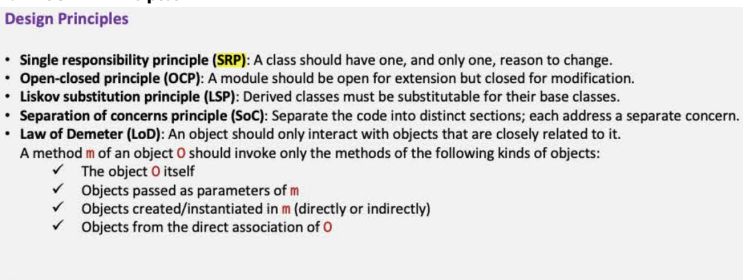
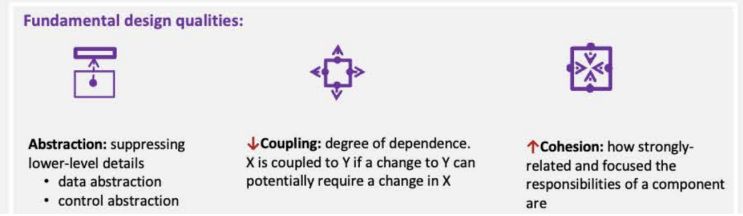
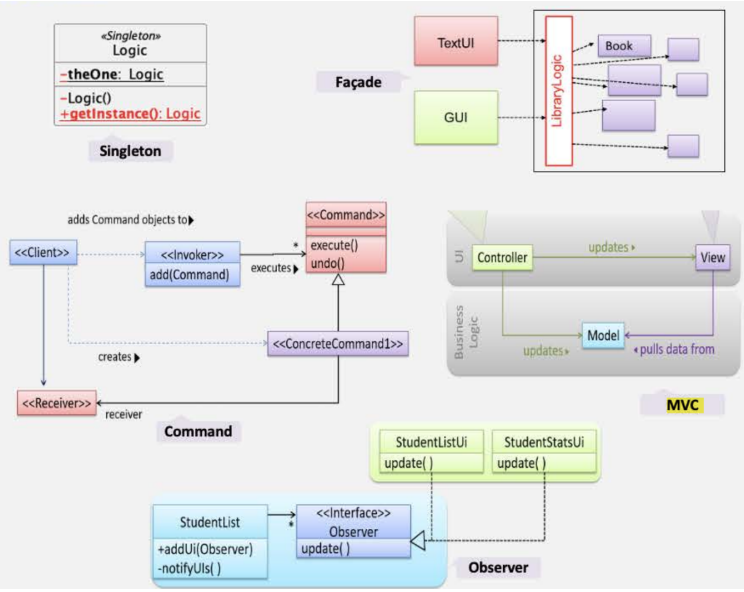
1 Image Notes

1.1 SOLID Principles





1.2 Design Patterns



2 Quiz Mistake List

2.1 Principles

- SRP ≠ 1 method 1 responsiblity. Violation E.g. 1 class adding to DB, sending email, and logging.
- OCP - violated when change in behavior requires modifying an existing class instead of extending it. Violation E.g. a lot of if-else statements to check for type before executing corresponding code.
- Liskov Substitution Principle (LSP) - a subclass must work in place of the super class. No unexpected exceptions, pre/postcondtions, invariants, etc.
- Separation of Concerns (SoC) - dont mix logic. Violation E.g. `AB.add(person); sout("Added person");`
- Law of Demeter (LoD): usually violated with train wrecks. Violation E.g. `b.getSomething().getMoreStuff().doSomething();`
- **SLAP**: Single Level of Abstraction Principle. Violation when many diff levels of indentation.

2.2 Design Patterns

- **MVC**: View: Displays data, interacts with the user, and pulls data from the model if necessary. | Controller: Detects UI events such as mouse clicks and button pushes, and takes follow up action. Updates/changes the model/view when necessary. | Model: Stores and maintains data. Updates the view if necessary.
- **Singleton Pattern**: can reduce testability.
- **Facade Pattern** increases the amount of code.

2.3 Java

- Constructors of superclasses can be called from subclass. But since constructors aren't members, they are **not** inherited.
- Polymorphism **allows** different behaviour if you feed subclasses of the argument class.
- Every object has an interface and implementation
- Packaging is an aspect of **encapsulation**
- Once a class has 1 or more abstract methods, it is an abstract class.
- Unchecked exceptions are not subject to Catch or Specify statements
- 2 Classes of Exceptions: Checked & Unchecked [Runtime exception, Error].
- **JavaDocs** comments can be (1) documentation for developer-as-user and for developer-as-maintainer.

2.4 Class Diagrams

- \+ & - are for visibility (not accessibility). [+] public [#] protected [~] package-private(default) [-] private
- **class-level notation**: underline in class diagram
- **inheritence** Foo → Bar: Foo inherits from bar / Foo extends Bar. Arrow points to super.
- Roles should appear at the end of the class that plays the role.
- Assosciation Role ≠ Association Label
- Bot[5] = 0..5
- Class should only appear once in a diagram.
- **Association Class**: If dotted line, then don't show variables making it up and vice versa.
- **Association**: can be shown as attributes or association lines **but not both**.
- Mandatory: Class Name, Associations, Multiplicities,
- Possibly Mandatory: Class-level members must be underlined, Attributes(if relevant), Methods(if relevant)
- Optional: Visibility, Attribute/Return/Parameter Types, Navigability Arrows
- **Composition** ♦ (Whole♦–Part, **solid** diamond) Implies (1) delete cascading (2) no cyclical links
- **Aggregation** ◊ (Container◊–Item, **Hollow** diamond)
- Observer –> Subject. Arrow points from observer to Subject - Observer depends on Subject.

2.5 Object Diagrams

- Labels must be underlined.
- Navigability of associations (arrows in associations) in Object Diagrams are optional. Either direction works, just keep the association role consistent.

2.6 Activity Diagrams

- Optional: Activation Bars | Return Arrows | Object creation/deletion Markers | Message labels (names, parameters, return values)
- Mandatory: Lifelines | Messages (solid arrows) | Top to Bottom
- Object Names should not be underlined
- Activation bar of method should not be broken in the middle (when it calls another method)
- Incoming/Return arrows must be at the very top/bottom

2.7 Conceptual Class Diagrams

- Should be used in place of CDs for **domain modelling**.
- **CCD**: Conceptual Class Diagrams (lighter version of class diagrams) capture class structures in the problem domain, AKA OO Domain Models (OODMs).
- **CCDs** do not show methods or navigability.

2.8 Sequence Diagrams

- Loop segments can happen 0 times.

2.9 Project Stuff: Requirements & Code Requirements

- **Brownfield vs Greenfield**: Brownfield harder to modify, Greenfield harder to design. Overall brownfield slightly harder.
- **User stories** are considered light-weight. Recc: Limit to 1 sentence.
- **Magic literals** Try to avoid, but loop counters or simple indexing is fine (not considered magic).
- **KISS**: Keep it simple, stupid - Clever solution only if the additional complexity is justifyable.
- Bug Fixing is not refactoring
- **Asserts**: Java assert ≠ JUnit(or other) assert. Java asserts are disabled by default at runtime, and are mainly used for internal consistency checks.
- **Abstract -> Interface**: In Java, abstract classes don't need to implement interface methods. Only concrete classes do.

2.10 Logging

- Logs are typically written to a log file but it is also possible to log information in other ways e.g. into a database or a remote server.
- Java has its own default logging mechanism

2.11 Use cases, More Project Stuff & Abstraction

- **Use cases** have no strict requirement to have unique IDs, but might be helpful.
- **Use case benefits**: Help uncover functional requirements, provide common understanding between stakeholders, Guide system design and implementation, Support testing
- Textbook recommends to **mix the bottom-up and top-down** approaches
- **Data abstraction**: abstracting away the lower level data items and thinking in terms of bigger entities.
- Types of **abstraction**: Procedural, Data, Control
- **Coupling Cases**: A has access to B's internal structure, A and B depend on the same global variable A receives an object of B as a parameter or return value, A inherits from B.
- **Defensive** programming can result in slower code.

2.12 Testing

2.12.1 Equivalence Partitioning (EP)

- Rememember that in Java, `isPrime(int i);` non-ints aren't valid, so "all nonints" is **not** a valid EP.
- EPs cannot give a **Neumann-complete** test suite.
- EP for **Day of Month**: [-MAX..0][1..28][29][29][30][31][32..MAX]
- (1) both invalid ends (2) 1-28 in all (3) 29 only leap feb (4) 30 OR 31 days in a month

2.12.2 Types

- unit - 1 item
- integration - work together
- regression - retesting to see if regress
- system - whole system meets specification?
- acceptance - done by end-users to see if meets req.
- **Alpha Testing**: performed by the users, under controlled conditions set by the software development team.
- **Beta Testing**: performed by a selected subset of target users in their natural work setting.

2.12.3 Approaches

- **Scripted testing**: perform testing based on a pre-defined set of test cases.
- **Exploratory testing**: design test cases on-the-fly.
- **Test-Driven Development (TDD)**: evolve functionality and tests in small increments, writing the test before the functional code.

2.12.4 Test Coverage

- **Function/method coverage**: based on functions executed, e.g., testing 90 out of 100 functions.
- **Statement coverage**: based on the number of lines of code executed.
- **Decision/branch coverage**: based on decision points exercised, e.g., an `if` statement evaluated to both true and false in separate test cases.
- **Condition coverage**: based on boolean sub-expressions, each evaluated to both true and false with different test cases.
- **Path coverage**: measures coverage based on possible execution paths through the code.
- **Entry/exit coverage**: measures coverage based on possible calls to and exits from operations in the SUT.

2.12.5 Other Notes

- **Dogfooding** - Creators using their own product to experience how end users experience it.
- **Unit Testing** isn't just about stubbing, **Integration Testing** the same as unit testing without stubs.
- **Coverage Analysis** can be useful in improving the quality of testing (not individual test cases).
- **SUT with multiple inputs**: testing all combinations is effective but possible inefficient.
- **___ Box Testing** - Knowledge of code: Black box < Grey box < White/Glass/Clear box
- **Path coverage** is much harder to achieve than statement coverage. PC is all possible paths, SC is just code lines.
- **Path coverage** & Neuman-complete PC - covering all paths is impossile irl since there are loops that may have infinite paths. in real life, they check: 0, 1, many, and boundaries. IRL tools like Jacoco count branch & statement coverage but do not attempt path.

2.12.6 Efficiency vs Effectiveness

Efficiency - How well a system or process achieves its intended goal (doing the right thing)

Effectiveness - How well a system or process uses resources while achieving its goal (Doing things well).

2.13 Architecture

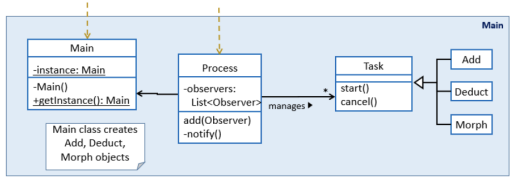
- Canvas uses Client-server Architecture style. Client, Server, Database, Communication(Requests)

2.14 Random stuff

- When developing a software to compete with Facebook(or im guessing other large large software), an iterative approach is more suitable than a sequential approach.

2.15 Straight up copy paste

2.15.1 Design Patterns in Main

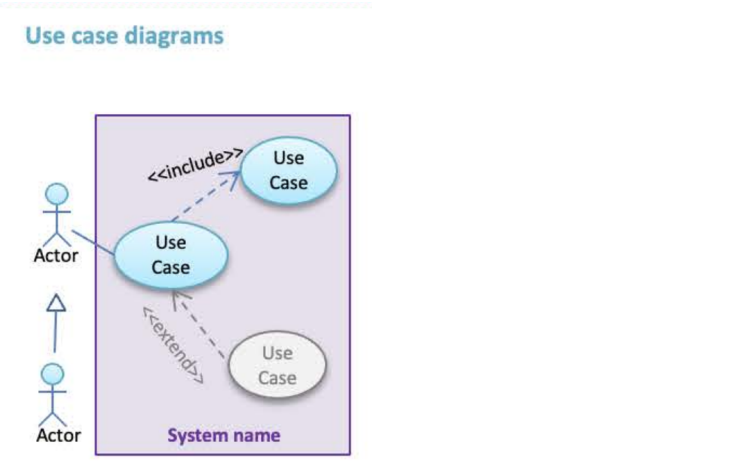


main uses Singleton, Command, Observer.

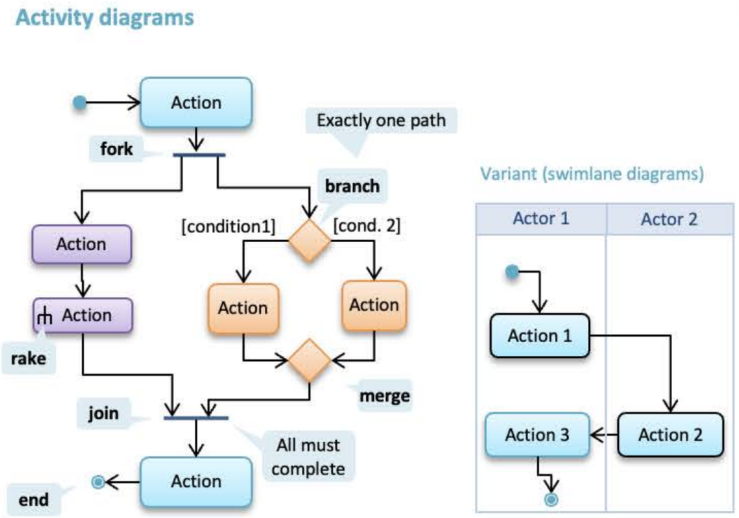
- Multiple classes are exposed to the outside, which means it is unlikely the Facade pattern is used.

3 Diagram Examples

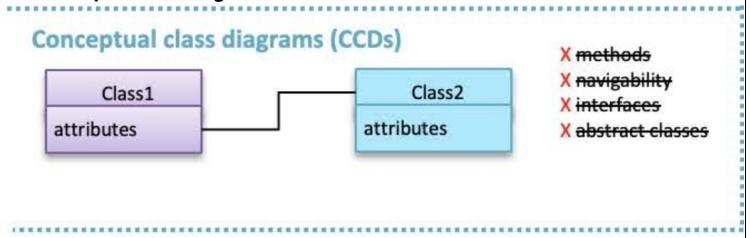
3.1 Use Case Diagram



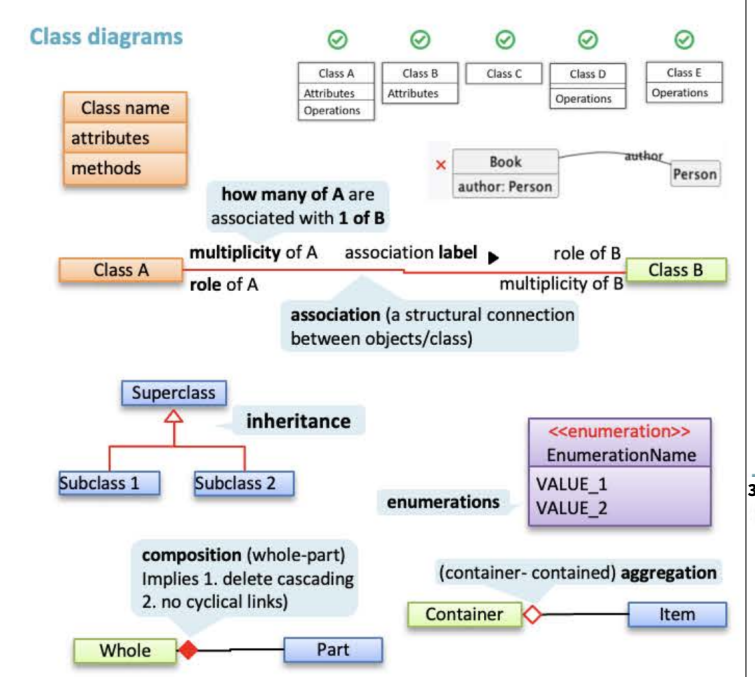
3.2 Activity Diagram



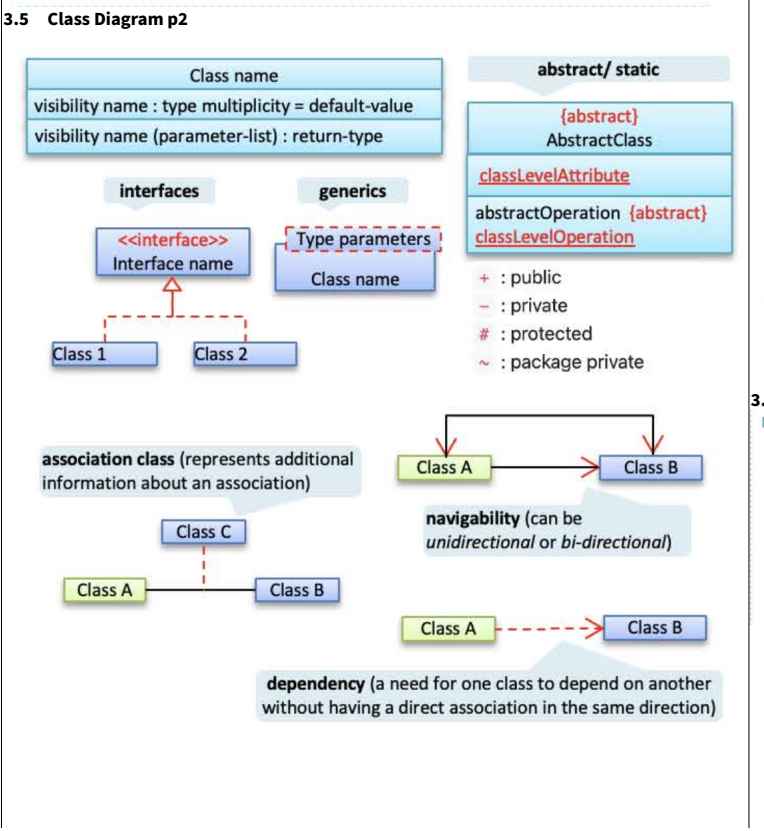
3.3 Conceptual Class Diagram



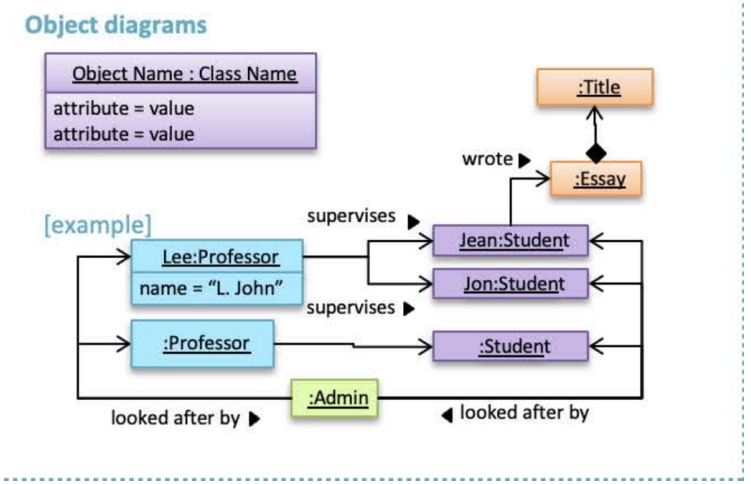
3.4 Class Diagram p1



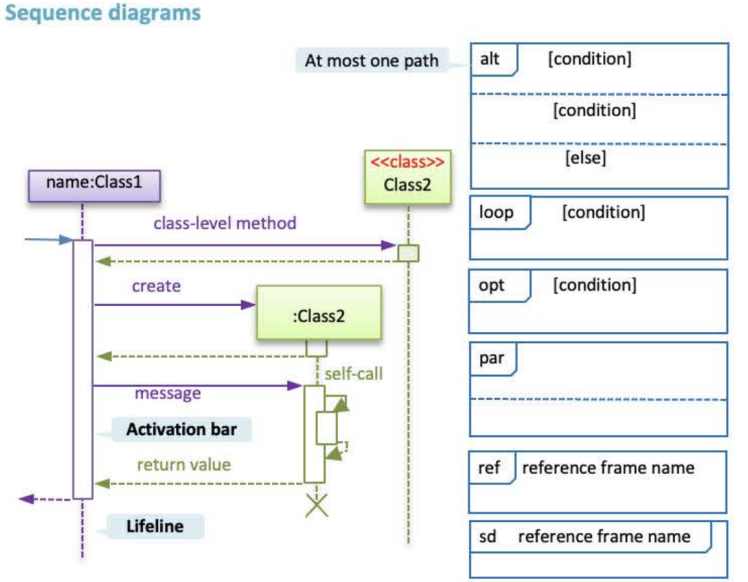
3.5 Class Diagram p2



3.6 Object Diagram



3.7 Sequence Diagram



3.8 Notes