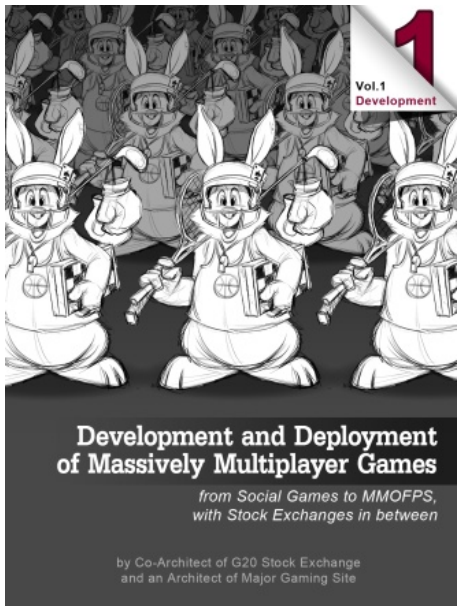
Pre-Coding Checklist: Things Everybody Hates, but Everybody Needs Them Too. From Source Control to Coding Guidelines

posted February 29, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter IX from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



We've discussed a lot of architectural issues specific and not-so-specific to MMOs, and now you've hopefully already drawn a nice architecture diagram for your multiplayer game.

However, before actually starting coding, you still need to do quite a few things. And I am perfectly aware that there are lots of developers. Let's take a look at them one by one.

Source Control System

"I don't want to go into a discussion *why* you need source control system, just saying that there is a consensus out there on it being necessary

To develop pretty much anything, you do need a source control system. I don't want to go into a discussion *why* you need source control system, just saying that there is a consensus out there on it being necessary for all the meaningful development environments. Even if you're single developer, you still need source control: the source control system will act as a natural backup of your code, plus being able to rollback to that-version-which-worked-just-yesterday, will save you lots of time in the long run. And if you're working in a team, benefits of source control are so numerous that nobody out there dares to develop without it.

The very first question about source control is "what to put under your source control system?" And as a rule of thumb, the answer is like

You should put under source control pretty much everything you need to build your game, but usually NOT the results of the builds

And yes, "pretty much everything" generally includes assets, such as meshes and textures.

On the other hand, as with most of the rules of thumb out there, there are certain (but usually very narrow) exceptions to both parts of this statement. For "pretty much everything" part, you MIGHT want to keep some egregiously large-and-barely-connected-to-your-game things such as in-game videos outside of your source control system (for example, replacing them with stubs), but such cases should be very few and far between. Most importantly,

the game should be buildable from source control system, and the build should be playable

, that's the strict requirement, other than that – you MIGHT bend the rules a bit.

For not including results-of-your-build – I've seen examples when having YACC-compiled .c files within source control has simplified development flow (i.e. not all

developers needed to setup YACC on their local machines), but once again – this is merely a very narrow exception from the common rule of thumb stated above.

Git

"Even I myself, being a well-recognized retrograde, has been convinced that git has enough advantages to qualify as "the way to go"

The next obvious question with regards to source control is "Which source control system to use?". Fortunately or not, there is a pretty much consensus about the-best-source-control-system-out-there being git.[1] Even I myself, being a well-recognized retrograde, has been convinced that git has enough advantages to qualify as "the way to go" for objective reasons (opposed to just being a personal preference).

Whether to host git server yourself or whether to use some third-party service such as github – is less obvious, especially for games. I would say that if by any (admittedly slim) chance your game is open source – you should go ahead with a github.

If your game is closed-source – then the choice becomes less obvious and depends on lots of things: from the size of your team to having on the team somebody who's willing to administrate (and backup!) your own git server (while it is certainly not a rocket science, it will involve some command-line stuff). Even more importantly, if your game has lots (such as multiple gigabytes) of assets – you probably should settle for an in-house git server, at least because downloading all that stuff over the Internet will take too much time.

---

[1] Actually, you can do more or less the same things with Mercurial, but unless you already have it in place, in most cases I suggest to stick with git, even in spite of Git's lack of support for locks, see "Git and unmergeable files" section below

Git and unmergeable files

For game development (and unlike most of other software development projects), you're likely to have binary files which need to be edited. More precisely, it is not only about binary files, but also includes any file which cannot be effectively merged by git. One example of these is Unity scene files, but there are many others out there (even simple text-based Wavefront .obj is not really mergeable).

A question "what to do with such files" is not really addressed by Git philosophy. The best way would be to learn how to merge these unmergeable files,[2] but this is so much work that doing it for all the files your artists and game designers need, is

hardly realistic ☹ .

The second best option would be to have a 'lock' so that only one person really works with the asset file at any given time. However, git's position with regards of locks is that there won't be mandatory locks, and that advisory locks are just a way of communication so that should be done out-of-git (!). The latter statement leads to having chat channels or mailing lists just for the purposes of locking (ouch!). I strongly disagree with such approaches:

all stuff which is related to source-controlled code, SHOULD be within source control system, locks (advisory or not) included

To use advisory (non-enforced) locks in git, I suggest to avoid stuff such as chat channels, and to use lock files (located within git, right near real files) instead. Such a lock file MUST contain the name (id) of the person who locked it, as a part of file contents (i.e. having lock file with just "locked" in it is not sufficient for several reasons). Such an approach does allow to have a strict way of dealing with the unmergeable files (that is, if people who're working with it, are careful enough to update – and push(!) – lock file before starting to work with the unmergeable file), and also doesn't require any 3rd-party tools (such as an IM or Skype) to work. For artists/game designers, it SHOULD be wrapped into a "I want to work with this file – get it for me" script (with the script doing all the legwork and saying "Done" or "Sorry, it's already locked by such-and-such user").[3] If you like your artists better than that, you can make a shell extension which calls this script for them.

The approach of lock-files described above is known to work (though creating commits just for locking purposes), but still remains quite a substantial headache. Actually, for some projects it can be so significant that they MIGHT be better with Mercurial and it's Lock Extension (which supports mandatory locking).

Let's note that there is also an issue which is often mentioned in this context, the one about storing *large* files in git, but this is a much more minor problem, which can be easily resolved (for example, by using git LFS plugin).

---

[2] Actually, for merging Unity scene files there was an interesting project [GitMergeForUnity], but it seems abandoned now ☹

[3] and of course, another script "I'm done with file", which will be doing unlock-and-push

Git and 3rd-party Libraries

"How to handle those 3rd-party libraries you're going to use?

One subtle issue with regards to source control system is how to handle those 3rd-party libraries you're going to use. Ideally, 3rd-party libraries should be present in your source control system as links-pointing-to-specific-version of the library, with your source control system automagically extracting them before you're building your game. It is important to point to a specific version of the library (and not just to head), as otherwise a 3rd-party update can cause *your* code to start crashing, with you having no idea what happened. On the other hand, such an approach means that it is *your* responsibility to update this link-to-specific-version to newer versions, at the points when you're comfortable with doing it.

git submodule does just that, and *git submodule update* will allow you to update your links to the most recent version of the 3rd-party library. However, it works only when your 3rd-party libraries are git repositories themselves

If your 3rd-party library is available as an svn repository instead of git – you may setup a git mirror of svn repository and then to use *git submodule* [StackOverflow.SvnAsGitSubmodule]. A similar trick can be used with Mercurial too (see [HgGitMirror] on creating git mirror from Mercurial).

And if you're using a non-open-source library – you'll probably need to put a copy of it under your source control system 🙁 .

BTW, about open-source and non-open-source 3rd-party libraries: there is another (even more important) issue with them, make sure to read "3rd-party Libraries: Licensing" section below.

Git Branching

"One of the things you need to decide when using git, is how do you work

As noted above, as a rule of thumb, you should be using git. And one of the things you need to decide when using git, is how do you work with branches. In pre-git source control systems, branching was a second- (if not third-) class citizen, and developers were avoiding branches at all costs. In git, however, everything is pretty much about branches, and in fact this ease-of-branching-and-merging is what gives git an advantage over svn etc.

IMO, most of the time you should be following the branching model by Vincent Driessen described in [GitFlow] and is known as "Git Flow". When you look at it for the very first time, it may

look complicated, but for the time being you'll just need a few pieces of it:

with branches

- *master* branch. As a rule of thumb, you should merge here only when milestone/release comes. All the commits to the *master* branch should come from merges from *develop* branch. Direct commits (i.e. commits which are not merges from *develop)* into *master* branch SHOULD NOT happen.
- *develop* branch. The branch which is expected to work. More precisely – it is usually understood as a branch that compiles and passes all the automated tests, though it is understood that no amount of automated testing can really guarantee that it is working. You should merge to *develop* branch as soon as you've got your feature working "for you" (and all the automated regression tests do pass). Direct commits (not from *feature* branches) into *develop* branch are usually allowed. On the other hand, leaving *develop* branch in a non-compilable or failing-automated-test state is a major fallacy, and you'll be beaten by fellow developers pretty hard for doing it (and for a good reason). Note that having *develop* branch temporary failing to compile or run tests (in a sequence like developer committed to *develop* – automated build failed – developer fixed the problem or reverted the merge[4]) is not considered a problem; it is leaving *develop* branch in unusable state for several hours which causes a backlash (and rightly so). More on it in "Continuous Integration" section below.
- *feature* branch. You should create your own *feature* branches as you develop new features. These *feature* branches should be merged into *develop* branch as soon as your feature (fix, whatever) is ready. Consider *feature* branch as your private playground where you're developing the feature until it is ready to be merged into *develop* branch. *Feature* branches are generally not required even to compile.
    - A note of caution: there is a breed of developers out there who prefer to live within their own *feature* branch for many weeks and months, implementing many different features under the same branch and postponing integration as long as they can. *This is a Bad Practice™,* and a rule-of-thumb of one-feature-for-one-feature-branch should be observed as soon as you've past your very first milestone.
    - It is also interesting and useful to note that modern source control systems (git included) tend to punish those who do their merges later. When both you and a fellow developer are working on your respective branches, and she got committed her merge 5 minutes before you, then it becomes *your* problem to

"There is a breed of developers out there who prefer to live within their own feature branch for many weeks and months, implementing many different features under the same

resolve any conflicts which may arise from the changes *both of you have made*. In most cases for a reasonably mature codebase, there won't be any conflicts, but sometimes they do happen, and

branch and postponing integration as long as they can

it is the second developer who becomes ~~a rotten egg~~ responsible for resolving conflicts

Print this profound truth in a 144pt font and post it on the wall to make your fellow developers merge their feature branches more frequently.

---

[4] not that revert of the merge in git is very peculiar and counter-intuitive, see [kernel.RevertFaultyMerge] for discussion

Continuous Integration

One thing which is closely related to source control, and which you should start using as soon as possible for any sizeable project, is Continuous Integration a.k.a. CI (not to be confused with Continuous Deployment, a.k.a. CD which is a very different beast and will be discussed in Chapter [[TODO]]).

"The basic idea behind Continuous Integration is simple: as soon as you commit something, a build is automatically run with all the tests you were able to invent by that time

The basic idea behind Continuous Integration is simple: as soon as you commit something (usually it applies to *develop* branch merges/commits as described above), a build is automatically run with all the tests you were able to invent by that time. If the build or tests fail – whoever made the "bad" commit, gets notified immediately.

In a sense, continuous integration is an extension of a long-standing practice of "night builds", but instead of builds being made overnight, they're made in real-time, further reducing the impact from "bad commits".

In general, continuous integration is almost a must-have for any serious development, how to implement it – is a different story.

In this regard, I tend to agree with [Bugayenko2014] that build-before-commit[5] is a better way to implement Continuous Integration than classical build-after-commit.

The problem with build-after-commit is the following. If (as in

nightly builds and with classical Continuous Integration) developers commit first, and only then automated build+test runs, then there is a risk that the build/test fails. And if it happens – there is a strong pressure to fix the commit-which-caused-failure (instead of reverting it) – in part, because of git peculiar behaviour when it comes to merge reverts (mentioned above). Which means that the whole team will stop development and will be working on the fix, ouch. This practice is very disruptive, and can easily introduce "commit fear" mentioned in [Bugayenko2014].

To address this problem, instead of running the build *after* commit has happened, you should make sure that your build is clean *before* committing.[6] It means that "faulty" builds never happen, yahoo!

> **"Instead of running the build *after* commit has happened, you should make sure that your build is clean *before* committing**

To follow "build-before-commit" approach within your CI system, you may want to do one of the following:

- create a VM image with your "build server" and give every developer a copy. Not that I really like this option, but it does exist.
  - it MUST be a responsibility of *every developer* to run a build+test before every commit to *develop* branch
- write your own script which takes your-*feature*-branch as a parameter, merges it with *develop* (without committing it yet!), builds, runs all the tests, and commits-the-merge-provided-that-everything-went-smoothly
  - it MUST be a responsibility of *every developer* to use ONLY this script for committing into *develop* branch
- use Travis CI as your Continuous Integration tool. Make all commits to *develop* branch ONLY via "pull requests" (they still should reside within your *feature* branches(!)).
  - in this case, Travis CI will report whether current pull request *would* build ok after merge [TravisPullRequests]. It MUST be a responsibility of every developer to check this "Travis OK" status *before performing merge* (at least in GitHub it is shown as a nice green checkbox near the pull request, if you have Travis integration enabled)
- use Rultor set of scripts (by the very same Bugayenko). *NB: I didn't try it, so I cannot vouch for it.*

---

[5] I don't agree with [Bugayenko2014] that "Continuous Integration is Dead", but I do agree that what he names "read-only master branch", and I name "building-before-committing" is a better way of doing things (though I consider it being yet another way to implement Continuous Integration, and not something radically different)

[6] an alternative would be to fix merge revert in git, and to rely on merge reverts

instead. However, as current behaviour is considered a feature rather than a bug, this is not going to happen any time soon

3rd-party Libraries: Licensing

One really important thing to remember when developing your game is that no 3rd-party library can be used without taking into account its license. Even open-source libraries can come with all kinds of nasty licenses which may prevent you from using them for your project.

In particular, beware of libraries which are licensed under GPL family of licenses (and of so-called "copyleft" licenses in general). These licenses, while they *do* allow you to use code for free, come with a caveat which forces you to publish (under the very same license) all the code which is distributed together with the 3rd-party library.[7] There are a few mitigating factors though. First, LGPL license (in contrast to GPL license) is not that aggressive, and usually might be used without the need to publish all of your own code (while changes to library code itself will still need to be published, this is rarely a problem). Second, if you're not distributing your server-side code[8] – then only the client-side code will usually need to be published (which tends to help a lot for web-based games). In any case, if in doubt – make sure to consult your legal team.

"Be careful with open-source projects which don't have any license at all

Another two things to be aware of in open-source projects, is (a) "something under license which is not a recognised open-source license (see [OpenSource] for the list of recognised ones), and (b) "something without any license at all" (you'll see quite a few such projects on github). (a) is usually a huge can of worms, and in case of (b) you cannot really use the project in any meaningful way (by default, everything out there is subject to copyright, so to use it – you generally need some kind of license).

On the other hand, anything which goes under BSD license, MIT license, or Apache license – can usually be used without licensing issues.

And of course, if you're using commercial libraries – make sure that you're complying with their terms (paying for the library does not necessarily mean that you are allowed to use it as you wish).

---

[7] in practice, it is more complicated than that, but if you want legally correct answers – you better ask your legal team

[8] distribution of server-side code may happen, for example, if you're selling your

server-side as an engine

Development Process

The next thing which you will need is almost-universally necessary (that is, unless you're a single-developer shop) and pretty much universally hated among developers. It is related to the mechanics of the development process. All of us would like to work at our leisure, doing just those things which we feel like doing at the moment. Unfortunately, in reality development is very far from this idyllic picture.[9]

For your game, you do need a process, and you do need to follow it. What kind of process to use – old-school project Gannt-chart-based planning with milestones, or agile stuff such as XP, Scrum, or Kanban – is up to you, but you need to understand how your development process is going to work.

"I am not going to discuss advantages and disadvantages of different processes here, as the associated debates are going to be even more heated then Linux-vs-Windows and C++-vs-Java holy wars combined.

I am not going to discuss advantages and disadvantages of different processes here, as the associated debates are going to be even more heated then Linux-vs-Windows and C++-vs-Java holy wars combined. Usually, however, you will end up with some kind of a process, which is (whether you realize it or not) will be some combination of agile methods; in at least two of my teams, we were using a combination of Scrum and XP long before we learned these terms 🙂 .

BTW, if you happen to consider Agile as a disease (like, for example, [AgileDisease]) – that's IMNSHO not because agile is bad per se, but most likely because you've had a bad experience dealing with an overly-confident (and way too overzealous) Certified Scrum Master who was all about following the process without even remote understanding of specifics of your project (and quite often – without any clue about programming). While I do admit that such guys are indeed annoying (and often outright detrimental for the project), I don't agree that it makes the concepts behind agile development, less useful even by a tiny bit.

One thing which should be noted about agile criticisms (such as [AgileDisease]), is that there is no real disagreement about *what* needs to be done; the sentiment in such criticisms is usually more along the lines of "we're doing it anyway, so do we need fancy names and external consultants?" To summarize my own feelings about it:

Do you need to have a well-defined development process?	**Certainly. All successful projects have one, even if it is not formalized.**
Do you need to have it written down?	Up to you. At some point you'll probably need some rules written down, but it is not a strict requirement.
Does your project need to be iterative?	**Certainly**
Do you need to have your iterations reasonably short (3 months being "way too much")?	**Certainly**
Do you need to name your iterations "sprints"?	Doesn't matter at all
Do you need to have your iteration carved in stone after it started?	It depends, pick the one which works for you at a certain stage of your project
Do you need to analyze how your iteration went?	**A good idea, whether naming it "iteration" or "sprint"**
Do you need to describe your goals in terms of 'use cases'/'user stories'?[10]	**Certainly**
Do you need to *name* them 'use cases'/'user stories'?	Doesn't matter at all
Do you need to name your project "Agile", or "Scrum", or <insert-some-name-here>?	Doesn't matter at all
Do you need a daily stand-up meeting?	Up to you, but often it is not so bad idea
	You SHOULD. It is damn important to

Do you need Product Owner (as a role)	**have opinion of stakeholders to be represented**
Do you need Product Owner as a *full-time* role?	Not necessarily, it depends
Do you need to name this role "Product Owner"?	Doesn't matter at all
Do you need Scrum Master (as a role)?	You will have somebody-taking-care-of-your-development-process (usually more than one person), whether you name it "Scrum Master" or not
Do you need a Kanban board?	Up to you
Do you need to use XP's techniques such as pair programming, merciless refactoring, test-driven development?	Up to you on case by case basis
Do you need a Certified Scrum Master on your team?	**Probably not**
Do you need an external consultant to run your Agile project?	**If you do – your team is already in lots of trouble**

Ultimately, whether you're using fancy names or not, your process *will* be a combination of agile processes, using quite a few agile techniques along the road. And it doesn't matter too much whether you're doing it because you read a book on agile, or because you've invented them yourself.

---

[9] it applies to *any* kind of development, game or not
[10] while they're not exactly similar, they're close enough for our purposes now

Issue Tracking System

Whether we like it or not, there will be bugs and other issues within our game. And even if there would be a chance that we wouldn't have any bugs – we'll have features which need to be added. To handle all this stuff, we need an issue tracking system.

> "Whether we like it or not, there will be bugs and other issues within our game.

If you're hosted on github, *and* your team is *really small* (like <5 developers) – you MIGHT get away with github built-in issue tracker. If you're hosting your own git server (*or* if your team is larger), you're likely to use some 3rd-party issue tracking. The most popular choices in this regard range from free Bugzilla, Trac, and Redmine, to proprietary (and non-free as in "no free beer") JIRA.

Which one is better – honestly, IMHO it doesn't matter much, and any of them will do the job, at least until you're running a 1000-people company[11] (in particular, all 4 systems above do allow to integrate with git).

One extra thing to think about in this regard is support for the artifacts used within your development process. Whether you want to use a Kanban board, Scrum "burndown chart", or a good old Gantt chart (or all of them together) – having these artefacts well-integrated into the same system which provides you with issue tracking can save you quite a bit of time. More importantly – it may help you to follow your own development process. So think about artifacts of your development process, and take it into account when choosing your issue tracking system. Also keep in mind that some of the plugins which implement this functionality (even for free systems(!)) can become pricey, so it is better to check pricing for them in advance.

On the other hand, this support-for-development-process-artifacts is only a nice-to-have feature of your tracking system; you can certainly live without it, and it only comes into play when all-other-parameters of your issue tracking system are about-the-same for your purposes. On the third hand 😉, these days issue tracking systems *are* pretty much about-the-same from purely issue-tracking point of view.[12]

---

[11] and if you do, you should look for a better source than this book for choosing your issue tracking system, as issue tracking is very far from being a focus here

[12] I realise how hard I will be beaten for this statement by hardcore-zealots-of-<insert-your-favorite-issue-tracking-system> but as an honest person I still need to say it

Issue Tracking: No Bypassing Allowed

There is one very important concept which you MUST adhere to while developing pretty much any software product:

ALL the development MUST go through the issue tracking system

It means that there MUST be an issue for ANY kind of development (and for each commit too). Granted, there will be mistakes in this regard, but you MUST have "each commit MUST mention its own issue" policy. The only exception to this rule should be if it is *not* a feature, but an outright bug, *and* the whole issue can be described by developer in the commit message.

It is perfectly normal for a BA to come into developer's cubicle and saying "hey, we need such and such feature, let's do it". What is not normal – is *not* to open an issue for this feature (before or after speaking to the developer). As for using e-mails for discussing features – I am against it entirely, and suggest to have an issue open on the feature, and to have all the discussion within the issue. Otherwise, 3 months down the road you will have lots of problems trying to find all those e-mails and to reconstruct the reasons why the feature was implemented *this way* (and whether it is ok to change it to a *different way*).

Even for a team of 5, for every change in the code, it is crucial to know *why* it has been made, and there should be one single source of this information – your issue tracking system.

Coding Guidelines

One last (but certainly not least) thing you should establish before you start coding, is coding guidelines for your specific project. In this regard my suggestion is *not* to copy a Big Document from a reputable source,[13] but rather start writing your own (initially very small) list of DO's and DON'Ts for your specific project. This list SHOULD include such things as naming conventions, and all the not-so-universal things which you're using within your project. More on naming conventions and project peculiarities below.

"One last thing you should establish before you start coding, is coding guidelines for your specific

Of course, your guidelines.txt file belongs to your source control system. And while you're at it – do yourself a favor and find for it the most prominent place you can think of (root directory/folder of your project is usually a pretty good candidate).

[13] this book included; in Chapter [[TODO]] there will be an example of my personal guidelines for C++, but as with any other source – don't copy it blindly

project

Naming Conventions

With naming conventions the situation is simple: it doesn't really matter which naming convention you use (myFunction() vs my_function() won't make any realistic difference, and debating it for hours is not worth the time spent). What *is* important though, is to do it uniformly across the whole project, so you should just quickly agree on *some* naming conventions and then adhere to them.

That being said, there is one thing in this regard which I actively dislike and which I am arguing against (on the basis that it reduces readability) – it is so-called "Hungarian notation". If you really really feel like naming your *name* variable as *lpszName* – the sky won't fall, but I suggest to drop these prefixes completely.

As for having some kind of naming convention for class data members – two popular conventions are *mDataMember* and *data_member_*, this is up to you whether to have such convention, it won't make that much difference anyway (that is, as long as you're using it consistently across the whole project).

Project Peculiarities

For pretty much every project you will have some peculiarities. For example, as we'll be programming within our ad-hoc FSMs, then threads will be pretty much out of question (at least outside of well-defined areas) – ok, so let's write it down into our guidelines.txt file (to the part which tells about FSMs). For a C++ project there is a common question whether you'll be using printf() or ostream for formatted output and logging – regardless of your decision,[14] it needs to be consistent for the whole project, so it also belongs to Code Guidelines. And so on, and so forth.

"For a C++ project there is a common question whether you'll be using printf() or ostream for formatted output – regardless of your decision, it needs to be

For C++, my personal set of Coding Guidelines will be discussed in Chapter [[TODO]], but as with any other 3rd-party source, you shouldn't copy it blindly and should develop your own one, based on your own task, your own style, and your own design decisions.[15]

[14] FWIW, my answer is 'neither – use cppformat instead', see Chapter [[TODO]] for further discussion

[15] roughly translated as: "whatever nonsense I write there, it is your responsibility to filter it out, so don't blame me if it doesn't work for you" 😉

consistent for the whole project

Per-Subproject Guidelines

One important thing to be mentioned here is that most of the projects will actually need more than one set of coding guidelines. Not only the subprojects can be written in different programming languages, but also subprojects can perform very different jobs, what in turn requires different guidelines.

For example, even if all your code is written in C++, the guidelines for infrastructure layer (the one outside of FSMs) and application layer (implementing FSMs) is going to be quite different. The former is going to use threads, will probably provide logging facilities so it will need to have direct file access (and probably access OS-specific services too), etc., and the latter is basically going just to call whatever-is-provided-by-infrastructure layer (concentrating on game logic rather than on "how to interact with OS").

As a result, I strongly suggest to use different guidelines for different layers of your game even if all of them are written in the same programming language; at the very least, they should be quite different between 3D engine, network engine, and game logic.

Enforcement and Static Analysis Tools

All the rules and guidelines are useless if nobody cares to follow them. Even if it is only *somebody* who ignores the guidelines, if such ignoring-guidelines-code is not rectified soon enough, it is often used as an example for some other piece of code, and so on, and so forth, which means a slippery road towards most of the code ignoring the guidelines ☹ .

To deal with *all* such guideline violations, there is no real substitute for code reviews. However, to catch *some* of them, it is usually a good thing to use an automated tool which will complain about most obvious violations. Such tools are specific to the programming language; list of such "static analysis" tools which (as I've heard, no warranties of any kind) work in real-world projects, include:

- checkstyle (Java). Checks for naming convention compliance etc.
- astyle (C/C++/Objective-C/C#/Java). Re-formats your source according to your preferences. Personally, I like to have a policy of "before committing to *develop* branch, all the code should be run through astyle".
- StyleCop (C#).

- cpplint (C++). Style checks against Google C++ style guide. *Not to be confused with lint.*

Actually, static analysis tools go much broader than mere style checking, and quite a few of them can find bugs. Most popular static analysis tools in this regard include:

- cppcheck (C++)
- PMD (Java)
- PC-lint (C/C++). Commercial.

There are also lots of other static analysis tools out there (see [Wikipedia.StaticCodeAnalysis.Tools]), but quite a few of them are known to cause more trouble then provide benefits (one of common problems of many tools is having too many false positives), so don't hold your breath until you tested the tool and see that it works for you.

[[This Concludes Vol.1 "Architecture". To Be Continued in Vol.2 "Development"…

This concludes beta Chapter IX from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Moreover, this concludes "beta" of the whole vol.1 "Architecture", yahoo! Further chapters from vol.2 "Development" will be published soon…]]

[–] References

[GitMergeForUnity] "GitMerge for Unity"
[StackOverflow.SvnAsGitSubmodule] "Is it possible to have a Subversion repository as a Git submodule?", StackOverflow
[HgGitMirror] "Create a Git Mirror", hg tip
[GitFlow] Vincent Driessen, "A successful Git branching model"
[kernel.RevertFaultyMerge] kernel.org, https://www.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.txt
[Bugayenko2014] Yegor Bugayenko, "Continuous Integration is Dead"
[TravisPullRequests] "Travis CI. Building Pull Requests"
[OpenSource] "Open Source Initiative. Licenses by Name"
[AgileDisease] Luke Halliwell, "The Agile Disease"
[Wikipedia.StaticCodeAnalysis.Tools] "List of tools for static code analysis", Wikipedia

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« ***Unity 5 vs UE4 vs Photon vs DIY for MMO***

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture, Uncategorized
Tagged With: Agile, game, git, issue tracking, multi-player