Principles of Software Design
Implementation, Integration

Robert Lukoťka
lukotka@dcs.fmph.uniba.sk
www.dcs.fmph.uniba.sk/~lukotka

M-255

Implementation

We covered a lot of stuff that affect software construction

- Design principles
- DRY, Rule of 3
- Code smells, Refactoring
- Testing
- ...

Coding conventions

Why to have coding conventions [2]:

- 40%–80% of the lifetime cost of a piece of software goes to maintenance.[3]
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Coding conventions

Coding conventions, levels:

- Programming language level
- Organization level
- Project level
- Package level
- Single source level

There are several reasons why to break a convention, but we should make an effort to be consistent, the lower the level, the greater the importance of consistency.

- If you decide not to clean up, you should follow the conventions of surrounding code.
- Reasons for specific conventions at unit level: uses a lot of old code that uses other convention.

Coding conventions [2]

- Comment conventions
- Indent style conventions
- Line length conventions
- Naming conventions
- Programming practices
- Programming principles
- Programming style conventions

Comment conventions

- How to use Docstrings (Python), etc. Javadocs
- When are comments mandatory
- /* ...*/ vs //   (C++)
- How should comments be formated
- ...

Indent style conventions

- space or tabs (spaces are more common) or even tabs for indentation and spaces for alignment.
- how many spaces
- how to split long lines (this is a surprisingly complex topic)
    - readability
    - ease of change
- Where to put { }, there are many choices: Indentation style
- . . .

Line length conventions

79, 80, 99, 100 (Python), 180 (Mono), unlimited (Go), . . .

- Characters per line in various programming languages

79 or 80 is somewhat short (leads to more lines, may encourage shorter names) but it is standard terminal width.

Naming conventions

What needs a name needs a convention?

- variable (local/global), namespace, constant, package, class, object, method, function, procedure, ...

What should be agreed?

- length
- uppercase/lowercase
- First letter
- How to shorten long names
- How to separate words
    - CamelCase
    - snake_case
    - kebab-case

Naming conventions Java

- classes - UpperCamelCase
- methods - lowerCamelCase
- variables - lowerCamelCase
- constants - UPPER_CASE_SEPARATED_BY_UNDERSCORES

Naming conventions

There is no big reason to avoid long names.
Reasons for short names (mostly historical):

- Old linkers had limits on variable the length of names.
- Editors lack autocomplete.
- Small monitors, screens (80 characters, still a valid reason if your line width convention is like this)
- Computer science has its orgin in mathematics where short names are common.

Examples - Python

Have a very quick look at the coding style guide for Python:

- PEP 8,

Examples - C++, C

Have an even quicker look:

- Google C++ coding standards
- SEI CERT C++ Coding Standard (Example)

Coding style

Python has several guiding principles:

- PEP 20, Examples, More examples

There are many standardized way to do stuff, e.g.

- check if a list is empty: “`if list:`.”

You should learn these and use these (this fact + familiarity with common libraries is what makes you “know the language”, not syntax, syntax is easy).

Example - iterating C++ vectors

- `for(auto element: vec)`
  - You should do this.
- `for(auto it=pole.begin(); it<pole.end(); it++)`
  - You expect inserting or deleting elements of the array, or some other more complex stuff.
- `for(unsigned int i=0; i<pole.size(); i++)`
  - Variable `i` should be of significant importance.

Linters

Automated tools to check if you follow the coding conventions and the coding style (to some degree).

- pylint
- Cpplint

Use them, it is very cheap and efficient way to improve code quality.

- If an exception is required, you can mark it in the code.
- This forces you to think more whether the exception is worth it.

Dealing with errors

How to deal with erroneous inputs?

- Defensive programming - we expect each input may be incorrect.
- Design by contract - a part of the function definition we have what the inputs should satisfy and what happens in that case (e.g. undefined behavior).

Dealing with errors

Response to detected failure: Fail fast

- Fail fast and visibly.
- Makes it easier to check and correct errors

Note that fault tolerance is not opposite to fail fast. One can fail visibly and still leave large part or the whole system intact.

Test driven development

TDD workflow:

1. Add a test
2. Run all tests and see if the new test fails
3. Write the code
4. Run tests
5. Refactor code
6. Run tests

Three rules of TDD

If you do TDD, you should add your code in very small increments. These rules require you to make even smaller increments.

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

TDD advantages

Advantages

- great unit test coverage
- efficiency
- little need for debugging

TDD encourage to put minimal code into hard to test or cannot be unit reasonably tested (user interfaces, working with databases, etc.).

Disadvantages:

- Code and tests are written simultaneously, they may share blind spots.
- You need to maintain the tests, problematic for badly written tests that need to change on refactoring.

Integration

How often should you integrate.

- Frequently - integrating frequently allows to detect disagreement between teams working on various parts faster.
- To be able to obtain customer feedback, you should even be able do deliver/deploy frequently.

To illustrate necessary practices we will focus on one specific approach and quite popular approach: **continuous integration**

Continuous integration

One can extend continuous integration further.

- Continuous integration - You integrate your system with each commit.
- Continuous delivery - Current version of system is always ready for deployment (not necessarily with each commit but e.g. daily).
  - More testing needed.
- Continuous deployment - You deploy your product continuously.
  - Requires architecture supporting deployment without affecting the production too much.
  - This is architecturally significant requirement.

Continuous integration - Best Practices

- Maintain a Single Source Repository.
  - VCS has everything necessary to build and deploy the project.
  - Minimal branches, stable mainline (Reasonable branches are bug fixes of prior production releases and temporary experiments.)
- Automate the Build
  - Automate each aspect of the build
  - IDE independent

Continuous integration

- Make Your Build Self-Testing
  - All levels of testing automated
- Everyone Commits To the Mainline Every Day
- Every Commit Should Build the Mainline on an Integration Machine
- Fix Broken Builds Immediately
  - Mainline is in sound state all the time.
  - If a build breaks very often you just return to previous version immediately

Continuous integration

- Keep the Build Fast $<$ 10min.
  - Build and test only what has changed
  - Use deployment pipeline if some tests have to take long.
- Test in a Clone of the Production Environment
  - This has its limit, but you should try your best.
  - It is worth some additional expenses in purchasing the software.
- Make it Easy for Anyone to Get the Latest Executable
- Everyone can see what's happening
  - At least the state of the mainline build.
- Automate the deployment (and rollback)

Deployment pipeline

Multiple builds done in sequence

- Commit build - when somebody commits into mainline
    - Fast, reduced ability to detect bugs, but stable enough for other people to work on.
- The second stage build runs a different suite of test, may take few hours.
    - Failure shows where commit build tests should be extended.
- If necessary, there may be more stages.

Resources I

- Wikipedia - Coding conventions
- Wikipedia - Naming conventions
- M. Fowler - Continuous Integration
- optional video - B. Martin - The Three Laws of TDD
- optional video - M. Fowler - Continuous delivery

References I

- Wikipedia - Software Construction
- Wikipedia - Coding conventions
- Wikipedia - Naming conventions
- Python - PEP 8
- Python - PEP 20
- pylint
- Google C++ coding standards
- SEI CERT C++ Coding Standard
- Cpplint
- M. Fowler - Continuous Integration

References II

- Wikipedia - Continious integration