Software development design guidelines

Foreword to editors of this document

If you add or edit existing text, please mark your edits with either different color or place your name before new text - for example:

(Tarmo) This should be done like that.

Or

This works ~~this way.~~ that way. (Tarmo: not correct)

Later on these edits will be reviewed or discussed and accepted to main document.

Do not place destructive or irrelevant comments - better to write - I don't agree with this whole statement - recommended to remove, because ... . Notice that your own vision / understanding of problem in text might be better than useless comments or existing text in this document.

Overview

Main purpose of this document is to improve source code and code design quality - that's modularity, testability, effectiveness and maintainability – but nevertheless you should always use common sense when, how and to what to

apply given document. Also this document itself is not designed to be final, ultimate and common – it's recommended also in future to extract best design approaches and rules – update this document as well.

1. Identify design requirements

Identify what are mandatory and what are optional requirements for your software. Requirements tends to change during product lifetime, and it's almost impossible to predict future. Live with current day and currently identified mandatory requirements. More complex requirements might require more serious restructuring of application (re-design, re-factoring, re-architecting) – and it's easier to restructure program which does not have heavy structure already (has less abstraction layers, glue or wrappers). It's good to take challenging requirements, but not "too" challenging - but requirements is always compromise of functionality, features, development time, and quality.

You can also introduce metrics to improve the project manageability: it makes the QA and planning easier and might help to identify the bottlenecks earlier. For example, the minimal frame-per-second on specific hardware, the latency of the client-server network communication, the minimal number of simultaneous connection supported etc.

2. Design more complex parts first

Identify first elements of your software which you consider as challenging to you. That might be unexplored, unknown area for you – something you don't have knowledge or expertise on. Try to make some sort of working prototype first. Use any coding or design technique which will help you to create non-heavy working prototype. It's acceptable for prototype to have incomplete error handling, but when starting to commit code to git or svn – you still should polish it to be in acceptable shape. It's acceptable to discard prototype and try other techniques as long as it does not take too much of your time. If you cannot identify complex parts – it makes sense to start prototyping from lower layers of your system, and then move up to upper layers (which use / call lower layers).
After lower layer parts are roughly designed, you should go to higher layer. Better to make lower layers design in cooperation with higher layer usage - so higher layer is easy to use, and all relevant error messages (and exceptions) are propagating to higher level.

Please consider existing solutions, developing and maintaining the own code in many cases is more expensive than using the 3rd party solutions. For example, if the application uses message queues to support the producer/consumer pattern, it might be tempting to just write a small class for this, however when the requirements change to use several processes or even distributed computing, the initial solution may fail and a lot of development efforts will be required to implement this. At the same time, solutions like ZeroMQ or RabbitMQ may help with it regardless of the usage. Surely, prototyping and research is still required, and maybe certain middle layers could be used to eliminate the hard dependency of the 3rd party solution.

3. Design recommendations

- Test each code branch.

  Any code snippet written by you can contain errors – so it makes sense to try to write as little code as possible with potential testing of each code branch. (Each if, each else, each for loop and so on) Try first to test shortest code paths – for example error handling, missing files / wrong input data, and at the end longer

code paths. Each code path should be reachable (avoid writing code which is never executed).
If you're confident that your error handling will work without any problem - then testing is not needed.

- Prototype: Create small test applications

  Create quick prototype applications (for example console, winforms, etc) – for each area which you don't understand – try to learn how some individual class or function works – what are input / output parameters, what are potential exception which each code snippet can throw. Please use unit-tests if possible, in that case if 3-rd party API changes, unit-tests may help to discover it earlier. Those unit-tests may run during continuous integration (regardless of do they belong the core project or side prototype project).

- End-user driven error handling

  Error handling must be driven from end-user perspective – besides succeeded / failed status / error code – there always should be error explanation with relevant to error parameters. For example – if failed to open particular file – then file name and its path should appear in error message. Avoid generic errors "Login failed", "Open file failed", "Internal error".

  Error explanation is simple for end-user to understand and potentially to solve - so he does not need to contact support team.

  It could be useful to record stack trace, error specific parameters and be able to report additional error information back to development team. If error is easily reproducible, you may activate extra error diagnostic information using for example special "developer known" registry keys or special keyboard combination - for example Ctrl+Shift+Enter. This way it's possible to report clear error to end-user, but also possible to get extra diagnostic information in case of failure.

- Minimize debug time.

  Focus on time in each debug session – try to minimize it. Use logging / hidden from end-user functionality in order to catch more complex problems. If time to develop still grows larger (for example because of database access or heavy file operation) – try to simplify / minimize your test case so you will spend less time in debugging. It's also sometimes possible to create glue / wrapper layers which mimic some particular API interface. Try to design all code as final one – no need to return and fix it again.

- Comment code.

  Keep such comments in code so you can remember after 5-10 years what you wrote previously. Don't comment trivial / self-explaining piece of code, but do comment everything that is not trivial to understand.

  Try to use self-descriptive function and class names when possible.

- Create new modules only when needed.

  Create new dll's / assemblies / classes / new abstraction layers only when absolute must. It's good not to keep same source code bigger than 1000 lines of code. If class, struct is trivial enough - it's ok to keep multiple classes in same source file - but keep in mind that it might be more difficult to find.

- Learn something new on weekly basis.

  Try to learn something new on weekly basis – but don't use too much time analyzing everything at once. Time spent in research / analysis without clear result don't directly benefit anyone. Some problems (like for example opengl object transparency) can be over complex – without clear success.

- Don't copy-paste same code over and over again (DRY Principe)

  Try to create new function instead. Other than that, templates, abstract classes, interfaces and refactoring are the main tools for keeping copy-paste at a minimum.

- Prefer error handling to exception handling

  Exception throwing and catching is slow and more difficult to handle - prefer error handling if possible. Uncommon error situations - like for example out of memory , end-user does not have permission to access my application config file - better to handle over exceptions. Please be consistent, if the 3rd party API uses various error handling, you may want to introduce the additional layer to make it consistent. In that case, the error handling may simplify development and less tempting to skip, For example, if all functions have signature: bool fn(...), you can do a cascade call: if (op1() && op2() && …) { success } else { error }.

- Avoid to reuse black box third party interfaces and modules for exactly designed for your needs

  Third party components are all components inherited as black box - source code upon we don't have access or we cannot influence its design - such modules are all Microsoft .NET code for example, or any 3-rd party component, like devExpress. Even if bug is found - developers might refuse bug fixing to it - for example because of backwards compatibility reasons. Use preferably properly designed white box source code. However, please consider the development and maintenance cost, certain functionality is difficult to implement. If you have any doubts, please use a middle layer to have the ability to switch the 3rd party library to another one or even your own if needed.

- Developer friendly order of functions

  Place most important function or which are first used at the top of file, helper classes, internal functions - closer to end of file. If you're reading a book, you read from first page - in order to understand what will happen on next pages. Similar approach in source code. (If programming language allows this kind of ordering).

- Reduce the amount of code

  The more code is written, the more problems it may have. Using various method to reduce the amount of the written code may improve the code quality and the cost of maintenance:

- Domain Specific Language (DSL) or scripting.

  While API is already some kind of DSL, the high level DSL may help a lot, for example, scripting in a game engine (even if it looks like script, it may work in two modes - interpretation for the testing, and compilation for the production); cross-language libraries (the DSL is translated to the required languages); machine-learning high level operations compiled into the native code for high performance etc

- Annotations.

  While annotations are usually used for the documentation and some compiler instructions, the developers may add their own annotations. One of the use-cases is an automatic unit-test generation - it may help with the trivial unit-tests and allows developers to save time and efforts. For example, for a math function a developer might add an annotation with the expected functionality like if arguments have certain values, the output should have the specific value too. Annotation parser will generate a unit-test with that condition, plus it may add boundary cases (like the function should not fail with the min and max values of its arguments).

- Avoid unnecessary complexity in code (KISS = Keep it simple, Stupid)

  If your application by its nature does not require complex paradigm - don't use use complex paradigm then. For example:

  - Desktop application uses event driven mechanism to operate. Event driven mechanism results quite often in complex state machines and makes code only more complex without justification. Why not to write everything by direct call approach ?!

  - Before introduce multithreading into application, make sure that you know what is eating application performance. Maybe you can recode code, so that performance gets on acceptable level without multithreading.

- Design patterns.

  Some design patterns are not obvious, however may help a lot. For example, carefully designed event-driven architecture may improve the error handling: instead of operating boolean values and global states, the developer may introduce various events (including the error ones) and a state machine. Instead of having high nested depth of if-else clauses, the logic could be flat ("chain methods and if any of them gives an error event, stop chaining and push the event forward", "if an error event received, show the error, otherwise chain other methods") and eventually be distributed if needed. Please note that the Option pattern may be applied to the described use-case too, but it has a slight overhead.

If cross class dependency hierarchy does not allow easy prototyping (Need to drag heavy class hierarchy in order to test one function) – then class hierarchy must be designed from testing perspective. Each layer in application should be possible to access using test application only for that particular layer.

Whether you should keep your test / prototype application in git / svn – it's up to you to decide. Sometimes test application is useful to update or test new functions created on same level / layer.

4. Bug fixing

You are making new feature or fixing a bug. During that time you find a new bug (which cannot be found in any existing error card). if bug fixing does not takes too long (less 0.5 day) - it's acceptable to fix it immediately - if it takes more time - better to (1) discuss about with with your manager or (2) raise new error card with bug description and pass it to your manager. Manager can prioritize urgency of bug and return given error card to you. For minor bugs approach (2) should be preferred.

If it's possible, please write a separate unit-test for each found bug - it will help to avoid bugs to be reintroduced ("regressions"). Please note that the unit-test should contain the original bug behaviour and an assertion that this is not happening anymore.

5. Development tools

Please note that the tools help enormously to improve the software quality. The main tools at your disposal should be (compilers, unit-testers with coverage and documentation tools are not described as obvious):

- Version control systems. Version control is used not only for the saving of the code, but to track changes back, restoring original functionality and making reports. Please set up a workflow for using the version control, for example: stable branch(-es) for the production code, dev branch for the code pending for QA, a branch per bug (to easily switch between bugs if got stuck) etc

- Code style checkers. A lot of code style checkers are customizable, and can be configured to enforce the organization style. While it may seem minor, the different code styles between various developers may lead to various issues (merge conflicts, time waste on reformatting the code just because one doesn't like it). It's much easier to enforce one style and make the enforcement automatic, so no additional code reviews are required.

- Linters. Everyone makes mistakes, so it's better to avoid it as early as possible. There are various linters, and sometimes several are required per project. Maybe it won't be possible to eliminate all issues at once, but the number of issues should be tracked and gradually decreased (there are several solutions that can apply the several checks at once, for example, SonarQube).

- IDE. While simple text editors may be enough, please don't underestimate the usefulness of IDE (including commercial one). A lot of them have automatic code style checkers, linters and other useful functionality that couldn't be gotten by the standalone tools. Please note that developers are mostly reading the code, not writing it, so navigation and other useful perks of IDE may really speed up the development process.
  For example:
  - IntelliJ IDEA: https://www.jetbrains.com/idea/
  - ReSharper C++: https://www.jetbrains.com/resharper-cpp/

- Code review systems. It's not necessary a standalone tool, even emailing patches may work, but it's very useful to review the code before moving forward. Certain systems (like Gerrit) allow to review the code before pushing into the development branch (a branch is created per review). Please note that the review is needed not only to discover the possible bugs, but to help others get familiar with the submitted code too.

- Continuous Integration systems. While developers may have their own tools to continuously build the code and run the unit-tests, it's not enough to find all the issues - integrations tests may take a while, so it's better to run them independently. Also there is a common problem with developers missing files in the commit. Clean checkout and full test cycle by the CI servers help with it and other requirements (like compiling the packages for various platforms and architectures).

- If there is a need to find some regression occurred long time ago, and you can't locate source code because of which error occurs, old binaries daily auto-backup can help out. (Must be enabled first before you can actually use it)

- If you need to quick search and check multiple web pages - linkclump (Chrome addon) might be useful to open multiple links simultaneously after google search. This is especially useful when you try to find documentation on some old or rarely used API, technology or concept.

- Tarmo: Sometimes Visual studio does not shows up whole call stack in call stack window - sometimes it's useful to use such tools as Process Hacker to check whole call stack of failing function.

6. Coding style

Overview

Typically nothing can replace code inspections / reviews, and here it's attempted to collect some good practices, which came from code review for future code reviews. It's recommended to update here more items, which pops ups during code review, but it's important that people more or less agree on new item.

(Tarmo) It's recommended by default to apply new modifications only to newly written code or modified functions (using common sense) - generally it should be avoided to rewrite whole code according to new coding style standards.

Shortest paths first

If you have many inner if-else branches, it makes sense to terminate shortest paths first using either '`return`' or '`break`' statements. Consider following code snippet:

```
void function( )
{
    if( statement1 )
    {
        DoThis(1);
        DoThis(2);
    } else {
        if( statement2 )
        {
            DoThis(3);
            DoThis(4);
            DoThis(5);
        } else
        {
            DoThis(6);
        }
    }
}
```

Shorter form of that function would be:

```
void function( )
{
    if( statement1 )
    {
        DoThis(1);
        DoThis(2);
        return;
    }

    if( !statement2 )
    {
        DoThis(6);
```

```
        return;
    }

    DoThis(3);
    DoThis(4);
    DoThis(5);
}
```

Don’t leave commented code

Professionals don’t leave commented code - it just puzzles reader of code - it that code intentional or is it just some non-tested functionality or remaining bug. Only reason to leave commented code is just some additional debug info, which can be enabled by recompiling code.