Project in Data Intensive Systems

4DV652

Lab Lecture 4

Welf Löwe

1

Agenda

- Maintenance sprint to get (back) to clean code*
- Cross validation
- Lab 4 task descriptions

*By and large following:
Robert C. Martin, Clean Code: A Handbook of Agile Software Craftmanship, Prentice Hall, 2012

2

Maintenance sprint

- Rushing to a deadline leads to shortcuts in design and development.
  - This is OK but accumulates to technical debts if not removed.
  - This, in turn, lets development slow down and hampers the onboarding of new developers.
    - Lower speed and productivity,
    - Lower development scalability
    - Lower valuation of the product/company
- Therefore, maintenance sprints should be inserted to get back to clean code again.

3

Issues to address

- Naming
- Abstraction of functions
- Comments
- Formatting
- Classes
- Error handling
- Testing
- Dependencies
- System

4

Meaningful Names

https://www.todaysoftmag.com/article/1029/clean-code-naming

- Use intention-revealing names
- Avoid disinformation
- Make meaningful distinctions
- Use pronounceable names
- Use searchable names
- Avoid encodings
- Avoid mental mapping
- Don’t be cute/funny, don’t pun
- Class names are nouns
- Method names are verbs
- Pick one word per concept
- Use solution domain names or
- Use problem domain names
- Add meaningful context but don’t add gratuitous context

5

Functions

- Small (<20 lines long)
- Blocks are usually a function call (and no nested blocks)
- Do one thing (no sections within a function, that’s another one)
- One level of abstraction per function
- Reading code from top to bottom
- Switch statements can get large, but one per function is tolerated
- Use descriptive names for functions (verbs and keywords) and arguments
- Arguments (no, one, two?, objects, lists, no flags)
- Avoid output arguments
- Command/query separation
  - No return for commands
  - No side effects for queries
- Prefer exceptions to returning error codes
- Extract try/catch blocks into own functions (error handling is one thing)
- Don’t repeat yourself (avoid cloning)
- Structured programming (Dijkstra’s rules): each function and block has one entry and one exit

6

Comments

- Comments do not make up for bad code – explain yourself in the code
- Good Comments: Legal, Informative, Explanation of Intent, Clarification, Warning of Consequences, TODO Comments, Amplification, Docs of Public APIs
- Bad Comments: Random, Redundant, Misleading, Mandated, Journal (log), Noise, Position Markers, Closing Brace, Attributions and Bylines, HTML
- Avoid commented-out code (use version control for this)
- Locality of information
- Don't use a comment when you can use a function or a variable (name)
- Not too much information

7

Formatting

- Code formatting is important for communication
- Vertical openness between concepts (e.g., methods)
- Vertical density within a concept
- Small vertical distance between related concepts: variable declaration and use, caller and callee within a class, etc.
- Vertical ordering (instance variables, public methods, private methods)
- Horizontal formatting use a common formatter implementing team rules
- For long lines, use random line breaks; indent (and align) the next line(s)
- Don't collapse blocks to the same line

8

Classes / Data Structures / Interfaces

- Use design patterns, avoid anti-patterns
- Data abstraction: interface vs implementation
- Data structure/Class anti-symmetry:
  - Classes hide their data behind abstractions and expose functions that operate on that data.
  - Data structures expose their data and have no meaningful functions.
  - Avoid hybrids
- The Law of Demeter: A method *f* of a class *C* should only call the methods of
  - *C* itself
  - An object created by *f*
  - An object passed as an argument to *f*
  - An object held in an instance variable of *C* (or local variable of *f*)
- Avoid Train Wrecks: *a.b.c.d*() => *x=a.b*; *y =x.c*; *z=y.d*() (Demeter normal form)
- Consider data transfer objects (serializable and deserializable)

9

Classes and Class Organization

- Encapsulation and separation of concerns
- Classes should be small!
- Mind the single responsibility principle
- Cohesion
  - Maintaining cohesion may results in many small classes
  - Extended reading Donald Knuth: Literate Programming
- Organize for change: private attributes, symbolic constants, …
- Isolate from change: façade pattern for interfaces that are likely to change (cf. dependencies)

10

Error Handling

- Use exceptions rather than return codes
- Write your try-catch-finally statement first
- Use Unchecked Exceptions
  - Checked Exceptions: signature of every method lists the exceptions it could pass
  - Unreadable signatures and unclear dependencies deep into library code
- Provide context with exceptions you raise
- Define exception classes in terms of a caller's needs
- Define the normal flow (and don't use exceptions for the normal flow)
- Don't return null
- Don't pass null
- Avoid error codes

11

Unit Tests

- The three laws of Test Driven Design (TDD): You may not write …
  - production code until you have written a failing unit test.
  - more of a unit test than is sufficient to fail, and not compiling is failing.
  - more production code than is sufficient to pass the currently failing test.
- Keep tests clean (readability)
- Test the non-functional requirements: scalability, efficiency, accuracy, …
- One assert per test
- Single concept per test

12

Software dependencies

- To make the software and software development processes work, they depend on this 3rd party software:
  - Language (e.g., Python 2.7 vs Python 3)
  - Services (e.g., AWS lambdas)
  - Frameworks (e.g., Django)
  - Libraries (e.g., Numpy, Scipy, Scikit-learn, Theano, TensorFlow)
  - Tools (e.g., pip, pylint) …
- Called dependencies

13

Software dependency problem

- Dependencies get further developed themselves
  - Errors and security issues get fixed,
  - provide new functionality,
  - optimize the performance etc.
- Good to adapt the own code to the latest version
- However, adaptation comes with costs
  - Potential new bugs introduced require additional testing
  - Unwanted features may make the dependency less performant
  - Incompatible APIs require new redevelopment

14

Semantic versioning

- MAJOR.MINOR.PATCH format.

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards compatible manner, and
- PATCH version when you make backwards compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

15

Dependency management

- Define an update strategy, e.g.,
  - Up to date with the latest MINOR level or
  - Not more that 3 patches behind
- Select a tool that does the work, e.g., pip, npm
- Pin your requirements, e.g., use requirements.txt, package-lock.json files
- Isolate your dependencies
  - Façade pattern
  - Virtual environments (if needed, e.g., pip)

16

Systems

- Separate constructing a system from using it
  - Separation of main (constructing the system)
  - Factories
  - Dependency Injection
  - Service oriented architecture, microservices, and registries
- Test drive the system architecture
  - Scale up – avoid Big Design Up Front (BDUF)
- Development facing documentation of (sub-)system/project level
  - How to build, test, deploy, run
  - Main entry points for new developers

17

How to organize a maintenance sprint

- Design rules agreed, documented, and communicated
- Automated formatting for simple rules that can be fixed automatically
- Automated linter for simple rules that can be detected automatically
  - Adapt configuration if needed
  - Use it and fix the warnings
- Automated testing
- Peer review for complex rules and design decisions
- Stop barriers in the CI/CD chain when any of the above fails

18

Agenda

- Maintenance sprint to get (back) to clean code
- Cross validation
- Lab 4 task descriptions

19

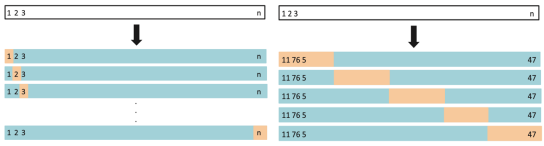
Cross-Validation

- Repeatedly
  - Separate training and validation using different splits.
  - Fit the model on the training and assess it on the validation set.
- Motivation
  - Avoid overfitting
  - Get statistical knowledge about the model
- Examples of cross-validation (CV):
  - Random split
  - Leave one out
  - k-fold

20

20

LOOCV (left) and k-fold CV



21

21

Agenda

- Maintenance sprint to get (back) to clean code
- Cross validation
- Lab 4 task descriptions

22

Lab assignment 4: Improve the Code and the Quality Assurance Process

- ML
  - Improve the regression and classification models using cross validation (lower variance)
  - Assess the test accuracy of the champion variants in scoring and classification
- Software development – Maintenance sprint improving the client/service app and the DevOps process
  - Define design and coding rules
  - Define and implement formatting, linting, dependency management, testing, peer review
  - Apply it to the code
  - Add barriers to CI/CD enforcing the above quality assurance (QA)
- Reporting in a fourth notebook:
  - CV approach and improvements in the accuracy of the models (if any)
  - Major improvements to the client/server app and the DevOps/QA process
  - Design and coding rules and how you enforce them
- Deadline: 2023-02-22

23