OOAD & Software Engineering (UE18CS353)
Unit 3

Aronya Baksy

March 2021

1 Design Principles

1.1 GRASP

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**rinciples (or **P**atterns) is a set of principles that help in assigning responsibilities to objects.
- Responsibility: a contract/obligation to be fulfilled by a component/class/module

1.1.1 Creator

- Assign B the responsibility of creating object A if:
  - Instances of B contain or aggregate instances of A
  - Instances of B record instances of A
  - Instances of B closely use instances of A
  - Instances of$B$ have the initializing information for instances of A and pass it on creation

1.1.2 Information Expert

- Assign those responsibilities to a class which has the information to fulfill that responsibility
- In case one class does not have all the information, assign the responsibility to the class that has the most information needed to fulfill it.

1.1.3 Low Coupling

- Coupling is a measure of how inter-dependent objects of 2 different classes are.
- Low coupling is always preferred as it minimizes the impact of changes in one class on the dependent classes.
- Low coupling makes a system maintainable, efficient and code reusable
- Two elements are coupled, if :
  1. One element has aggregation/composition association with another element.
  2. One element implements/extends other element

1.1.4 Controller

- Minimizing dependency between UI classes and system classes that represent system-level operations.
- The controller class receives requests from UI layer objects and then controls/coordinates with objects of the domain layer to fulfill the request.
- The controller can be reused, can maintain the state of the use case and can control the sequence of the activities

1.1.5 High Cohesion

- Merge related responsibilities into single modules. Clearly defining the purpose of a single manageable unit.
- High cohesion allows for more code reuse, more understandability and maintainability.

1.1.6 Polymorphism

- Responsibility for defining the variation of behaviors *based on type* is assigned to the type for which this variation happens
- Handle related but varying elements based on their type.

1.1.7 Pure Fabrication

- A pure fabrication is a class that does not represent any real-world entity, but completes an associated set of responsibilities.
- Decomposes the behaviour of a complex class, and thus increases cohesion, reduces coupling and increases reusability.
- Implemented in Adapter and Strategy design patterns
- e.g.: A class called DBStore does all database operations and takes that responsibility away from a "Shape" class that does all Shape-related computation
- e.g.: logInterface which is responsible for logging information

1.1.8 Indirection

- Avoid direct coupling between 2 classes by introducing an intermediate unit to communicate between the other units, so that the other units are not directly coupled
- Implemented in Adapter and Façade design patterns

1.1.9 Protected Variations

- How to avoid impact of variations of some elements on the other elements
- Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them
- Implementation: Provide an interface so that the there will be no effect on other units and then use polymorphism to create various implementations of this interface
- Provides flexibility and more structured design

1.2 SOLID

- Just like GRASP, another set of patterns for implementing best practices/low coupling/high cohesion etc
- Enables changes, makes code more flexible, maintainable, stable, and reusable

1.2.1 Single Responsiblity Principle

- A class should have only one responsibility
- This leads to only one reason for changing that class, hence reducing side effects during maintenance
- The aim of splitting responsibilites is to increase cohesion, minimize coupling.

1.2.2 Open-Closed Principle

- A class should be open for extension but closed for modification. Also stated as: "One should be able to extend the class behavior without modifying it"
- Implemented using inheritance and composition. In inheritance, the base class behaviour does not need to be touched in order to extend its behaviour using a subclass

1.2.3 Liskov Substitution Principle

- A derived class must be substitutable for its base class
- Formally, let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

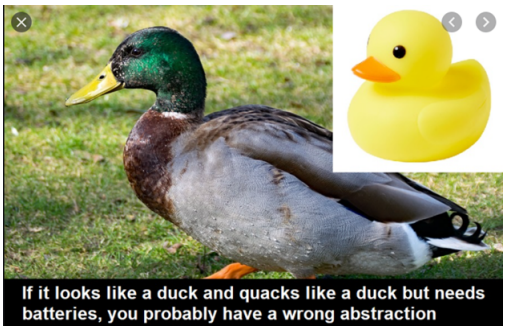

Figure 1: Meme about Liskov Substitution

1.2.4 Interface Segregation Principle

- Clients should not be forced to depend upon the interfaces that they do not use.
- Make fine-grained interfaces that are specific to each client.
- Instead of having one interface class that can handle N special cases, it is better to have N interfaces, one for each special case.

1.2.5 Dependency Inversion Principle

- Program to an interface, not to an implementation
- Formally, it is stated as:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
  - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

2 Software Implementation Phase

- The detailed creation of working software through a combination of coding, reviews and unit testing.
- **Entry criteria** is design (low-level and high-level, detailed design document) and **exit criteria** is unit-tested and peer-reviewed, functioning code.
- Involves choice of language, choice of dev env, building and following a config management plan and actual development of code.
- Primary goals for implementation phase:
  - Minimize complexity

  - Anticipate change
  - Construct readable, testable and reusable code

- Characteristics of implementation phase:
  - Tool dependent: Compiler, debugger, IDE, GUI builders etc.
  - Large volume of config items: source files, test case files, documentation etc.
  - Directly affects software quality
  - Use of CS knowledge: DSA, coding practices etc.

2.1 Primary Considerations

- **Handling errors**: using exception handling constructions, checking return values, and using assertion statements
- **Security breaches**: consider every opportunity for a security breach in the code, e.g. buffer overflow, SQL injection
- **Effective use of computation power** on host system, considering other programs also running on that system
- **Documentation** about the code, and within the code (combined with machine-readable source code)
- **Optmization**: Use performance analysis tools to identify slow-running sections of code, optimize those to run faster

2.2 Choice of programming language

- Assembly languages: operate at the machine level directly (e.g.: ARM assembly, x86_64 so on)
- Procedural languages: modest level of abstraction above hardware (e.g. early versions of Fortran/ C/ Pascal)
- Aspect-oriented languages: Allow more separation of concerns and increase modularity (e.g. basically any language you can think of)
- Object Oriented: Highest level of abstraction, developers now think of the problem in terms of real-life objects and their behaviour (e.g. C++, Java, C#, Python, etc.)

2.3 Choice of Development Environment

- Commercial vs Open Source: Developers prefer open source tools (customizable) but enterprises feel that commercial tools are more fully featured
- Support of process: Availability of extensions that support debugging, static analysis, test building etc.

2.4 Coding Bugs

- Bugs are caught at review or at unit/integration/system testing, acceptance testing or at production
- The cost of catching a bug at each subsequent stage increases exponentially. Hence catching bugs ASAP is crucial.

2.5 Characteristics of Code

- **Well structured, well designed**. Not too many lines per function, lines per file, args per function, levels of nesting and conditions
- **Meaningful, descriptive but short variable names** that follow consistent naming (Pascal case, camel case or underscore prefix)
- Restrictions enforced by programming **language syntax** (keywords, identifier length etc.)
- Highlight dependencies in software through **file organization**. Logically grouped and partitioned files, properly initialized data structures (Abstract Data Types or ADTs)
- **Easy to read** code with indentation, brackets (when applicable), spaces, newlines, parenthesis for grouping,
- **Well commented** code. Comment should be brief and concise but not cryptic, misleading or redundant.

2.6 Refactoring

- Improve internal structure of code without changing external behaviour
- Improve non-functional and objective attributes of the code (cyclomatic complexity, length, duplication, coupling/cohesion) that improve understandability and maintenance.
- Does not involve rewriting code, fixing bugs or changing observable interfaces.

2.7 Coding Standards and Guidelines

- Standards are mandatory rules, guidelines are just recommendations. Both aim to increase overall code quality.
- Give an uniform appearance to code that has been written by different people.
- Improve readability, maintainability of code by removing effects of different personalities that work on code

2.7.1 Coding Standards

- Promote code reuse, sound practices, and increase efficiency.
- **Defensive Programming** is a standard that involves checking for all implicit assumptions and handling each case. Involves adding redundant code that checks system state after each modification
- **Secure Programming** is the practice of developing computer software in a way that guards against the accidental introduction of security vulnerabilities. Practices involved in secure programming are:
  - Validate input from untrusted source
  - Heed compiler warnings and eliminate them using analysis tools
  - Default deny all permissions to all modules, unless a permission is absolutely essential for a particular module to function properly.
  - Principle of least privilege. Each process gets the most minimal set of privileges for the most minimal amount of time that is needed to finish it's work and nothing more
  - Sanitize data sent between external components and your code.
- **Programming for Testability** involves the use of the following:
  - **Assertions** identify out-of-range and inappropriate values
  - **Test Points** are functions that set and get current module status for debugging
  - **Scaffolding** emulates the functionality of not-yet-built features for message exchange

- **Test harness** is code written to drive incomplete objects/modules as if they were complete
- **Test Stub** is a function that returns a fixed value as it has not yet been implemented
- **Instrumentation**, aka logging
- **Building test dataset**

2.7.2 Coding Guidelines

- Generic suggestions regarding the coding style for betterment of understanding, readability of the code
- Helps in early error detection that reduces costs
- Simplifies maintenance

2.8 Evaluating code quality

2.8.1 Evaluation Metrics

- Code construction is evaluated from 2 perspectives: progress (in development according to the schedule in the project plan) and quality
- Progress and productivity are **measured** in terms of active days spent, scope completed, code churn etc. **Metrics** are LoC generated, LoC purged, LoC per effort day
- Code quality **measured** in terms of static code analysis results on length, insturction paths, complexity. **Metric** is number of errors per KLoC

2.8.2 Special Evaluation Metrics for Agile projects

- **Sprint Breakdown** is a chart wherein time is plotted on the x axis and the work remaining (in terms of number of user stories) is plotted on the y axis. The expected and actual progress can be measured and compared using such a chart
- **Team Velocity** measures the number of stories the team completes per sprint. It can be measured in story points or hours, and can be used for estimation and planning.
- **Throughput** is the total value added work per unit time. Measured as number of tickets (i.e. stories) completed per unit time.
- **Cycle Time** is the time elapsed between a team starting work on an item and completing that work.

2.8.3 Ensuring code quality: Review

- **Peer Review**: informal advice given by other developers about the code
- **Unit Test**: write code that calls functions and checks their correct functionality
- **Test-first**: First build test harness, then write code. Ensures that developers don't cheat by writing tests that pass on their code only
- **Code Stepping**: Execute code line by line and check system state (variable values etc.) at each time to check control flow.
- **Pair Programming**: Two developers, one focusing on function logic, another one on syntax and accuracy
- **Code Inspection**: formal review of code in front of a review board that consists of code inspectors

2.8.4 Code Inspection

- Build checklist:
  - Based on common errors, past experience
  - Checklists are language dependent and reflect the most common errors that are likely to occur in that language
  - In general, the 'weaker' the type checking, the larger the checklist.
  - e.g.: Initialization, Constant naming, loop termination, array bounds, etc.
- Planning: Select team, venue. Make documentation and code available to inspection team in advance.
- Inspection takes place, errors are noted, modifications are made if necessary to repair those errors.
- Re-inspection may or may not be required post the changes
- All of these are documented and archived

2.8.5 Unit Testing Tools

- **Unit-Testing Frameworks**: Enter method to be tested, arguments, and expected output. Framework automates the method call, test and reporting process. (e.g.: Nunit for .NET apps, Junit for Java apps)
- **Coverage Analyzers and Debuggers**: Find percentage of code that is covered by unit tests, and locate and fix errors (e.g.: JaCoCo available with Eclipse for Java apps, Cobertura, PureCoverage)
- **Wizards**: automated test generation given input parameters
- **Record-Playback Tools**: Start a session, record a sequence of mouse and keyboard activities and replay those later quickly and accurately (e.g.: Selenium)

3 Software Configuration Management (SCM)

- A process for systematic organization, management and control of entities generated during each phase of the SE process for the product under consideration
- Increase productivity, increase and plan coordination among the programmers in a team and eliminate confusion
- In a Scrum team, SCM is the responsibility of the entire team. Max amount of automation is used for SCM process.
- In a Scrum team, evelopers pull from central repo. Make changes in their local env, perform unit tests, once the tests all pass then push changes to central repo.

3.1 Benefits of SCM

- Permit orderly development, as well as orderly implementation and release of software items.
- Only approved changes (ones that conform to specification) are deployed. No unauthorized changes are added to the central repo.
- Keeps documentation updated to reflect changes. Communicate changes and their impact to all stakeholderss

3.2 Roles in SCM

- **Configuration Manager**: Identify config items, identify procedures for promotion and release
- **CCB Member**: Approve/reject change requests
- **Developer**: Implement versions based on project plan or change request. Check changes, resolve conflict
- **Auditor**: review the selection process for promotions for release. Ensure completeness and consistency of release

3.3 Config Management Plan

- This is the outcome of the SCM Planning phase. Done by the config manager in a full time or part time capacity.
- Can be made company-specific or according to some template like IEEE 828
- SCM Plan normally contains the following:
  - Config Items (CIs) to be managed, and naming scheme for the same
  - Who is responsible for config management and baseline creation
  - Change control and version control policies
  - Tools to assist in config management. Limitations of those tools
  - Config management database used to record info during the SCM process

3.4 Activities in SCM

3.4.1 CI Identification

- A config item (CI) is an entity (software or hardware or aggregation of the two) that is designated for SCM
- e.g.: code files, test drivers, documentation, manuals, software/hardware config,
- Selection of CI similar to class modelling. Identify entites and their relationships.
- Starting SCM too early leads to bureaucratic complications. Starting SCM too late leads to chaos.

3.4.2 CM Directories

- The **programmer's directory** is controlled by the developer only. It contains newly created or modified CIs only
- The **master directory** is an entry-controlled, authorized directory used for managing current baselines and controlling changes made to those baselines.
- The **software repository** is an archive for the various baselines released to the public. These are distributed to requesting parties

3.4.3 Baseline

- A specification or product that has been formally reviewed and agreed to by responsible management, that serves as the basis for further development, and can only be changed through formal change control procedures
- Baselines may or may not be tied to the schedule defined in the project plan.

3.4.4 Branch Management

- A branch is a copy of the source code (fully or a part) within the repository. Branching is done for the following reasons:
  - Support concurrent development
  - Support multiple versions of a solution
  - Enables experimentation by developers without affecting the central codebase
- Merging is bringing back and integrating the changes done in branches to the working branch. Frequent merging from related branches into the working branch helps decreasing the likelihood and complexity of a merge conflict
- Branch strategies:
  - Single working branch
  - Branch by developer or workspace
  - Branch by customer or organization
  - Branch by module or component

3.4.5 Version Management

- Tracking different versions of components used in a software project
- Typically tools like Git are used for this purpose.
- Features of version control:
  - All versions are identified by a unique version number
  - All changes are traceable
  - Change history is recorded to allow rollback of changes if needed
  - Conflict resolution, maintenance and quality monitoring.
  - Less software regression

3.4.6 Build Management

- Compiling all source code files and libraries, and linking them together to build an executable program or a binary.
- Build utilities compile the files in the correct logical order (in case there are no changes to a file it may not be compiled again) and then links the generated binaries together
- Build automation tools like Maven, make (the GNU standard tool), Apache Ant
- Build Process:
  1. Fetch code from source control repo
  2. Compile the code. Check dependencies and then link the libraries and code appropriately
  3. Run unit tests (manual or automated)
  4. Once all tests pass, build the artefact and store them. Archive the generated build logs.
  5. Notify all stakeholders, then update the version number
- A build can either be a **full build** (from scratch, all files) or an **incremental build** (only changed files are recompiled and linked)
- Build process can be triggered by:
  - Manual trigger
  - Scheduled trigger
  - Repository trigger
  - Post process build trigger

3.4.7 Install Management

- Involves placing multiple executable files, downloading or copying from a repository.
- Could also involve downloading executable files, config files, libraries, images etc. from the internet
- Also involves interaction with the OS when requesting for resources to run the software, and managing appropriate permissions.
- Automated install management is performed by tools like InstallShield, InstallAware, Jenkins etc.

3.4.8 Change Management

- General process of change management:
  - Change is requested. This change request is logged into a tool along with some metadata about the change (change ID, date, who raised, etc.)
  - The cost and overall development impact of the change is assessed. The result of this assessment is either the change request being accepted or rejected
  - If accepted then the change is implemented, validated and audited
  - The documentation is updated. Plans made for versioning, merging, delivery are followed.
- Smaller projects have simple and informal change management. Larger projects have a change control board (CCB) that documents all changes and accepts/rejects requests
- A change is described by its description, reason for making the change and other items affected by this change.
- Change management is highly automated and tool driven. Change logging tools like Jira, file comparision tools for tracking changes, etc.
- Change control policies:
  - Change policies guarantee that each version, revision or release conforms to commonly accepted criteria and ensures consistent process throughout.
  - These change policies are enforced through engineering processes and tools. They are also audited to ensure they conform to the processes defined

3.4.9 Promotion Management

- Policies that control the pushing of code from programmer directory to the master directory.
- Policies are based on baselining criteria, and include validation, followed by a process of verification and authorization of the changes made.
- e.g. for promotion policy: "No developer is allowed to promote source code which cannot be compiled without errors and warnings"

3.4.10 Release Management

- Policies that control movement of code from master directory to the software repository, and its release to the final customer.
- The policy is based on:
  - Quality criteria, verification of metrics for the same
  - If quality criteria are met, release is authorized and code is archived to the software repository
- e.g. for release policy: "No baseline can be released without having been beta-tested by at least 500 external persons"
- A **patch** is a set of updates made to a code to improve, update or fix any issues in it.
- A service pack or SP or a feature pack (FP) comprises a collection of updates, fixes, or enhancements to a software program delivered in the form of a single package.
- Patches can be generic, hot patches, source code/binary patches, or emergency patches.

3.4.11 Defect Management

- Defect is a variation between the actual behaviour of the software product and the expected business requirements.
- Bugs are tracked and fixed using automated tools like BugZilla
- Steps in defect management:
    1. Discover bug. Report it and log it into a tool with an unique ID
    2. Validation, analysis, priority (critical, high, med, low)
    3. Open formal request and get approval for the same.
    4. Bug resolution (assignment, schedule, fix, test, report)
    5. Verification by submitter of bug report. Once verified merge code
    6. Update version number, plan release of the bug fix.
    7. Close the process, generate report.