within PawLIB, we use the following numbers:

ID	Sector
0x	Data Types
01	Trilean
1x	Data Structures
10	FlexArray
11	FlexMap
12	FlexQueue
13	FlexStack
14	SimplyLinkedList
15	FlexBit
16	Pool
20	IOChannel
30	PawSort
4x	OneString (Sector)
41	OneChar
42	QuickString
43	OneString
5x	Blueshell
6x	Utilities

Thus, looking again at `P-tB102`, that would be a behavior test relating to data structures. (We actually reserve the second digit for further subtyping - `10` relates specifically to FlexArray.)

Adoption

In many cases, tests should start as Proposed (`...-p???`). Then they are added later by the lead developer to the official library. This is to prevent conflicts when two developers add tests with the same name.

For example, `tC679-pJCM` would be a proposed test by Jason C. McDonald.

This step may be skipped if a single developer is working alone on a section.

Permanence

Most tests, with the possible exception of Fatality and some Stress tests, should remain in the code. This way, they can be run from a developer terminal by the end-user, as a component of long-distance technical support.

For example, a tech support agent could ask the user to bring up a developer Ratscript terminal (which would probably involve entering a unique key), and then type test `tC971`. The results could then be read back to the tech support agent (i.e. `FAILED: Could not create data structure.`, at which point the exact cause of the problem on the user's computer can be pinpointed.

2.3 Quantified Task Management [QTM]

Version: 2.0.0

Last Updated: 2021-07-07

2.3.1 Vision

For decades, programmers and managers have struggled to create a reliable system of tracking tasks and bugs. Managers need quantified (countable) ways to determine how much work has been done, and how long it will take until a task or project is completed. Programmers need to be able to manage their time, determine the importance of bugs and tasks, and predict release dates.

As depicted in Dreaming in Code by Scott Rosenberg, among many other books, quantifying programming is extremely difficult. Some companies have tried tracking lines of code written, but how should optimization (which actually *removes* lines in the process of making the code more efficient) be tracked? Others try to watch how many bugs were fixed or how many features are completed. However, not all bugs and features are equal in size, importance, or priority.

Three elusive main goals are needed for efficient project management:

- Quantification: It needs to be based in numbers.
- Granularity/Standardization: Everything must be measurable by the same system in order to make accurate comparisons. (We can't accurately compare equal sized crates, one filled with feathers, and the other with bowling balls.)
- Specificness: Over-generalization makes for less accuracy. The system should be able to track important task/bug metrics separately, not mush them all into one cover-all term.

What's New in This Version

- Clarify Priority.
- Added Distance.
- Improve Friction, drop `f0`.
- Improve Relativity, drop `r0`.
- Add Energy Points.

2.3.2 Measures

QTM consists of six measures:

- Priority: How soon?
- Gravity: How important?
- Distance: How much effort?
- Friction: How many available resources?
- Relativity: How much uncertainty?
- Volatility: How long did the bug go undiscovered?

Distance, Friction, and Relativity combine into Energy Points, which can be used as the points measure for Agile methodologies.

Priority

Priority refers to **how soon** this task needs to be accomplished. A task can be a low priority temporarily, and yet still have a high Gravity.

- **p5: Emergency.** Reserved for escalating a task above all other priorities; "drop everything and do this!"
- **p4: Now.** The usual top priority, and whatever is currently being worked on.
- **p3: Next.** Tasks that should be completed in the current sprint, after current `p4` and `p5` tasks.
- **p2: Later.** Tasks that might not be completed in the current sprint.
- **p1: Eventual.** Tasks which are not slated for the current sprint.
- **p0: Wishlist.** Tasks that don't necessarily need to be completed.
- **pT: Triage.** Tasks which are not yet prioritized.

If you're using a more traditional Kanban board, you may be able to skip implementing Priority.

Gravity (Importance)

A task's **Gravity** is its **importance to project goals**, including stability and ease-of-use. It plays an important role in planning. This is the only measure over which a client should have *direct* control.

This measure is especially useful when selecting features for removal to expedite project completion.

- **g5: Critical.** Project can't exist without. Must be completed, period. Should *never* include aesthetic and convenience functionality.
- **g4: Significant.** Must be completed, only cut if desperate. Includes only functional requirements which directly improve on g5 features. (No bells-and-whistles.)
- **g3: Major.** Non-essential, but should be completed if time permits. "Polishing" tasks, and the most important bells-and-whistles.
- **g2: Minor.** Not slated for current release, but may be g4 or g5 for next release.
- **g1: Trivial.** "Would be nice" tasks; these take backseat to the completion of all other tasks. Should contain only tasks that could reasonably belong in the project at *some point*; this is the pool for selecting g3 tasks for future releases.
- **g0: Wishlist.** All other tasks which have been properly discussed, but no higher Gravity rating could be determined.
- **g: Sum of all subtasks.** Use this for umbrella tasks that don't have a Gravity of their own. The Gravity will be the sum of the Gravity scores of its subtasks.
- **gT: Triage.** Proposed and unconfirmed tasks.

Note: Gravity is discussed in detail in an article by Lead Developer Jason C. McDonald entitled Three Ground Rules for Sane Project Planning

---

Distance (Effort)

Distance is a measure of **how much effort** it would take to complete a task, *given full domain knowledge*. The qualifier is important, as it allows developers to achieve consensus on Distance despite differences in their knowledge and skill level.

Distance measures effort in terms of time frame relative to the team's sprint length, although it should be understood that *effort* is being measured, rather than the timeframe itself. This is not a deadline.

- **d5: Exceeds Sprint.** Indicates task should almost certainly be broken down into subtasks.
- **d4: Within Sprint.**
- **d3: Within Half-Sprint.**
- **d2: Within Quarter-Sprint.** For sprints longer than two weeks, this can also be "Within Week".
- **d1: Within Session.** Work can be completed in one sitting.
- **dT: Triage Distance.**

There is no `d0` because all work requires some degree of effort.

Friction (Available Help)

Friction is a quantified measure of difficulty, based on how many resources are available to help complete a task, versus how much innovation (new invention and experiementation) will be needed. The overall health of the source code — good practice, patterns, clean coding — should also be taken into account.

Friction should always be objective and empirical; it should never involve the developer's actual experience level.

- **f5: Jungle.** Uncharted territory. You're on your own.
- **f4: Trail.** Little precendence and/or documentation. Mostly innovation, or work occurs in particularly unhealthy source code.
- **f3: Off-Road.** Some precendence and/or documentation. Work might occur in unhealthy source code, or significant innovation is required.
- **f2: Street.** Good precedence and/or documentation. Work likely occurs in healthy source code. Some innovation may still be required.
- **f1: Highway.** Low-skill tasks, tutorial-guided work, fixing typos. Work occurs in healthy source code.
- **dT: Triage Friction.**

There is no `f0`, as all work involves some friction.

Relativity (Uncertainty)

It can be easy to predict how much effort and time will go into a task, or it can be very hard. We can call this uncertainty "flux". Relativity is essentially a measure of how much flux is present in a task, and conversely, how reliable our time and effort predictions are.

A task becomes a black hole when you have absolutely no idea how much time or effort the task will require.

A good rule of thumb: you will know the relativity within the first hour of working on a task.

- **r5: Collapsing/Black Hole.** Total flux. Task needs to be abandoned or re-factored, as it is virtually impossible in its current state.
- **r4: High.** Significant flux. Completion possible, but unlikely.

- **r3: Moderate.** Moderate flux. Completion within sprint is uncertain.
- **r2: Low.** Some flux, but completion within sprint is likely.
- **r1: Trivial.** Very little flux. Probably safe.
- **rT: Triage.** Relativity not yet determined for task.

There is no `r0`, as one can virtually never say that a task is truly without flux.

Note: Relativity and flux (QTM v1) are discussed in detail in the article Gallifreyan Software Project Management

Energy Points

QTM can be used to inform the "points" used in Agile methodologies. When the points are determined this way, they are known as **Energy Points**, or just **Energy**.

The following formula is used to calculate Energy:

```
energy = (distance + friction) * relativity
```

A task's Energy score is a combination of direct effort (distance), research and experimentation effort (friction), and the task's uncertainty (relativity).

Over time, a developer will get used to how Energy maps to their individual time and effort on a team. A more senior member of the team will be able to complete more Energy Points worth of tasks in a work session than a junior team member, in general.

Higher Energy tasks will also likely require more focus than lower Energy tasks, even those of longer overall duration, due to Relativity. This fact informs a developer when selecting work from the backlog to complete. For example, if a developer is selecting work to do in the half hour before a meeting, they would probably find they could make more meaningful progress on an Energy 7 task than an Energy 25 task, even if both had the same Distance.

Finally, Energy is a good way to control how much work is selected for a Sprint, because it explicitly takes Relativity and Friction into account, rather than just overall effort.

Volatility

Cumulatively, Volatility measures how late in the development process bugs are being caught. This can be used to spot issues in software quality processes, and to provide an estimation of software stability.

Volatility has two parts, although only one is absolutely necessary. The first is the Volatility measure on the bug itself, indicating what development stage it was caught in.

- **vN: Not a bug.** Feature requests and other non-bug issues should *always* have this rating (or else `v0` if you can't implement `vN`.)
- **v0: Caught in Design phase.** This means the bug was anticipated before coding even began.
- **v1: Caught in Coding phase.** This means the bug was caught before it reached a protected branch, such as `devel` .
- **v2: Caught in SQA (Testing) phase.** This means the bug landed a protected branch, such as `fresh` , but was caught before reaching production.
- **v3: Caught in Production phase.** This means the bug actually shipped to end-users (i.e. it reached `stable`).

The second part of Volatility is optional, but may be useful to certain teams. *Origin* indicates which development stage the bug originated at.

- **oN: Not a bug/Unknown** This should be used for non-bug issues, and also if the origin cannot be determined.
- **o0: Originated in Design phase.** This usually means the bug is a logic error or impossible expectation that formed during the pre-coding Design process.
- **o1: Originated in Coding phase.** Almost all bugs are created during the actual code-writing process.
- **o2: Originated in SQA (Testing) phase.** For example, if a bugfix made at this stage causes another bug to form, this would be the origin.
- **o3: Originated in Production phase.** This usually means the bug was created during the process of preparing `devel` for shipment.

You can combine these two metrics to get the Adjusted Volatility [AV] score for any bug:

```
AV = v-o
```

The Adjusted Volatility allows you to account for how much opportunity developers had to *catch* the bug. For example, a mistake made during packaging is worth noting, but it isn't nearly as alarming as a bug introduced in the design phase, but not caught until after it shipped to users.

Volatility's true strength is in project management. See *Project Volatility Scoring* to learn how to calculate and use this metric.

Note: Volatility is based on the article How I Measured The Software Testing Quality and the subsequent comment chain.

2.3.3 Accomplishment

To get the best sense of what has been done by a developer in a given time period, we'd look at the average Gravity, Priority, and Friction.

Here is a table of examples of the system in action.

Legend: `measureTOTAL(AVERAGE))`

Tasks	Total G	Total P	Total F	Conclusion
5	g21(4.2)	p8(1.6)	f8(1.6)	Important (but probably easy) overall accomplishments, though few of them needed to be done now. A good week's work.
5	g8(1.6)	p21(4.2)	f8(1.6)	The tasks were urgent right now, but not important in the big scheme of things. Probably easy. A good week's work.
5	g15(3)	p15(3)	f23(4.6)	Moderately important tasks, all extremely difficult. A HUGE accomplishment.
20	g20(1)	p20(1)	f20(1)	A lot of tasks were done, but none were very urgent or important, and all were really easy. Not as impressive as the task count seems.

These numbers have to be taken in context with other factors, of course, but they give a MUCH more accurate picture than other management and tracking methods.

2.3.4 Project Volatility Scoring

The Volatility metric is most useful in catching problems within an overall project or team.

To calculate a project's Adjusted Volatility score, use the following equation:

```
A = project Adjusted Volaility score
M = project's Mean Volatility score
b = number of bugs
v = sum of all bug volatility scores
o = sum of all bug origin scores

A = (bv - bo)/b
M = v/b
```

You may want to record both the project's Mean Volatility (`M`) and Adjusted Volatility (`A`), as useful information can be garnered from both.

For example…

- A very high `A` indicates that many bugs are slipping past review processes.
- A high `M` and low `A` indicates that a lot of bugs are actually originating in SQL or Production phases.

Sometimes, tracking Origin just isn't useful for your team, in which case you can just use `M`.

CHAPTER

THREE

IN-HOUSE STANDARDS

These standards were written specifically for MousePaw Media.

3.1 Coding Standards

3.1.1 Repository and Versioning Conventions

Roles

The following roles are standardized in our repository and release management:

- Lead: The individual in charge of the department or a project. (Lead roles and hierarchy are defined in the MousePaw Media Employee Handbook, and are beyond the scope of this document.)
- Repository Master: A trained individual who is granted administrative control of the repositories and build system, and who is entrusted with deciding if and when to commit to protected branches.
- Trusted developer: All staff members, as well as any open source contributors who have been explicitly granted trust privileges by a staff member.

Branches

We reserve and protect three branches in our Git repositories:

- `devel` is for the latest development version.
- `fresh` is for the latest testing release.
- `stable` is for the latest stable release.

Any developer can push to `devel` with an approved code review from a Trusted developer. However, only a Repository Master can push directly to `devel`.

A Repository Master is the only individual who can push to `fresh` or `stable`, and thus must be the one to land (commit) any code which passes code review.

Versioning

Starting from January 2020, we use Semantic Versioning for all software development projects. Our version numbers follow the format `X.Y.Z`

- `X` is a major release, reserved for changes to the API or CLI.
- `Y` is a minor release, for releases with new features.
- `Z` is a patch release, for bugfix-only releases.

Note: Fixing a bug in a non-released version does not qualify for a patch release number! For example, if a bug is fixed in `2.4.0-beta2`, a bugfix would create `2.4.0-beta3`, and *not* `2.4.1-beta3`.

About Zero-Versions

Zero-Versions are versions at Major Release `0`. They are considered inherently unstable and feature incomplete, and thereby un-releasable. They cannot be promoted to either `fresh` or `stable`.

A project remains at a Zero-Version until it is considered complete and stable enough to promote to ALPHA. The Major version remains at `0`, but the Minor and Bugfix versions can be incremented at the discretion of the Lead and Repository Master.

When a version is deemed complete and stable enough to promote to an ALPHA, the version number immediately jumps from `0.N.N` to `1.0.0-alpha.N`, and only `N` is incremented.

ALPHA Versions

ALPHA versions are committed to the `devel` branch. They are generally feature-frozen, and represent candidates for promotion to `fresh`.

In versioning, `-alpha.N` is amended for ALPHA-versions, where `N` is the ALPHA version number. For example, in preparation for release of 2.4.0, the first candidate of the development code for promotion to `fresh` would be version `2.4.0-alpha.1`, and should be tagged as such.

If an additional feature is to be added to an ALPHA, it is kicked back down to a Development version, and the ALPHA suffix is dropped. If it was promoted to `1.0.0-alpha.N` from a Zero-Version, it will be reverted to the next appropriate Zero-Version until the feature is considered stable enough to warrent another ALPHA Release.

Todo: Determine ALPHA candidacy/testing standards.

BETA Versions

BETA versions are committed to the `fresh` branch. They are feature-frozen, and are intended for testing, but are not yet considered stable enough to be a "release candidate".

In versioning, `-betaN` is amended for BETA-versions, where `N` is the BETA version number. For example, after `2.4.0` is first promoted to the `fresh` branch, it would be tagged `2.4.0-beta.1`.

Additional features *may not be added* to a BETA. Once a version is promoted from an ALPHA to a BETA, it should be considered feature-frozen, for better or for worse. If the feature is considered to be a blocking feature to the Release,

the Release cycle should be *abandoned*, and the version kicked back down to a development version on `devel` . It will then gain a new Minor release number before it is promoted to ALPHA and BETA again.

Todo: Determine BETA candidacy/testing standards.

Release Candidates

When a BETA version is believed to be stable enough for release, it would be promoted to a Release Candidate (RC). It would remain in `fresh`, but would be considered a candidate for release to `stable`.

In versioning, `-rcN` is amended for RC-versions, where `N` is the Release Candidate number. For example, if `2.4.0-beta.6` is believed to be stable after field testing, it should be promoted to `2.4.0-rc.1`.

A Release Candidate is feature-frozen in the exact same manner as a BETA. However, if a Release cycle is abandoned due to a missing feature, the candidacy and testing processes should be audited.

Todo: Determine RC candidacy/testing standards.

Releases

Once a Release Candidate is confirmed to be stable enough for release, it should be promoted to `stable`, and the suffix dropped, leaving only the `X.Y.Z` version.

If a Lead Developer or Repository Master believes a Major or Minor Release is stable enough to skip the ALPHA or BETA phase, they may do so. However, all major and minor releases should always be tested at Release Candidate phase before final release.

Patch Releases can be expedited directly to Release Candidate or Release phase by a Lead Developer or Repository Master. Care should be taken in making this call, however, as some bugfixes can break other things.

Todo: Determine Release candidacy/testing standards.

Build Numbers

If build numbers needed, such as during Debian packaging, the build number may be appended to the version in the format `+YYYYMMDDHHMMSS`, where `YYYY` is the year, `MM` is the two-digit month number, `DD` is the two-digit day number, `HH` is the two-digit hour in military time UTC, and `MM` and `SS` are the two-digit minute and second respectively.

The build metadata and script files *are* considered part of the project, and are versioned as Patch releases. Assuming the build metadata and script files are unchanged, the build itself should *never* increment the build; in that case, only the build number should be updated.

3.1.2 C and C++

File Types

- C++
  - Headers: `.hpp`
  - Implementation: `.cpp`
- C
  - Headers: `.h`
  - Implementation: `.c`

The reason behind this is so we can use C and C++ in parallel with one another, without confusing what language any given file is written in.

Naming Conventions

- Variables: `lower_snake_case`
- Constants: `SCREAMING_SNAKE_CASE`
- Functions: `lower_snake_case`
- Classes: `UpperCamelCase`
- Filenames: `lower_snake_case`

Formatting

Note: We are currently debating whether to switch to OTBS.

- Use Allman bracketing and indentation style.

```
if (x == y)
{
    x++;
    foo();
}
else
{
    x--;
    bar();
}
```

- Avoid code beyond 80 characters. Never exceed 120 characters.
- Indentation should use tab characters (4 space width), with spaces for further alignment.
- A switch's block should be indented, and each case's block should be indented in the same manner.
- Line up lists so a) the start of list item lines align, and b) the end of list item lines roughly align. Each line should end with either a comma or, in the case of the last line, the closing token.

```
string names[9] = {"Bob", "Fred", "Jim",
                   "Chris", "Dave", "Jack",
                   "Ozymandius", "Randall",
                   "Andrew"};
```

- Pointer and reference indicators * and & should be aligned to the type part of the statement, not the name.
- Insert space padding around operators.
- Insert space padding after parenthesis headers (after `if`, `switch`, etc.)
- One-line blocks (i.e. one line `if` statements) should still have brackets.

Comments

- Use CSI Commenting Standard.
  - All functions and variables should have a doc comment at declaration.
  - CSI comment *every logical block*.
  - Header files and standalone implementation files should *always* have CSI-style description and license comments at the top.
- Use `//` and `/* ... */` for CSI comments.
- When a comment spans multiple lines, prefer multiline `/* ... */` comments. We recommend using line-leading ``*``.

```
/* This is a multiline comment
 * that spans multiple lines.
 * See how nice this looks?
 */
```

- Use `///` and `/** ... */` for doc comments.
  - Each parameter description in doc comments should be preceded by `\param` on a new line.
  - The return description in doc comments should be preceded by `\return` on a new line.
- Do not commit commented out code.
- Avoid inline comments whenever possible.
- Use `//TODO`, `//NOTE`, and `//FIXME` notation where necessary.

Structure

- `main.c` and `main.cpp` should reside in the root directory.
- `.h` and `.hpp` files should be in an the `include/` directory. For libraries, header files should be in a `<project>` subfolder (i.e. `include/anari/` or `include/pawlib/`).
- `.c` and `.cpp` files should be in the `src/` directory.
- Documentation files should be in the `docs/` directory.

3.1.3 Python

Based on PEP8 and PEP257.

Naming Conventions

- Variables: `lower_snake_case`
- Constants: `SCREAMING_SNAKE_CASE`
- Functions: `lower_snake_case`
- Classes: `UpperCamelCase`
- Filenames/Modules: `lower_snake_case` (Underscores discouraged, however. Avoid when possible.)

Formatting

- Four-space indentation ONLY.
- Avoid code beyond 80 characters. Use \ as necessary to break lines. Never exceed 120 characters.
- Line up multi-line structures as follows, with the opening and closing brackets on separate lines, and the start of the items lined up. Each item *may* be on its own line, but this is not required.

```
names = [
    "Bob", "Fred", "Jim",
    "Chris", "Dave", "Jack",
    "Ozymandius", "Randall",
    "Andrew"
]
```

Comments

- Include docstrings for all functions, classes, and modules, following PEP257
- Please avoid inline comments. Comment above lines.
- Use single line comments when possible. (`#`)
- Please comply with the CSI Commenting Standard as much as possible.
- Use `#TODO`, `#NOTE`, and `#FIXME` notation where necessary.
- All files should precede with CSI-style description docstrings and license comments.
- Do not commit commented out code.

Python Code Formatter

black should be used as the code formatter.

3.2 Licensing Standards

There are two different types of licenses covered in this document - **software licenses** and **content licenses**.

"Inbound" means a license on someone else's intellectual property, which we are using in some form. "Outbound" refers to a license we place on our own intellectual property.

3.2.1 Software Licenses

These licenses relate exclusively to source code and software libraries.

Inbound License Criteria

Any acceptable inbound software license must meet the following criteria:

- The license must meet the Open Source Definition, and be approved by the Open Source Initiative (OSI).
- The license must be permissive. This means that no restrictions are placed on derivative works in regards to licensing or usage.
- The license must be GPLv3 compatible.

Inbound Source Licenses (Short List)

The following licenses are already verified as meeting our criteria, and thus, are fully usable. This list is not exhaustive. Other licenses may be usable as well.

- Apache 2.0
- BSD-2/FreeBSD
- BSD-3
- Free Public License 1.0.0
- MIT License
- University of Illinois/NCSA Open Source License
- SIL Open Font License (for fonts)
- zlib/libpng

Note: The GNU Lesser General Public License v3 (LGPLv3) is usable if we are only linking against the LGPLv3-licensed library. Under no circumstances may LGPLv3 source code be used directly. We do prefer to avoid this license when possible.

Outbound Licenses

At this time, there are two licenses which we use for our code.

The BSD 3-Clause License is to be used for all code which shall be permissively open source. This supersedes our prior use of the MIT License. This license was selected for the third clause, which guards against implied endorsement.

The GNU General Public License (GPL v3) is used for libraries which we want to offer under a dual-license setup. Other GPL projects may be use GPLv3 code, but no other code may.

Open Source/Free Software Stance

MousePaw Media completely and publicly endorses the platform and goals of the Open Source Initiative, which we aim to further support and promote through our company's activity. We want to encourage the widespread and dominant adoption of open source software and open standards, especially in the sphere of education.

We do NOT endorse the strict anti-proprietary stance of Free Software Foundation or said intent behind its licenses.

3.2.2 Content Licenses

These licenses relate to anything that is **not** source code, including but not limited to graphics, music, audio, text, and video.

Inbound Licenses

We can use any material which has one of the following licenses:

- Apache 2.0 (for fonts)
- Creative Commons 0 (CC0)
- Creative Commons Attribution 1.0 (CC-BY-1.0)
- Creative Commons Attribution 2.0 (CC-BY-2.0)
- Creative Commons Attribution 3.0 (CC-BY-3.0)
- Creative Commons Attribution 4.0 (CC-BY-4.0)
- SIL Open Font License (for fonts)
- Public Domain

Outbound Licenses

We may wish to make some of our material publicly available via Creative Commons. When we do so, we use one of the following licenses.

Important: The licensing terms of any content must be approved by Anne McDonald or Jason C. McDonald! A large majority of our content is copyrighted, as that is essential to our ability to sell our products.

We use Creative Commons Non-Commercial No-Derivatives 4.0 (CC-NC-ND-4.0) when we want to make content freely available, but do not want it modified or used in commercial projects. This licenses allow teachers to use the content directly, as long as they don't modify it.

We use Creative Commons Non-Commercial 4.0 (CC-NC-4.0) when we want to be make content freely available to use and modify, but do not want it used in commercial projects. This license also accommodates use in schools, but it additionally allows for the material to be used creatively.

We use Creative Commons Attribution 4.0 (CC-BY-4.0) when we want to allow anyone to use the content freely, for any project they want, even commercially. Because we rely so heavily on this license for inbound content, it is only fair that we give back some content as well.

3.3 Software Standards

3.3.1 Legend

Mandatory software is in boldface. Mandatory software must be installed and configured on your machine, and you should be prepared to use it as needed. You *may* use alternatives, as long as your work is fully compatible with the mandatory software. Documentation assumes you're using the mandatory software, unless otherwise noted. In-house support may be limited for workflows with non-mandatory software.

Recommended alternative software is in regular face. If an alternative to mandatory software is needed or desired, we recommend these be considered first. In-house support is still generally available.

Optional alternative software is in italics. In-house support for these is limited.

† Transitional only, not for final file output.

‡ Limited support, no in-house tech support.

$ Proprietary, permitted but not officially supported by the company. Should be used transitionally.

3.3.2 Software Recommendations

Office Software

Office Suites

- **LibreOffice 6**
- *Calligra*
- *OpenOffice.org* ‡

Documents

- **LibreOffice Writer 6**
- *AbiWord*
- *Calligra Words*
- *OpenOffice.org Writer* ‡

Spreadsheets

- **LibreOffice Calc 6**
- *Calligra Sheets*
- *OpenOffice.org Calc* ‡

Presentations

- **LibreOffice Impress 6**
- *Calligra Stage*
- *OpenOffice.org Impress* ‡

Flowcharts

- **Dia**
- LibreOffice Draw 6
- *Calligra Flow*
- *OpenOffice.org Draw* ‡

Office Database

- **LibreOffice Base 6**
- *Calligra Kexi*
- *OpenOffice.org Base* ‡

PDF Annotation

- **Xournal**

Math and Formulas

- LibreOffice Math 6
- GeoGebra
- ZeGrapher
- *OpenOffice.org Math* ‡

Collaborative Editing

- **Visual Studio Code: Live Share** (text/code only)
- Etherpad [DevNet] (text/code only)
- *AbiWord* ‡

Desktop Publishing

- **Scribus**
- LibreOffice Writer 6
- *Calligra Words*
- *OpenOffice.org Writer* ‡

Mindmapping

- Calligra Braindump
- Freemind

Desktop Email Client

- Evolution
- Thunderbird
- *Geary*

Web Browser

- Brave
- Firefox
- *Chromium*
- *Opera*
- *Vivaldi*

Emulator

- VirtualBox

Graphics Design

Image Manipulation

- **GIMP**

Screenshots

- FlameshotJS
- *Shutter*

Raster Graphics

- GIMP
- Krita (Calligra)

Vector Drawing

- **Inkscape**
- *Calligra Karbon*

Photography

- **Darktable**
- *RawTherapee*

Image Conversion

- Converseen

3D Design

- **Blender**

Video Editing

- **Kdenlive**
- *Shotcut*
- *Pitvi*
- *OpenShot*
- *Roxio NXT Creator 2* $

Audio/Music

Recording and Editing

- **Audacity**
- *Ardour DAW*
- *Apple GarageBand* $†

Music Creation

- LMMS
- Hydrogen
- Garritan $†
- *Apple GarageBand* $†

Music Score

- **MuseScore**

Programming

Text Editor

- **Visual Studio Code**
- Atom
- Geany
- *KATE*
- *Nano*
- *Vim*
- *Brackets* ‡
- *Emacs* ‡
- *Sublime* $‡

Build Tools

- **CMake**

C/C++ IDE

- **Visual Studio Code**
- *Atom*
- *Code::Blocks*
- *Geany*
- *Vim*
- *Anjuta* ‡
- *Brackets* ‡
- *CodeLite* ‡
- *Eclipse CDT* ‡
- *Emacs* ‡
- *Kdevelop* ‡
- *Netbeans* ‡
- *Sublime* $‡

C/C++ Debuggers and Dynamic Analysers

- **Visual Studio Code** (debugging frontend)
- **gdb** or **lldb**
- **KCacheGrind**
- **Valgrind**
- Bless Hex Editor
- Nemiver
- Sysprof

C/C++ Static Analysers and Formatters

- **clang-format**
- **cppcheck**
- AStyle
- cccc

C/C++ Testing

- **Goldilocks**

Containers

- **Docker**

RestructuredText IDE

- **Visual Studio Code**
- *Atom*
- *Geany*
- *Vim*
- *Brackets* ‡

Collaboration/Pair Programming

- **Visual Studio Code: Live Share**

Python IDE

- **Visual Studio Code**
- *Atom*
- *Geany*
- *NINJA-IDE*
- *PyCharm Community Edition*
- *Vim*
- *Aptana* ‡
- *Brackets* ‡
- *Emacs* ‡
- *Eric* ‡
- *Pydev* ‡
- *Kdevelop* ‡
- *Spyder* ‡
- *Sublime* $‡

Python Debuggers and Dynamic Analysers

- **pdb**
- *pudb*

Python Static Analysers and Formatters

- **black**
- **flake8** (includes PyFlakes, pycodestyle, mccabe)
- bandit
- flake8-bandit
- flake8-datetimez
- flake8-docstrings
- flake8-pytest
- flake8-mypy
- flake8-regex
- flake8-requirements
- mypy
- pydocstyle
- *pylint*

Python Testing

- **pytest**
- **ward**

Version Control Software

- **Git**
- **Arcanist**
- **Meld**
- Git Cola

Operating Systems

- Ubuntu
- *Debian*
- *Kubuntu*
- *Linux Mint*
- *Lubuntu*
- *Ubuntu Studio*
- *Xubuntu*
- *Arch Linux* ‡
- *Fedora* ‡
- *Windows 10 with WSL* ‡

3.3.3 Disallowed Software

Note: This list doesn't apply to open source contributors, although we strongly discourage use of the software below.

Why Disallow Software?

The idea of "Officially Disallowing" software for company use might initially seem to be overkill, but there is a logic to it. The decision is, again, not made lightly. In most cases, the software title in question contains security and privacy issues, bugs, or compatibility issues that make its use a significant business and development risk. In other cases, the software is disallowed on grounds of licensing issues. Paying several thousand dollars extra for commercial licensing is impractical when there is equivalent open-source software available.

It is worth noting that, while not the sole factor, drastic conflicts in business ethics were also taken into consideration. MousePaw Media is built around the conviction that educational and creative technologies should be accessible to everyone, not just those with a lot of money. Relying on corporations whose business practices are at stark odds with this ethic is, frankly, counter-intuitive.

Disallowed Software List

The following may NOT be used for company purposes, under any circumstances, unless otherwise noted or unless special permission is given by a supervisor. If you need more details, talk to Jason C. McDonald directly. (You are welcome to use these for personal reasons all you want.

Adobe

No Adobe products may be used for development, due to licensing costs, file-type compatibility, and ethical concerns. (All useful Adobe products have an open-source equivalent in our present standards.) This includes Adobe Flash.

Autodesk

No Autodesk products may be used for development, due to licensing costs, file-type compatibility, and ethical concerns. (All useful Autodesk products have an open-source equivalent in our present standards.) This includes AutoCAD, 3ds Max, Maya, and Sketchbook.

Existing files may be opened with personal copies of Autodesk software for review and export purposes only.

MP3 File Format

Due to licensing and patent concerns, the MP3 format may NOT be used for any audio.

EXCEPTION: A copy (NOT the master) of the audio may be saved as an MP3 for compatibility with third-party services and software. Distribution in MP3 is only allowed if the distribution platform strictly requires that format.

Microsoft Internet Explorer

Due to serious security and performance issues, Microsoft Internet Explorer is NOT to be used under any circumstances for company purposes, including (but not limited to) accessing the staff network, company-commissioned web design, or accessing any website for work-related reasons.

Microsoft Office

Due to some ODT compatibility issues, and a lack of in-company training and support, Microsoft Office is NOT to be used on any company documents.

Trimble SketchUp (formerly Google SketchUp)

Due to licensing costs, SketchUp may only be used for internal idea drafting, and is strongly discouraged even for this.

Microsoft Windows

As of 2019, due to revisions in the Terms of Service and Privacy Policies for Microsoft, we have lifted the ban on Microsoft Windows. However, **we still require Linux for development work**. In circumstances where Linux is directly uninstallable on a work machine, Windows 10 with Windows Subsystem for Linux may be used.

3.4 Technical Standards

3.4.1 Languages

C++

Standard: C++17 ISO Standard

Compiler: Clang 9 (9.0 or later) or GNU GCC 7 (7.2 or later)

Windows Compiler: Clang (see above), either native or via WSL.

CSS

Standard: W3C CSS

Version: CSS3

Encoding

Standard: UTF-8

HTML

Standard: W3C HTML5

Version: HTML5

Python

Standard: PEP 8

Version: 3.7.x

XML

Standard: W3C XML

Version: 1.0

3.4.2 Libraries

Library	Use	Version	Binding	Source	License
Cairo	2D graphics	1.16.0	C/C++	Package: `libcairo2-dev`	LGPL/MPL
CPGF	Reflection, Introspection	1.6.0	C++	Repo: `libdeps`	Apache v2
Eigen	Linear algebra	3.3.1	C++	Repo: `libdeps`	MPL2
eventpp	Signals	0.1.0	C++	Repo: `libdeps`	Apache v2
Opus	Audio codec	1.3.1	C++	Repo: `libdeps`	BSD-3
PySide2	GUI	5.11.x	Python	Pip: `PySide2`	LGPL
pugixml	XML parsing	1.8	C++	Repo: `libdeps`	MIT
SDL	Hardware abstraction layer	2.0	C/C++	Package: `libsdl2-dev`	zlib

3.4.3 Operating System Support Targets

These show minimum specifications for a goal.

Support Goals:

- 5: Essential
- 4: High Priority
- 3: Moderate Priority
- 2: Low Priority
- 1: Collateral Support
- 0: No Support

OS	Memory	Goal
Microsoft Windows		
Windows 95	256 MB	2
Windows 98	256 MB	3
Windows 2000	256 MB	3
Windows ME	256 MB	3
Windows XP SP1	256 MB	4
Windows XP SP2	512 MB	5
Windows XP SP3	512 MB	5
Windows Vista	512 MB	5
Windows 7	1 GB	5
Windows 8	(Any)	1
Windows 8.1	(Any)	1
Windows 10	(Any)	5
Apple Mac		
OS X 10.0-10.3 PPC	256 MB	2
OS X 10.4 PPC	256 MB	4
OS X 10.4 Intel	256 MB	4
OS X 10.5	512 MB	3
OS X 10.6	1 GB	4
OS X 10.7-10.10	2 GB	4
Linux		
Ubuntu LTS	512 MB	5
Debian	512 MB	4
Puppy Linux	256 MB	3
Fedora	1 GB	4
OpenSUSE	1 GB	3
SoaS	1 GB	3

For support consideration:

- Guadalinux
- Qimo4Kids

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

CHAPTER

FIVE

FEEDBACK

These standards are written and maintained by the MousePaw Media Standards Board. Feedback is welcome via email (developers@mousepawmedia.com).

You can also frequently find us in the #mousepawmedia channel on Freenode.

Pull requests to this repository are not accepted. If you wish to propose a change, email your patch to us at the address above.

For more information about contributing to MousePaw Media projects, see mousepawmedia.com/contributing.