Implementation

Tuesday, 4 December 2018 5:00 PM

Integrated Development Environments (IDEs)

- Support all development-related work within the same tool
- Compiling, Syntax error highlighting, Debugging, Code navigation, Simulation, Code analysis, Reverse engineering design/documentation, Visual programming, Syntax assistance, Code generation, Extension

Debugging

- BAD: temporary print statements, manual tracing
- Use breakpoints in debugger
- Step forward, into, out of

Code Quality

Maximise readability

- Other dimensions: runtime efficiency, security, robustness
- Avoid long methods, deep nesting, complicated expressions, magic numbers
    - Pyramid/ arrowhead code: too much indentation
- Make the code obvious: explicit type conversion, parentheses for grouping, enums
- Structure code logically
- Do not trip up reader
    - Unused parameters, similar things that look different, multiple statements in the same line, data flow anomalies (pre-assigning values and not using them)
- KISS: keep it simple, stupid
- Avoid premature optimisations
    - Unclear where is the real bottleneck, can complicate code further
    - Let compiler optimise
    - Make it work, make it right, make it fast (unless environment is resource-constrained)
- SLAP: Single Level of Abstraction Per method
- Make the happy path prominent (unindented)
    - Use guard clauses

Follow coding standard

- Make entire code look like it was written by one person
- IDEs can help enforce coding standards
- Naming: use nouns for things (plural?), verbs for actions
    - Standard words that are not misleading
    - Explains entity to reader at a sufficient level of detail

Avoid unsafe shortcuts

- Always include default branch in case statements/ if-else (else)
- Do not recycle variables and parameters
- Never write empty catch statements, or explain when they are empty
- Delete dead code (can always be retrieved from RCS)
- Minimise variable scope: declare where it is first used
- Minimise code duplication (DRY principle)

Comment minimally but sufficiently

- Good code is its own best documentation (self-explanatory)
    - Comments are either wrong or will be wrong
    - Steve McConnell: Improve code so that comment is unnecessary
- Write to the reader

- Explain WHAT (supposed to achieve) and WHY (rationale for current implementation, e.g. To follow some format), not HOW

Refactoring

- Improving a program's internal structure in small steps without modifying its external behaviour
    - Not rewriting (has to be small steps)
    - Not bug fixing (changes external behaviour)
    - Hidden bugs easier to spot, may improve performance (easier for compiler to optimise)
    - Benefit of refactoring should justify cost
- Regression Testing (check for unintended side effects)
- Consolidate Duplicate Conditional Fragments
    - Code appears in all branches -> move outside of conditional
- Extract vs. Inline method (opposites)
- Define symbolic constants (no magic numbers)
- Guard clauses: reduces nesting
    - Main part should not be in an indented "if" block
    - `if (errorCondition) { // handle error }`
- Decomposes Conditional: every branch calls a method
- Remove flag argument, directly call new method
- Renaming files

Documentation

- Developer documentation-as-user: API, tutorial-style
    - Documents how software should be used
    - Javadoc: generate API in HTML from comments
- Developer documentation-as-maintainer: technical details are important
    - Note that these readers have access to the source code
- Tools: Javadoc, Markdown (lightweight markup language), AsciiDoc (similar to MarkDown)

Guidelines

- Top-down documentation (breadth-first)
    - User can travel down path they are interested in
- Aim for comprehensibility on top of accuracy and being comprehensive
    - Use diagrams (e.g. UML), examples, simple and direct explanations
    - Diagrams for completeness and not understanding should be in appendix
- Document minimally but sufficiently (just enough)
    - Overhead of maintaining documentation
    - Provide higher-level information not readily visible in code or comments

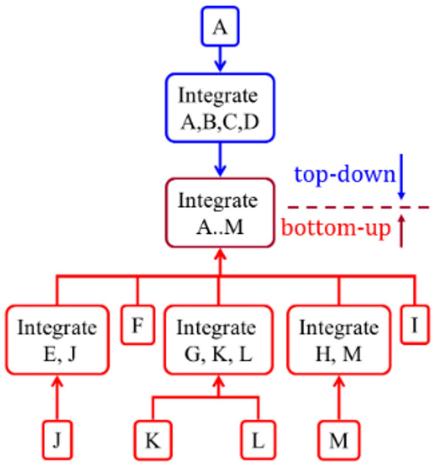
Error Handling

- Exceptions: unusual but not unexpected situations (disrupts normal flow of program instructions)
    - Throwing exception: create object and pass to runtime system
    - Runtime system finds exception handler in call stack (to catch exception)
    - Propagating error information to code that can handle it
- Assertions: assumptions about program state
    - Make assumptions explicit and confirm them
    - Invariants in internals, control flow and class, pre/ post conditions
    - Can be disabled without modifying code (-enableassertions/ -disableassertions)
    - Disabled by default
    - JUnit assertions: more powerful, customised for testing
        - Java assertions more used in functional code
    - Low impact on performance, and provides additional safety
- Exception = unusual user behaviour, assertion = programmer mistake

- Logging: recording information, usually to a log file
    - Useful for troubleshooting, should be meaningful (has a cost)
    - Record system information regularly (e.g. Black box - find out what happened to cause error)
    - `import java.util.logging.*`
        - `Logger.log(Level.INFO, "message")`
    - Logging intensity
- Defensive Programming
    - "Anything that can go wrong will go wrong", identify holes and actively plug them
    - Return copies of objects
    - Enforce compulsory associations
        - Non-null constructor param)
    - Enforce 1-1 associations
        - Call one constructor in the other, so both are simultaneously created
    - Enforce referential integrity in bi-directional associations
        - If A sets B, B must also set A
    - How critical is the system? Will code be used by other authors? Support for defensive programming by language? Overhead?
    - What if everyone agrees not to misuse?
- Design-by-Contract (DbC)
    - Define formal, precise and verifiable interface specifications for components
    - Responsibility of caller to ensure preconditions are met
        - Honour contract only if preconditions met, else unspecified
    - Supported by Eiffel (not Java or C++) - Bertrand Meyer

Integration

- Combining software components
- Late and One Time vs. Early and Frequent (timing and frequency)
    - Better to have a walking skeleton (e.g. By one developer)
- Big-Bang vs Incremental
    - Slowly integrate components together
- In what order to integrate code incrementally? Top-Down vs Bottom-Up
    - Top-Down: discover higher-level problems early, but requires stubs
    - Bottom-Up: drivers needed to test integrated components
    - Sandwich



Build Automation

- Build scripts to automate building (usually automated by IDEs)
- Java tools: Gradle, Maven, Apache Ant, GNU Make
    - Dependency management (3p libraries), e.g. Maven and Gradle
- Continuous Integration (CI) vs Continuous Deployment (CD)
    - CI: integration, building, testing every PR/ merge

    - CD: deployed to end-users
    - Travis, Jenkins, Appveyor (windows + unix), CircleCI

Reuse

- Reuse tried-and-tested components
- Libraries and frameworks come at a cost: overkill, immature/ unstable, dying off, license issues, bugs and security vulnerabilities that can sneak into your product
- Application Programming Interface (API): contract between component and clients
    - Should be well designed and documented
    - Interaction between components can be shown in sequence diagrams
- Libraries (e.g. Java JDK classes, Natty for dates)
    - Read documentation, check license, download library (or use dependency management tool)
- Frameworks: reusable implementation of software or part of software with generic functionality (can be customised for specific app)
    - Facilitates adaptation and customisation of desired functionality
        - E.g. Via plugins
    - Can be fully functional, e.g. Eclipse IDE
    - Component-specific (e.g. JavaFX, TkInter)
    - Web: Drupal(PHP), Django(Python), Ruby on Rails (Ruby), Spring (Java)
    - Testing: JUnit (Java), unittest (Python), Jest (Java Script)
- Framework vs Library
    - Libraries = as-is, frameworks = customised/ extended
    - Library code is called, framework calls your code (inversion of control, Hollywood principle)
- Platforms: runtime environment for apps (e.g. OS)
    - Usually bundled with libraries, tools, frameworks and technologies
    - JaveEE, .NET: used for enterprise apps (also frameworks)
        - JaveEE uses JVM, .NET uses Common Language Runtime (CLR)
- Cloud Computing: service delivered over network, centralised server farm
    - IaaS (infrastructure): users can deploy virtual servers and use cloud storage
    - PaaS (platform): platform for developers to build apps
    - SaaS (software): apps accessed over network