CS2103T Notes

tags: `CS2103T`

Git Commands

```
# Add tag "some-tag" to current commit
git tag some-tag
# Delete tag "some-tag" from current commit
git tag -d some-tag
# Push tags to remote
git push --tags

# Merge "some-branch" to "master" without fastforward
git checkout master
git merge --no-ff some-branch

# Undo local commit till "commit-hash"
git reset --hard commit-hash

# To checkout a branch "some-branch" that exists on the remote but not locally
git fetch
git checkout some-branch

# List all branches, including remote (if fetched)
git branch -a
```

Gradle Commands

> Assumes commands are run at the project root

```
# List out all possible commands
./gradlew tasks

# Compile and run the program
./gradlew run

# Compile and test the program
./gradlew test
```

Week 1

- **Programming paradigm**: guides programmers to analyse programming problems and structure programming solutions in a specific way
- **Object-Oriented Programming (OOP)**: is a programming paradigm
  - Views the world as a network of interacting objects

    - Tries to create a similar object network inside the computer's memory so that a similar result can be achieved programmatically
    - Does not demand that the virtual world object network follow the real world exactly
- **Objects**: has both state (data) and behavior (operations on data)
    - Has an interface and an implementation
    - Interact by sending messages
    - Is an abstraction mechanism because it allows us to abstract away the lower level details and work with bigger granularity entities
- **Encapsulation**: protects an implementation from unintended actions and from inadvertent access
    - An object is an encapsulation of some data and related behavior in terms of two aspects:
        1. **Packaging aspect**: an object packages data and related behavior together into one self-contained unit
        2. **Information hiding aspect**: the data in an object is hidden from the outside world and are only accessible using the object's interface.
- **Class**: contains instructions for creating a specific kind of objects
    - The `this` keyword is a reference variable in Java that refers to the current object
    - `this` can be used to refer to a constructor of a class within the same class too
- **Enumeration**: a fixed set of values that can be considered as a data type
    - Java enums can have behaviours (ie. methods) defined in them
- **Inheritance**: an *is-a* relationship
    - A superclass is said to be more general than the subclass
    - Applying inheritance on a group of similar classes can result in the common parts among classes being extracted into more general classes
    - Multiple inheritence is allowed in Python and C++ but not Java and C#
- **Polymorphism**: allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object
    - Polymorphic code is shorter, simpler, and more flexible
- **Abstract class**: cannot be instantiated, but it can be subclassed
    - In Java, a class that does not have any abstract methods can be declared as an abstract class
    - Children classes of abstract classes must provide implementations or declare abstract the abstract methods of the inherited class
- **Interface**: an *is-a* relationship
    - A collection of method specifications
    - If a class implements the interface, it means the class is able to support the behaviors specified by the said interface
    - Can inherit from other interfaces using `extends`
- **Dynamic binding (aka late binding)**: mechanism where method calls in code are resolved at runtime, rather than at compile time
    - Overridden methods are resolved using dynamic binding, and therefore resolves to the implementation in the actual type of the object
- **Static binding (aka early binding)**: When a method call is resolved at compile time
    - Overloaded methods are resolved using static binding
- **Collection (aka container)**: object that groups multiple elements into a single unit
    - Used to store, retrieve, manipulate, and communicate aggregate data
    - **Collections framework**: a unified architecture for representing and manipulating collections

- **Exceptions**: used to deal with 'unusual' but not entirely unexpected situations
    - **Checked exceptions**: exceptional conditions that a well-written application should anticipate and recover from
        - All exceptions are checked exceptions except `Error`, `RuntimeException`, and their subclasses
        - eg. `FileNotFoundException`
    - **Errors**: exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from
        - eg. `IOError`
    - **Runtime exceptions**: conditions that are internal to the application, and that the application usually cannot anticipate or recover from
        - eg. `NullPointerException`
    - **Unchecked exceptions**: errors + runtime exceptions
- **Java**: primarily an OOP language
    - Supports limited forms of functional programming
    - Can be used (but not recommended) to write procedural code
    - **Strongly-typed language**: code works with only the object types that it targets
        - Can lead to unnecessary verbosity

Week 2

- **Software Engineering**: the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (IEEE Standard Glossary of Software)
- **Integrated Development Environments (IDEs)**: support most development-related work within the same tool
    - Source code editor + compiler/interpreter + debugger + other tools like testing/versioning
- **Testing**: operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component
- **Test cases**: can be determined based on the specification, reviewing similar existing systems, or comparing to the past behavior of the SUT
    - **Failures**: a mismatch between the expected behavior and the actual behavior, and indicates a potential defect (or a bug)
- **Regression**: modification which results in some unintended and undesirable effects on the system
    - **Regression testing**: the re-testing of the software to detect regressions; more practical when it is automated
- **Revision control**: the process of managing multiple versions of a piece of information
- **Remote repositories**: copies of a repo that are hosted on remote computers

Week 3

- **Javadoc**: a tool for generating API documentation in HTML format from doc comments in source
    - Can be read by maintainers as well as users (as JavaDocs are used for generating API documentation)
- **Package**: used to organise types for easier management
    - Package of a type should match the folder path of the source file
    - Package names are written in all lower case (not camelCase), using the dot as a separator

    - Importing a package does not import its sub-packages, as packages do not behave as hierarchies despite appearances
- **JAR (short for Java Archive)**: file format which Java applications are delivered in
- **Coding standards**: aim to make the entire code base look like it was written by one person
    - May contain rules that are subjective
- **Developer testing**: the testing done by the developers themselves as opposed to professional testers or end-users
    - Better to do early testing as late testing has the following cons:
        - Locating the cause of such a test case failure is difficult due to a large search space
        - Fixing a bug found during such testing could result in major rework
        - One bug might 'hide' other bugs
        - Delivery may have to be delayed if too many bugs were found during testing
- **Test driver**: the code that 'drives' the SUT for the purpose of testing
- **JUnit**: a framework for automated testing of Java programs
- **Unit testing**: testing individual units (methods, classes, subsystems, etc.) to ensure each piece works correctly
    - Java assertions are NOT used for unit testing
- **Stub**: has the same interface as the component it replaces, but its implementation is so simple that it is unlikely to have any bugs
    - Mimics the responses of the component, but only for the a limited set of predetermined inputs
    - Mimicked responses are hard-coded in the stub rather than computed or retrieved from elsewhere
    - Stubs are not the same as Mocks

Week 4

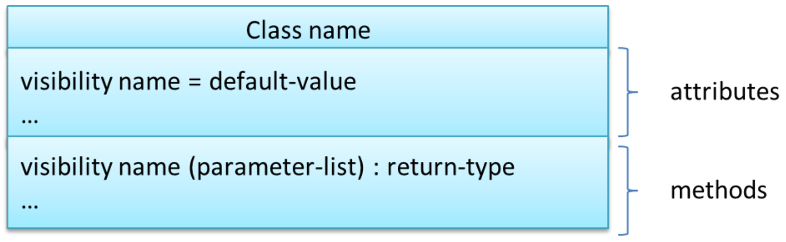
Models

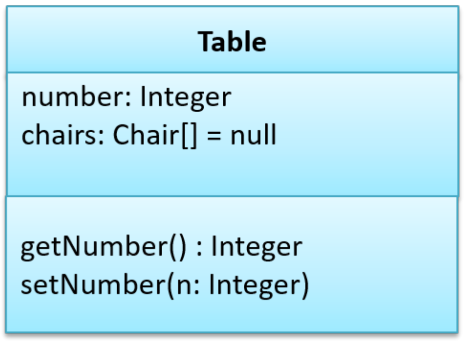
- **Model**: representation of something else
    - Can be considered abstractions because they provide a simpler view of a more complicated thing
    - Multiple models of the same entity may be needed to capture it fully
    - Can be used as a blueprint for creating software
- **Class diagrams**: models that represent a software design
    - Used to model class structures of an OO solution
    - Describes the structure but not the behaviour of software design

Usefulness of Models

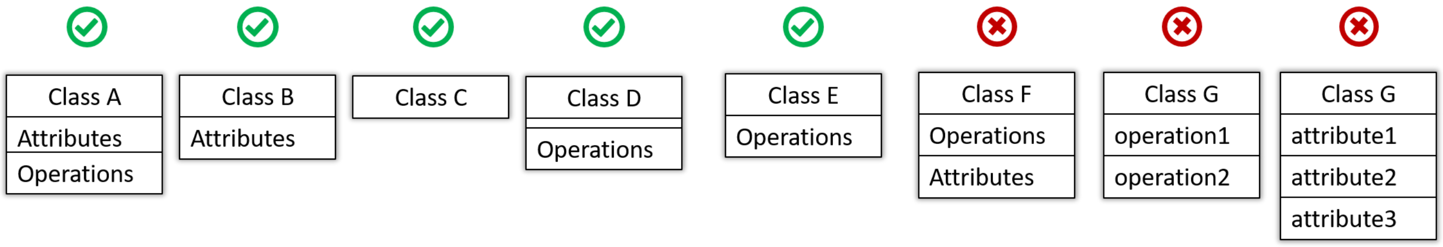
- Analyse a complex entity related to software development
- Communicate information among stakeholders
- Blueprint for creating software

Class Diagrams

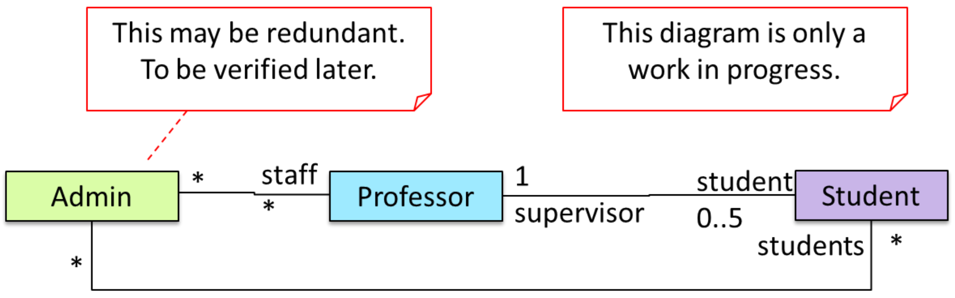

Class name
visibility name = default-value
...
visibility name (parameter-list) : return-type
...
attributes
methods



```
class Table {

    Integer number;
    Chair[] chairs = null;

    Integer getNumber() {
        //...
    }

    void setNumber(Integer n) {
        //...
    }
}
```

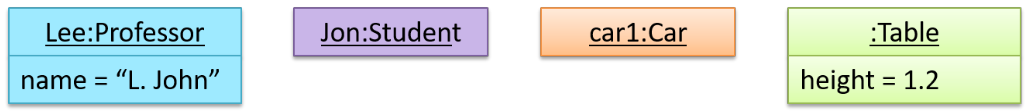


- Class attributes should be above class operations
- All attributes should be grouped in one compartment
- All operations should be grouped in one compartment
- Underlines denote class-level attributes and variables
- **Visibility** (NOT accessibility): `+` | `-` | `#` | `~` → `public` | `private` | `protected` | `package`
- UML Notes (for additional info) can be shown with or without a connection
- A constraint can be given inside a note, within curly braces `{}`


This may be redundant. To be verified later.
This diagram is only a work in progress.
Admin
*
staff
*
Professor
1
supervisor
student
0..5
Student
students
*
*

- Each `Student` must be supervised by exactly one `Professor`.
- A `Professor` can supervise 0 to 5 `Student`s inclusive.
- An `Admin` can handle any number of `Professor`s and any number of `Student`s, including none.
- A `Professor` / `Student` can be handled by any number of `Admin`s, including none

Object Diagrams

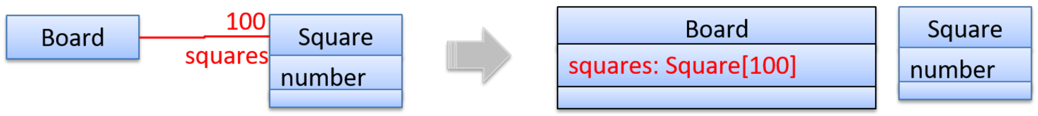


- The class/object name, like `car1:Car`, is underlined
- `objectName:ClassName`: "`objectName` is an instance of type `ClassName`".
- Methods are omitted
- Multiplicities are omitted
- Attributes compartment can be omitted
- Object name can be omitted e.g. `:Car`, an anonymous instance of type `Car`

Association

Associations are the main connections among classes in a class diagram

- Associations in an object structure can change over time
- Associations among objects can be generalised as associations between the corresponding classes
- Instance level variables to implement associations



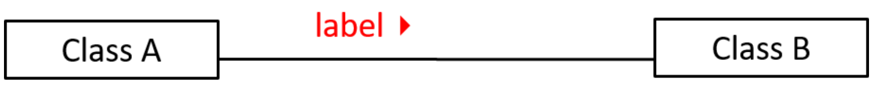
- An association can be shown as an attribute instead of a line
- Association that a `Board` has 100 `Square`s
- Show each association as either an attribute or a line but **not both**. A line is preferred is it is easier to spot

Roles

Associaiton Role labels are used to indicate the role played by classes in the association

Labels

Association labels describe the meaning of the association. Arrow head indicates direction in which label is to be read.



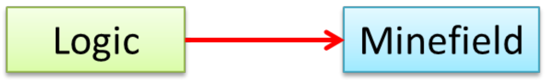
Multiplicity

- `0..1` 0 or 1
- `1` compulsory
- `*` 0 or more
- `n..m` number of linked objects must be `n` to `m` inclusive

Navigability

A *knows-a* relationship.

```
class Logic{
    Minefield minefield;
}

class Minefield{
    ...
}
```

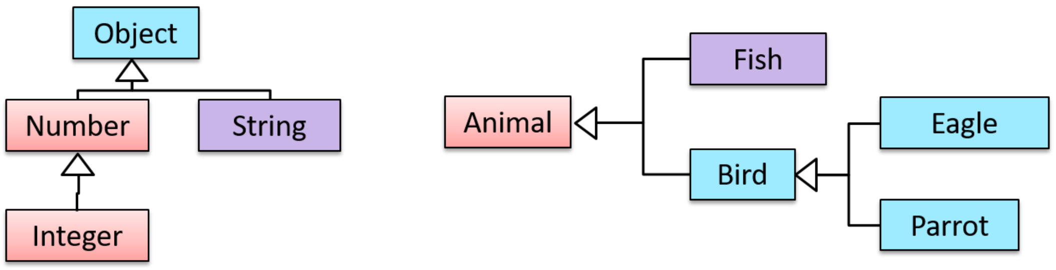


- `Logic` is aware of `Minefield`, but `Minefield` is not aware of `Logic`
- Is **not transitive**; If A $\rightarrow$ B and B $\rightarrow$ C, it does not imply that A $\rightarrow$ C

Inheritence

An *is-a* relationship. Eg. in diagram below, `Parrot` *is a* `Bird` and also *is an* `Animal`.

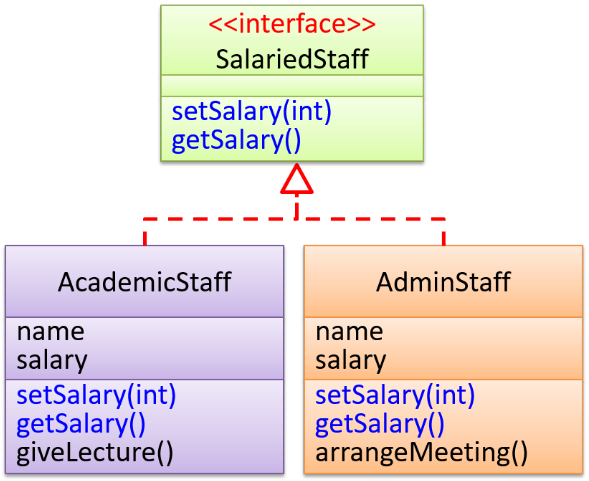
```
class Parrot extends Bird {
    ...
}
```



- Triangle points to parent class
- Superclass is more general than subclass
- Multiple inheritance is allowed in Python and C++ but not in Java, C#

Interface

An *is-a* relationship. An interface cannot implement from other interfaces but can extend from one or more interfaces. A class can implement from multiple interfaces.

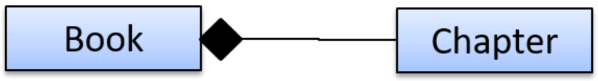


- Use the `<<interface>>` keyword

Composition

A *whole-part* relationship, when the *whole* is destroyed, *part*s are destroyed too. Ideally, the *part* may not even be visible to clients of the *whole* object.

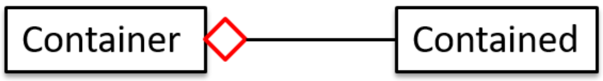
```
class Book {
    private Chapter chapter = new Chapter();
    ...
}
```



Aggregation

A *container-contained* relationship, weaker than a composition relationship. Unlike composition, the *contained* object can exist even after the *container* object is deleted.

```
class Team {
    Person leader;
    ...
    void setLeader(Person p) {
        this.leader = p;
    }
}
```



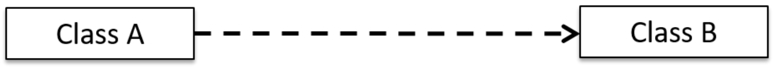
- Aggregation is not recommended to be used in UML because it adds more confusion than clarity

Dependency

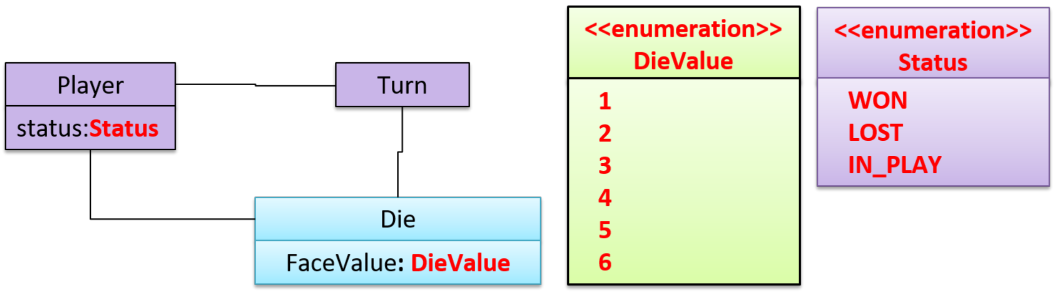
A *uses-a* relationship; a need for one class to depend on another without having a direction association with it.

```
class Foo {
    int calculate(Bar bar) {
        return bar.getValue();
    }
}

class Bar {
    int value;

    int getValue() {
        return this.value;
    }
}
```

- `Foo` has a dependency on `Bar`, but not an association
- The `Foo` object does not keep the `Bar` object it receives as a member of `Foo`



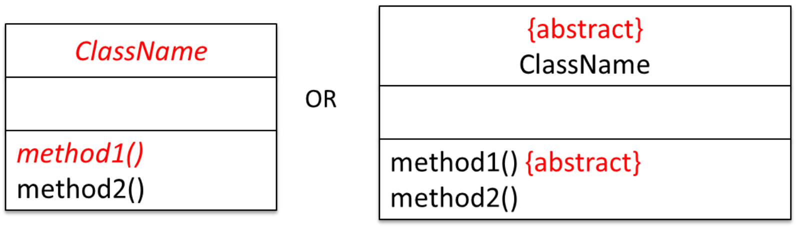
Enumeration



- Use the `<<enumeration>>` keyword
- This diagram has 2 enumerations in use: `DieValue` and `Status`
- **For object diagrams**, prefer to write as an inline attribute rather than a line

Abstract Classes

An `abstract` class cannot be instantiated, but it can be subclassed. Only `abstract` classes can have `abstract` methods. Non-`abstract` methods (ie. with method implementation) can be in `abstract` classes.

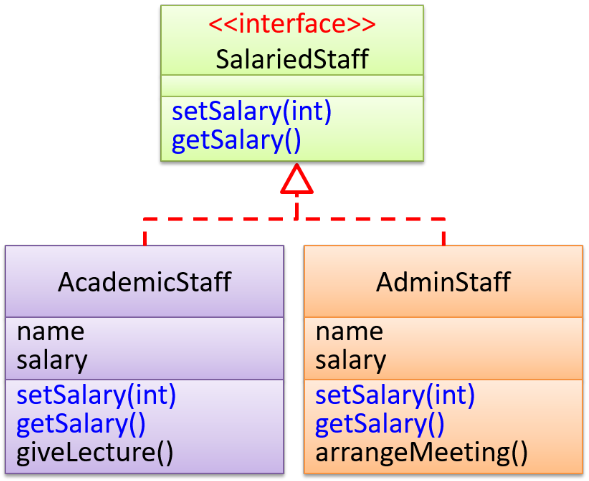


- Use *italics* or `{abstract}` keyword (preferred) to denote an `abstract` class/method
- A class that has an abstract method becomes an abstract class, not possible to create objects

Interface

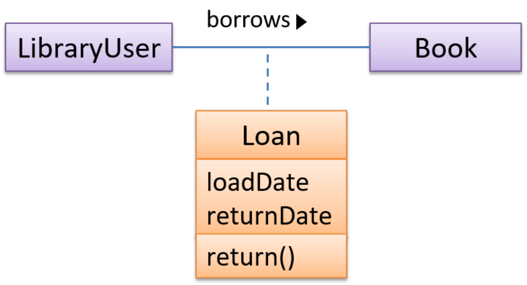
An interface is a behaviour specification ie. a collection of method specifications

- a class inplementing an interface results in an *is-a* relationship



Association Class

An association class represents additional information about an association.



- `Loan` is an association class because it stores information about the `borrows` association between the `User` and the `Book`

Week 5

- **Stakeholder**: a party that is potentially affected by the software product
  - eg. users, sponsors, developers, interest groups, government agencies, etc.
- **Software requirement**: a need to be fulfilled by the software product
- **Brown-field project**: develop a product to replace/update existing software product
- **Green-field project**: develop a totally new system with no precendent
- **Brainstorming**: a group activity to generate (not validate) a large number of diverse and creative ideas for the solution of a problem
- **Product surveys**: studying existing products to unearth shortcomings of existing solutions that can be addressed by a new product
- **User surveys**: used to solicit responses and opinions from a large number of stakeholders
- **Focus groups**: an informal interview within an interactice group setting
- **Prototype**: a mock up/scaled down version/partial system constructed to

    - Get users' feedback
    - Validate a technical concept
    - Give a preview/compare alternatives
    - Field test under controlled conditions
    - *Discover* as well as *specify* requirements
- **Feature list**: list of features of a product grouped accordingly to some criteria (such as aspect, priority, order of delivery, etc.)
- **Glossary**: serves to ensure all stakeholders have a common understanding of the noteworthy items/abbreviations/acronyms etc.
- **Refactoring**: improving a program's internal structure in small steps without modifying its external behaviour
    - Is not rewriting: discarding poorly-written code entirely and re-writing it from scratch is not refactoring because refactoring needs to be done in small step
    - Is not bug fixing: By definition, refactoring is different from bug fixing or any other modifications that alter the external behavior (e.g. adding a feature) of the component in concern
    - Can improve the performance of the refactored code
- **Assertions**: used to define assumptions about the program state so that the runtime can verify them
    - Java disables assertions by default
        - Thus, do not use assertions to do actual work
    - Not used in unit testing nor in exception handling
    - Suitable for verifying assumptions about internal invariants, control-flow invariants, preconditions, postconditions, and class invariants
    - Both Exceptions (indicates unusual condition created by user) and Assertions (indicates programmer's mistake) serve different purposes and thus should both be utilised in code
- **Integration**: combining parts of a software to form a whole
- **Build automation tools**: automate the steps of the build process by means of a build script
    - Also serves as dependency management tools

Requirements

- **Functional requirements, FRs**: specify what the system should do (for a specific user type)
    - eg. the application admin should be able to view a log of user activities
- **Non-functional requirements, NFRs**: specify the constraints under which the system is developed and operated
    - Business rules: eg. size of minefield cannot be smaller than 5
    - Technical requirements: eg. system should work on both 32 and 64 bit systems
    - Easier to miss, stakeholders tend to think of FRs first
- **Supplementry requirements**: to capture requirements that do not fit elsewhere, typically where most NFRs are listed
- Well defined requirements are:
    - Unambiguous
    - Testable
    - Clear (concise, terse, simple, precise)
    - Correct
    - Understandable

    - Feasible
    - Independent
    - Atomic (not divisable any further)
    - Necessary
    - Implementation-free (ie. abstract)
- **Atomic**: cannot be divisible any further
- **Implementation-free**: the what, not the how
- Can be discarded if they are considered "out of scope"

User Stories

Short, simple descriptions of a feature told from the perspective of the person who desires said feature written in natural language (NOT formal language).

- Format: "As a `{user type/role}`, I can `{function}` so that `{benefit}`"
    - `{benefit}` (third part) can be omitted if it is obvious; rest must be present
- eg. As a lecturer, I can create discussion forums, so that students can discuss things online
- Different from traditional requirement specifications: do not contain enough details to form a complete system specification
- Fine to add more details such as conditions, priority, urgency, effort estimates etc.
- Convenient for scoping, estimation, and scheduling
- Can capture NFRs because even NFRs must benefit some stakeholder
- Handy for recording requirements during early stages of requirements gathering
- High-level user stores, called epics (or themes) cover bigger functionality
    - eg. `[Epic]` As a lecturer, I can monitor student participation levels
        - As a lecturer, I can view the forum post count of each student so that I can identify the activity level of students in the forum
        - As a lecturer, I can view webcast view records of each student so that I can identify the students who did not view webcasts
        - As a lecturer, I can view file download statistics of each student so that I can identify the students who do not download lecture materials

Steps to Take

1. Define the target user
2. Define the problem scope
3. Don't be too hasty to discard "unusual" user stories
4. Don't go into too much details
5. Don't be biased by preconceived product ideas
6. Don't discuss implementation details or whether you are actually going to implement it

Code Quality

- Use guard clauses to make the "happy path" more prominent
- Class names should be nouns and method names should be verbs

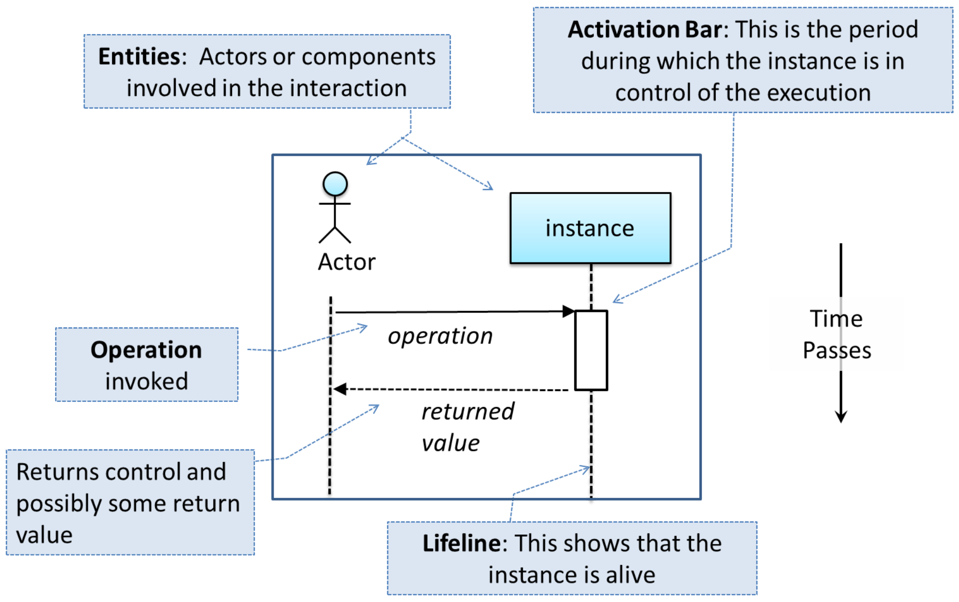
- Variables should be defined in the least possible scope (ie. within the if block if it is only used in there)
- Comments should explain WHAT and WHY, rather than HOW

Week 6

- **UML sequence diagram**: captures the interactions between multiple objects for a given scenario
  - Time goes from top-to-bottom
- **Architecture**: shows the overall organisation of the system and can be viewed as a very high-level design
  - Should be simple and technically viable structure
  - Forms basis of implementation
  - Typically designed by the software architect
- **Architecture diagram**: are free-from diagrams; no universally adopted standard notation
  - Minimise variety of symbols, if the symbols you choose do not have widely-understood meanings, explain
  - Avoid the indiscriminate use of double headed arrows to show interactions
- **Multi-level design**:
  - Design of the entire system can be shown in one place in a smaller system
  - Design of bigger systems needs to be done/sown at multiple levels
- **Debugging**: the process of discovering defects in the program
  - Bad
    - Inserting temporary print statements
    - Manually tracing through code
  - Good
    - Use a debugger: pause the execution, step through one statement at a time while examining the internal state if necessary
- **Logging**: the deliberate recording of certain information during a program execution for future reference
  - Useful for troubleshooting problems
  - Most programming environments come with logging systems that allow sophisticated forms of logging.
  - Features: ability to enable and disable logging easily or change logging intensity
  - Java has default logging mechanism: `import java.util.logging.*`
- **Log file**: like the black box of an airplane
- **Markdown**: a lightweight markup language with plain text formatting syntax

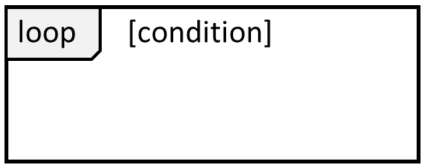
UML Sequence Diagrams

Basic Notation


Entities: Actors or components involved in the interaction
Activation Bar: This is the period during which the instance is in control of the execution
Actor
instance
Time Passes
Operation invoked
operation
returned value
Returns control and possibly some return value
Lifeline: This shows that the instance is alive

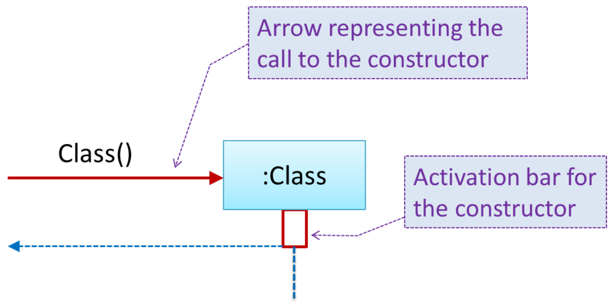
- Object names do not contain underlines
- Activation bar must start when method call arrives, remain unbroken, and end when method has returned

Loops



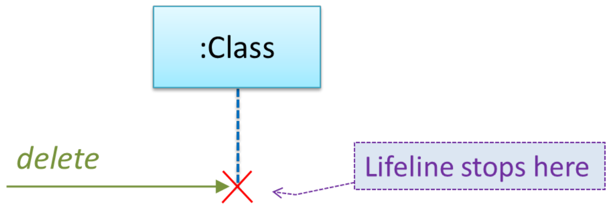
- Note the "chipped" end
- `loop` keyword must be used, regardless of kind of loop

Object Creation


Arrow representing the call to the constructor
Class()
:Class
Activation bar for the constructor

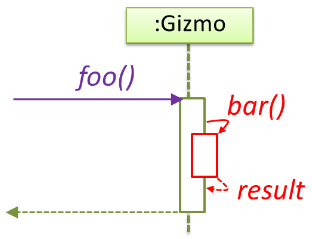
- The arrow that represents the constructor arrives at the side of the box representing the instance
- The activation bar represents the period the constructor is active

Object Deletion

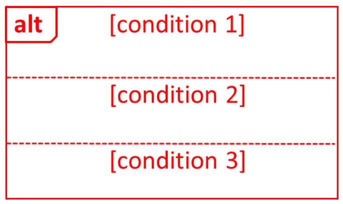


- Note that the lifeline should not go beyond the X

Self Invocation

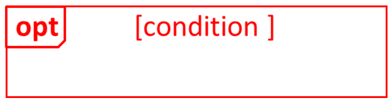


Alternative Paths



- Similar to `if-else` blocks, **one of the cases must run**

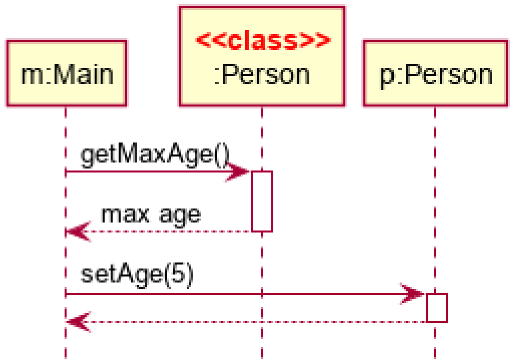
Optional Paths



- Similar to an `if` block, **code need not run**

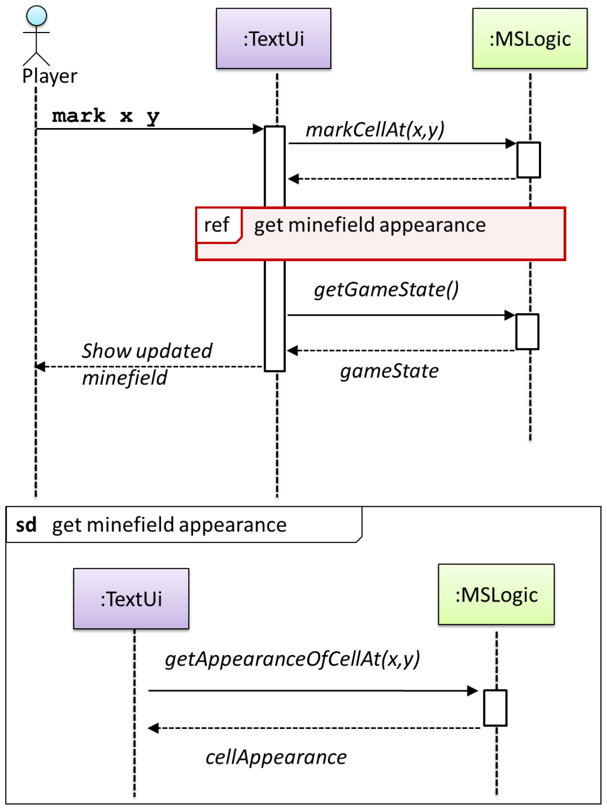
Calls to Static Methods

Method calls to `static` i.e., class-level methods are received by the class itself, not an instance of the class



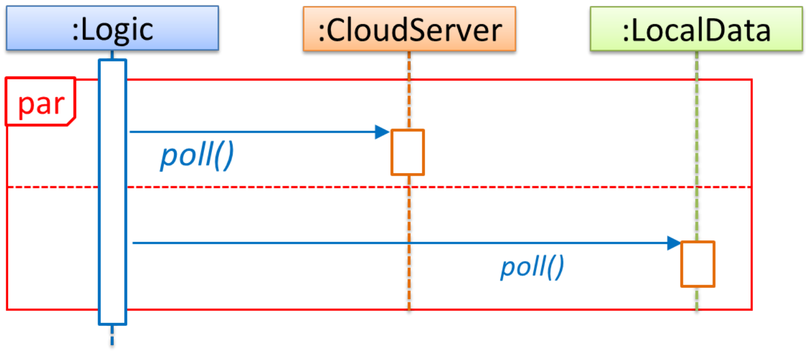
Reference Frames

Allows a segment of the interaction to be omitted and shown as a separate sequence diagram. Reference frames help us to break complicated sequence diagrams into multiple parts or simply to omit details we are not interested in showing.



Parallel Paths

Methods call in parallel. Corresponding Java implementation is likely to be multi-threaded as a normal Java program is single-threaded.



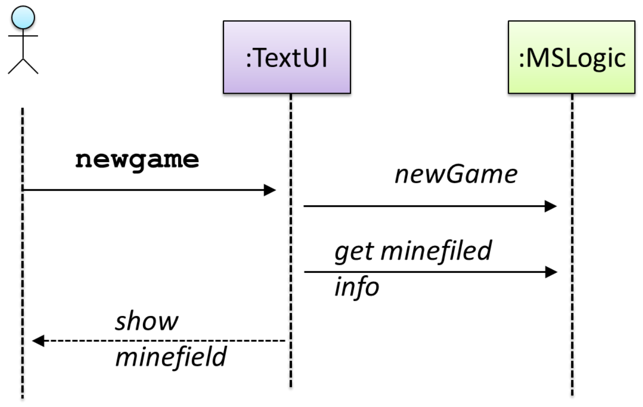
- `Logic` is calling `CloudServer#poll()` and `LocalServer#poll()` in parallel

Optional Components

Some components may be omitted if they do no result in ambiguities.

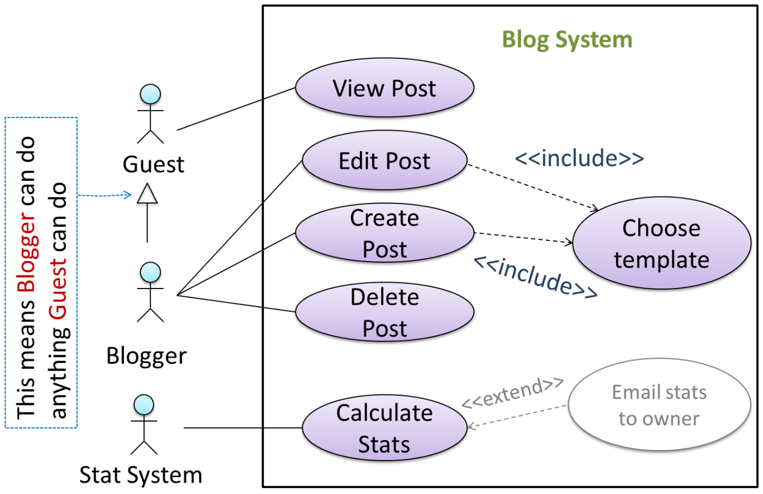
- Activation bars
- Return arrows

Example:



Week 7

- **Use case**: description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor
    - Describes an interaction between the user and the system for a specific functionality of the system
    - Can have multiple **actors** (who are not part of the system and reside outside)
    - "System" can be an actor to indicate that something is done by the system itself without being initiated by a user or an external system
        - NOT recommended; limit use cases for modeling behaviours that involve an external actor
    - Main body: sequence of steps that describes the interaction between the system and the actors
    - Describes only the externally visible behaviour, not internal details of the system
        - ie. no need to write "saves into the cache"
        - should minimise details that are not part of the interaction between user and system
    - Steps give the intention of the actor (not the mechanics)
        - ie. UI details are usually omitted or as general as possible with regards to UI
    - Captures the functional requirements of the system
    - Can be specified at various levels of detail
    - Advantages of documenting system requirements as use cases:
        - Can be fairly detailed but still natural enough for users for users to understand and give feedback
        - UI-independent, allowing system designers more freedom to decide how a functionality is provided
        - Extensions encourage us to consider all situations a software product might face during its operations
        - Encourage us to identify and optimise the typical scenario of usage over exceptional usage scenarios
    - Not good for capturing requirements that does not involve a user interacting with the system; should not be used as the sole means to specify requirements
    - Can be given (but not necessary) a unique ID to identify them
    - **Main Success Scenario (MSS)**: describes the most straightforward interaction for a given use case assuming nothing goes wrong
        - Self-contained: give us complete usage scenario
    - **Extensions**: "add-on"s to the MSS that describe exceptional/alternative flow of events
        - Either of the extensions marked `3a.` and `3b.` can happen just after step `3`
        - Extension marked as `*a`. can happen at any step
    - **Inclusions**: including another use case (denoted with underlines)
        - Don't want to clutter use case with too many low-level steps

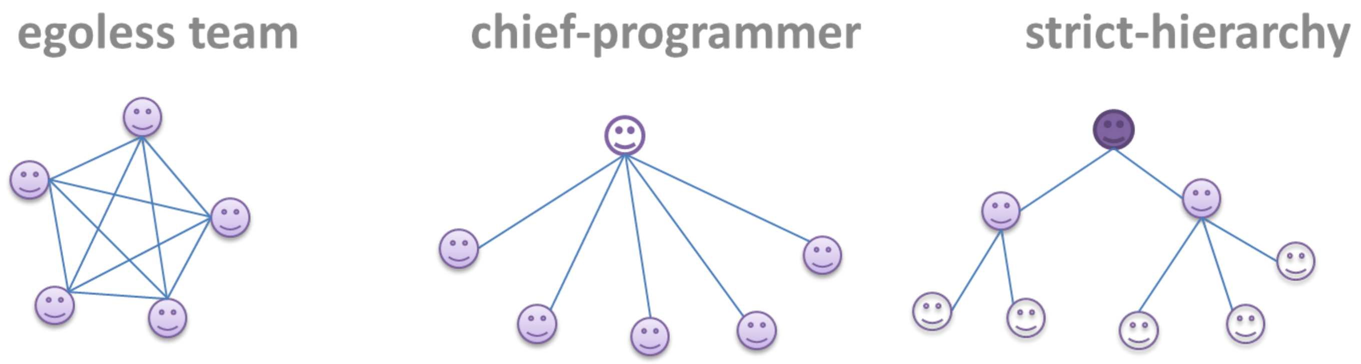
        - Set of steps is repeated in multiple use cases
    - **Preconditions**: specify the specific state we expect the system to be in before the use case starts
    - **Guarantees**: specify what the use case promises to give us at the end of its operation
- **Use case diagrams**: provide a visual "table of contents" of the use cases of a system
    - `<<extend>>` arrows to show extensions
        - Direction of the arrow is from the extension to the use case it extends, arrow uses dashed line
    - `<<include>>` arrows to show inclusion
        - Direction of arrow is from the original use case to the use case it includes (opp from extend), dotted arrow
- **Actor**: role played by a user, can be a human or another system
    - Not part of the system/reside outside the system
    - Can be invovled in many use cases
    - A single person/system can play many roles
    - Many persons/systems can play a single role
    - Actor generalisation:
        - 

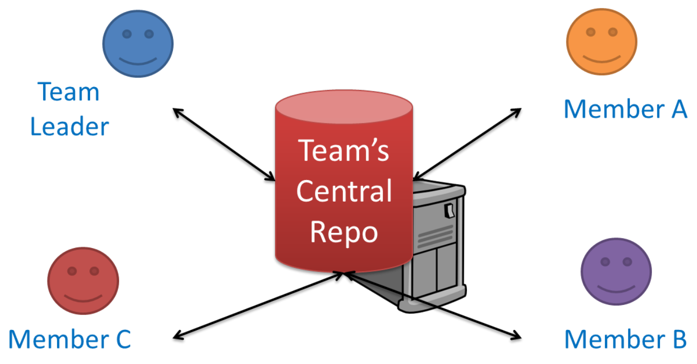
- **Design**: the creative process of transforming the problem into a solution; the solution is also called design
- **Product/external design**: designing the external behavior of the product to meet the users' requirements
- **Implementation/internal design**: designing how the product will be implemented to meet the required external behavior
- **Abstraction**: technique for dealing with complexity
    - Works by establishing a level of complexity we are interested in and suppressing the more complex details below that level
    - Can be applied repeatedly to obtain progressively higher levels of abstractions
    - Can be divided into two sub-types (but NOT limited to):
        - **Data abstraction**: abstracting away the lower level data items and thinking in terms of bigger entities
        - **Control abstraction**: abstracting away details of the actual control flow to focus on tasks at a higher level
- **Coupling**: a measure of the degree of dependence between components, classes, methods, etc.
    - `X` is coupled to `Y` if a change to `Y` can potentially require a change in `X`
        - If `Foo` class calls the method `Bar#read()`, `Foo` is coupled to `Bar` because a change to `Bar` can potentially (but not always) require a change in `Foo`
    - Some examples of coupling:

    - `A` has access to internal structure of `B` (high coupling)
    - `A` and `B` depend on same global variable
    - `A` calls `B`
    - `A` receives an object of `B` as a param or return value
    - `A` inherits from `B`
    - `A` and `B` are required to follow the same data format or communication protocol
  - High/tight/strong coupling is discouraged as:
    1. Maintenance is harder due to ripple effect; risk of regression is higher
    2. Integration is harder because multiple components coupled with each other have to be integrated at the same time
    3. Testing and reuse of the module is harder
- **Cohesion**: a measure of how strongly-related and focused the various responsibilities of a component are
  - Prefer higher cohesions, disadvantages of low/weak cohesions:
    1. Lowers the understandability of modules as it is difficult to express module functionalities at a higher level
    2. Lowers maintainability because a module can be modified due to unrelated causes or many modules may need to be modified to acheive a small change in behaviour
    3. Lowers reusability of modules because they do not represent logical units of functionality
  - Can be present in many forms:
    1. Code related to a single concept is kept together
    2. Code that is invoked close together in time is kept together
    3. Code that manipulates the same external data structure (like storage) is kept together
- **Multi-level design**: can be done in a top-down manner, bottom-up manner, or as a mix:
  - **Top-down**: Design the high-level design first and flesh out the lower levels later
    - Useful when designing big and novel systems where the high-level design needs to be stable before lower levels can be designed
  - **Bottom-up**: Design lower level components first and put them together to create the higher-level systems later. Usually not scalable
    - Useful when designing a variation of an existing system or re-purposing existing components to build a new system
    - Not usually scalable for bigger systems
  - **Mix**: Design the top levels using the top-down approach but switch to a bottom-up approach when designing the bottom levels
- **Agile design**: the overall system design will emerge over time, rather than planning it all at the start
  - Some initial architectural modeling is done at the very beginning
  - Does not produce fully documented set of models in place before coding
  - However, the architecture is still expected to remain relatively stable
- **Late and one-time integration**: wait till all components are completed and integrate all finished components near the end of the project
  - Not recommended as integration often causes many component incompatibilities (due to previous miscommunications)
- **Early and frequent integration**: integrate early and evolve each part in parallel, in small steps, re-integrating frequently

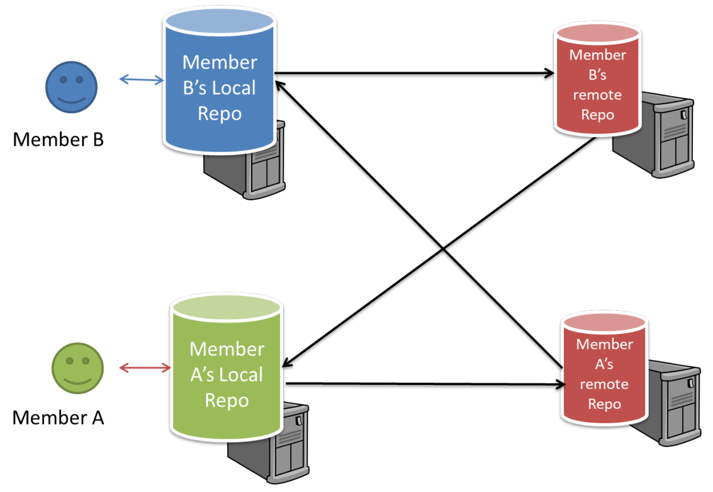
- **Walking skeleton**: an initial integration written by one developer which other develeopers can flesh out in parallel over time

- **Big-bang integration**: integrate all components at the same time
  - Not recommended as it will uncover too many problems at the same time, making debugging more complex
- **Incremental integration**: integrate few components at a time
  - Better than the big-bang integration because it surfaces integration problems in a more manageable way
- **Milestone**: end of a stage which indicates a significant progress
- **Buffer**: time set aside to absorb any unforseen delays
  - Should not inflate task estimates to create hidden buffers, have explicit buffers instead
- **Issue trackers**: Also called bug trackers, commonly used to track task assignments and progress (eg. GitHub, SourceForge, BitBucket)
- **Work Breakdown Structure (WBS)**: depicts information about tasks and their details in terms of subtasks
  - **Effort**: traditionally measured in man hour/day/month
  - **Task**: should be well defined
- **Team Structures**:



  - Egoless Team: every member is equal in interms of responsibility and accountability, absence of an authority figure to manage the team and resolve conflicts
  - Chief-programmer: Single authoritative figure
  - Strict-hierarchy: Strictly defined organisation among tem members, good for large, resouce-intensive and complex projets as communication overhead is reduced
- **Centralized RCS (CRCS)**: uses a central remote repo that is shared by the team
  - eg. SVN, CVS
  - 

- **Distributed/Decentralized RCS (DRCS)**: allows multiple remote repos
  - eg. Git/Mercurial
  -



Week 8

- **Integration Testing**: testing whether different parts of the software work together (i.e. integrates) as expected
  - Suppose a class `Car` uses classes `Engine` and `Wheel`
    1. Unit test `Engine` and `Wheel`
    2. Using stubs for `Engine` and `Wheel`, unit test `Car` in isolation
    3. Integration test for `Car` using it together with `Engine` and `Wheel`
  - In practice, developers often use a hybrid of unit+integration tests to minimise the need for stubs
    - Skip step 2 for above example
    - Downside is that `Car` is never tested in isolation of its dependencies
    - Risk is minimal as the dependencies are already unit tested in step 1
  - Not simply a case of repeating the unit test cases using the actual dependencies (instead of stubs); integration tests are additional test cases that focus on the interactions between the parts
- **System Testing**: testing the *whole system* against the *system specification*
  - Typically done by the QA team
  - To verify that it conforms to the specified external behavior of the system
  - Test cases are based on the specific external behaviour of the system
  - System tests can go beyond the bounds defined in the specification
  - Includes testing against NFRs, eg. **performance, load/stress/scalability, security, compatability/interoperability, usability, portability testing**
- **GUI testing**: much harder than testing the CLI/API as
  - Most GUIs can support large number of different operations in arbitary order
  - Harder to automate compared to API/CLI testing
  - Appearance of a GUI can be different across platforms/environments
  - Moving as much logic out as possible to make GUi testing easier
  - Tools: TestFx, VisualStudio, Selenium
- **User Acceptance Testing (UAT)**: testing the system to ensure it meets the user requirements
  - Done by actual users or professional testers representing the users
  - Done on the deployment site or on a close simulation of the deployment site
- **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 of the system in their natural work setting

- **Scripted testing**: write a set of test cases based on the expected behavior of the SUT (Software Under Test), and then perform testing based on that set of test cases
    - More systematic, more likely to discover more bugs given sufficient time compared to exploratory testing
- **Exploratory testing**: devise test cases on-the-fly, creating new test cases based on the results of the past test cases
    - Success depends on the tester's prior experience and intuition
    - While it may detect some problems in a relative short time, it is not prudent to use this as a sole means of testing a critical system
    - Also known as **reactive testing, error guessing technique, attack-based testing, bug hunting**
    - A mix of both scripted and exploratory testing is better
- **Dependency injection**: the process of 'injecting' objects to replace current dependencies with a different object
    - Often used to inject stubs to isolate the SUT fromits dependencies so that it can be tested in isolation
- **Testability**: an indication of how easy it is to test an SUT
- **Test coverage**: a metric used to measure the extent to which testing exercises the code
    - **Function/method coverage**: based on functions executed
    - **Statement coverage**: based on the number of line of code executed
    - **Decision/branch coverage**: based on the decision points exercised
    - **Condition coverage**: based on the boolean sub-expressions, each evaluated to both true and false with different test cases
        - Is NOT the same as decision coverage
        - `if (x > 2 && x < 44)`: one decision/branch but two conditions
    - **Path coverage**: measures coverage in terms of possible paths through a given part of the code executed
        - 100% path coverage means all possible paths have been executed
        - 100% path coverage does NOT mean code is bug free (ie. some branches missing from the original code like `if` without `else`)
    - **Entry/exit coverage**: measures coverage in terms of possible calls to and exits from the operations in the SUT
    - **Coverage analysis tools**: to measure coverage
    - **Coverage analysis**: useful in improving the quality of testing
- **Code review**: the systematic examination code with the intention of finding where the code can be improved
    - Can detect functionality defects as well as other problems such as coding standard violations
    - Can verify non-code artifacts and incomplete code
    - Do not require test drivers or stubs
    - Manual process and therefore, error prone
- **Static analysis**: analysis of code without actually executing the code
    - **Linters**: a subset of static analyzers that specifically aim to locate areas where the code can be made 'cleaner'
    - Can perform more complex analysis like locating potential bugs, memory leaks, inefficient code structures etc.
    - Most modern IDEs come with some inbuilt static analysis capabilities
- **Dynamic analysis**: analysis of code by executing the code
    - eg. code coverage as reported by Intellij IDEA

Acceptance Test vs. System Test

- Acceptance testing comes after system testing
- Passing system tests does not necessarily mean passing acceptance testing
    - System might work on the testbed environments but might not work the same way in the deployment environment, due to subtle differences between the two environments
    - System might conform to the system specification but could fail to solve the problem it was supposed to solve for the user, due to flaws in the system design

System Testing	Acceptance Testing
Done against the system specification	Done against the requirements specification
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	Done on the deployment site or on a close simulation of the deployment site
Both negative and positive test cases	More focus on positive test cases

Requirements Specification	System Specification
limited to how the system behaves in normal working conditions	can also include details on how it will fail gracefully when pushed beyond limits, how to recover, etc. specification
written in terms of problems that need to be solved (e.g. provide a method to locate an email quickly)	written in terms of how the system solve those problems (e.g. explain the email search feature)
specifies the interface available for intended end-users	could contain additional APIs not available for end-users (for the use of developers/testers)

- In many cases, one document serves as both a requirement specification and a system specification

Week 9

- **Object Oriented Domain Models (OODMs)**: used to model objects in the problem domain
    - Aka **conceptual class diagrams**
    - Can be helpful in building an OO solution
    - Do not contain solution-specific classes
        - OODMs describe the problem domain while class diagrams describe the solution
    - Represents the class structure of the problem domain and not their behaviour, just like class diagrams
    - **Omits methods and navigability**, otherwise notation is similar to class diagrams
    - Can pass off as a class diagram since OODM uses a subset of class diagram notations
- **Workflows**: the flow in which a process or a set of tasks is executed
    - Flow charts and AD can be used to model workflows
- **Activity Diagrams (AD)**: used to model workflows
    - Captures an activity of actions and control flows that makes up the activity

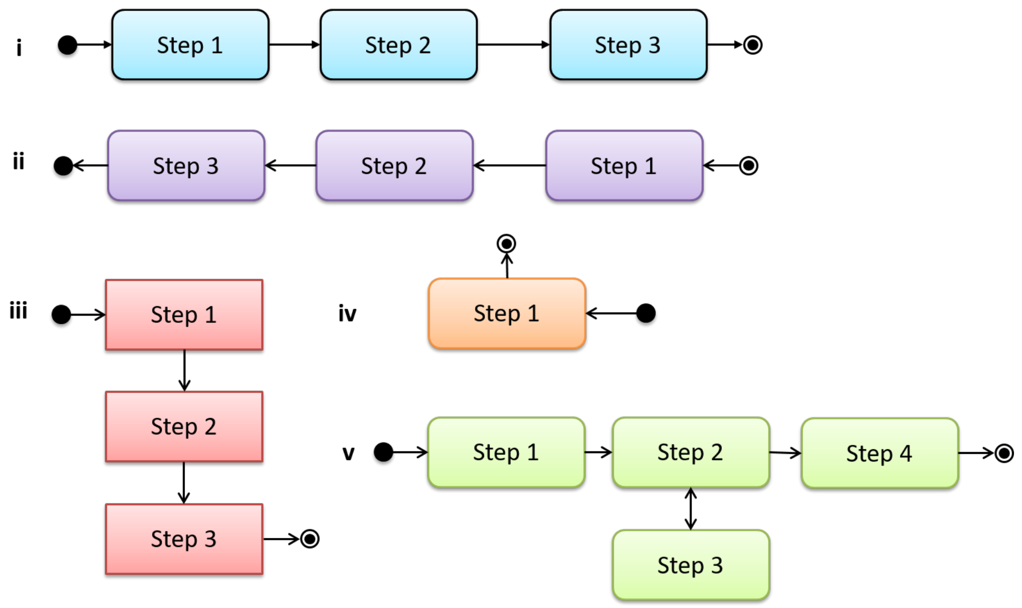
- **Single Responsibility Principle (SRP)**: a *class* should have one, and only one, responsibility and reason to change
    - Highly related to the concept of *cohesion*
    - Only refers to *classes*, NOT *methods*
- **Open-Closed Principle (OCP)**: modules should be open for extension but closed for modification (ie. can be extended, without requiring them to be modified)
    - Aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself
    - We should be able to change a software module's behavior without modifying its code
    - In OOP, this often requires separating the specification (ie. interface) of a module from its implementation
        - Eg. the behavior of a Java generic class can be altered by passing it a different class as a parameter
            - Eg. `ArrayList students = new ArrayList<Student>();`
- **Separation of Concerns Principle (SoC)**: seperation of the code into distinct sections, such that each section addresses a separate concern
    - **Concern**: set of information that affects the code of a computer program
        - Eg. A specific feature, such as the code related to `add employee` feature
        - Eg. A specific aspect, such as the code related to persistence or security
        - Eg. A specific entity, such as the code related to the `Employee` entity
    - Reduces functional overlaps among code sections
    - Limits the ripple effect when changes are introduced to a specific part of the system
    - Can be applied at the class level, as well as on higher levels
    - Should lead to higher cohesion and lower coupling
- **Liskov Substitution Principle (LSP)**: derived classes must be substitutable for their base classes
    - A subclass should not be more restrictive than the behavior specified by the superclass
    - Eg. `Square` class should not extend from `Rectangle` class since squares cannot change their width without changing their length, unlike rectangles
    - Any time inheritance is being used, LSP must be considered
- **Law of Demeter (LoD)**: aka. principle of least knowledge
    - An object should have limited knowledge of another object
    - An object should only interact with objects that are closely related to it
    - Aims to prevent objects navigating internal structures of other objects
    - Method `m` of an object `O` should invoke only the methods of the following kinds of objects:
        1. `O` itself
        2. Objects passed as parameters of `m`
        3. Objects instantiated in `m` (directly or indirectly)
        4. Objects from the direct association of `O`
- **Interface Segregation Principle (ISP)**: no client should be forced to depend on methods it does not use
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules, both should depend on abstractions
    - Abstractions should not depend on details; details should depend on abstractions
- **SOLID Principles**: SRP, OCP, LSP, ISP, DIP
- **Software Development Life Cycle (SDLC)**: different stages such as requirements, analysis, design, implementation, and testing
    - SDLC models (aka Software Process Models) describe different ways to go through the SDLC

        - Prescribes a roadmap which describes the aims of the development stage(s), the artifacts or outcomes of each stage, and the workflow between stages
- **Sequential model (aka Waterfall model)**: models software development as a linear process
    - When one stage of the process is completed, it should produce some artifacts to be used in the next stage
        - Eg. upon completion of the requirement stage a comprehensive list of requirements is produced that will see no further modifications
    - Could be a useful model when the problem statement that is well-understood and stable
    - Major problem with this model is that requirements of a real-world project are rarely well-understood at the beginning and keep changing over time
- **Iterative/Incremental model**: advocates having several iterations of SDLC
    - Each of the iterations produces a new version of the product
    - Can take a breadth-first (all major components in parallel) or a depth-first (fleshing out some components) approach to iteration planning
        - Most project use a mixture of breadth-first and depth-first iterations
- **Agile model**: an alternative to documentation-driven, heavyweight software development processes
    - Requirements are prioritized based on the needs of the user, are clarified regularly (at times almost on a daily basis) with the entire project team, and are factored into the development schedule as appropriate
    - Team works based on a rough project plan and a high level design that evolves as the project goes on, instead of doing a very elaborate and detailed design and a project plan for the whole project
    - Strong emphasis on complete transparency and responsibility sharing among the team members
- **Scrum**: a process skeleton that contains sets of practices and predefined roles
    - **The Scrum Master**: maintains the processes (typically in lieu of a project manager)
    - **The Product Owner**: represents the stakeholders and the business
    - **The Team**: a cross-functional group who do the actual analysis, design, implementation, testing, etc.
    - **Sprint**: basic unit of development which scrum projects are divided into
        - Tends to last between one week and one month
        - Each sprint is preceded by a planning meeting
            - Tasks are identified
            - Estimated commitment for the sprint goal is made
            - Review or retrospective meeting of progress and lessons for the next sprint
        - During each sprint, the team creates a potentially deliverable product increment
    - Enables the creation of self-organizing teams by encouraging co-location of all team members
    - Designed to accommodate requirements churn
    - Recognises that during a project the customers can change their minds about what they want and need (often called requirements churn)
    - **Daily scrum**: a meeting strictly time-boxed to 15 minutes where each member answers the following:
      1. What did you do yesterday?
      2. What will you do today?
      3. Are there any impediments in your way?
        - Note: NOT used for problem-solving or issue resolution; issues that are raised are taken offline and usually dealt with by the relevant subgroup immediately after the meeting
- **Extreme Programming (XP)**: an agile process
    - Stresses customer satisfaction

    - Aims to empower developers to confidently respond to changing customer requirements, even late in the life cycle
    - Emphasizes teamwork where teams self-organizes around the problem to solve it as efficiently as possible
    - Aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage
    - Has a set of 5 simple rules where individual pieces make no sense, but when combined together a complete picture can be seen
        - Planning, managing, designing, coding, testing
- **Developer-to-developer documentation**: can be in one of two forms,
    1. **For developer-as-user**: documentation for software components written by developers for use by other developers, can take forms like
        - **API documentation**: APIs expose functionality in small-sized, independent and easy-to-use chunks, each of which can be documented systematically
        - **Tutorial-style instructional documentation**: In addition to explaining functions/methods independently, some higher-level explanations of how to use an API can be useful
    2. **For developer-as-maintainer**: documentation for how a system or component is designed, implemented, and tested
        - Is usually harder because of the need to explain complex internal details
        - Only some information need to be included in the documentation, as code (and code comments) can also serve as a complementary source of information
    - Can be divided into 4 parts: tutorials, how-to guides, explanations, and technical references
    - Use plenty of diagrams
    - Use plenty of examples
    - Use simple and direct explanations
    - Get rid of statements that do not add value
    - Not a good idea to have separate sections for each type of artifact
    - Top-down breadth-first explanation is easier to understand than a bottom-up one
        - Main advantage of top-down breadth-first is that the document is structured like an upside down tree (root at the top) and the reader can travel down a path she is interested in until she reaches the component she is interested to learn in-depth
    - Aim for 'just enough' developer documentation
        - Writing and maintaining developer documents is an overhead which should be minimised
        - If the readers are developers who will eventually read the code, the documentation should complement the code and should provide only just enough guidance to get started
    - Focus on providing higher level information that is not readily visible in the code or comments
    - Describe the similarity in one place and emphasize only the differences in other places to reduce duplicating chunks or text

Activity Diagram (AD)

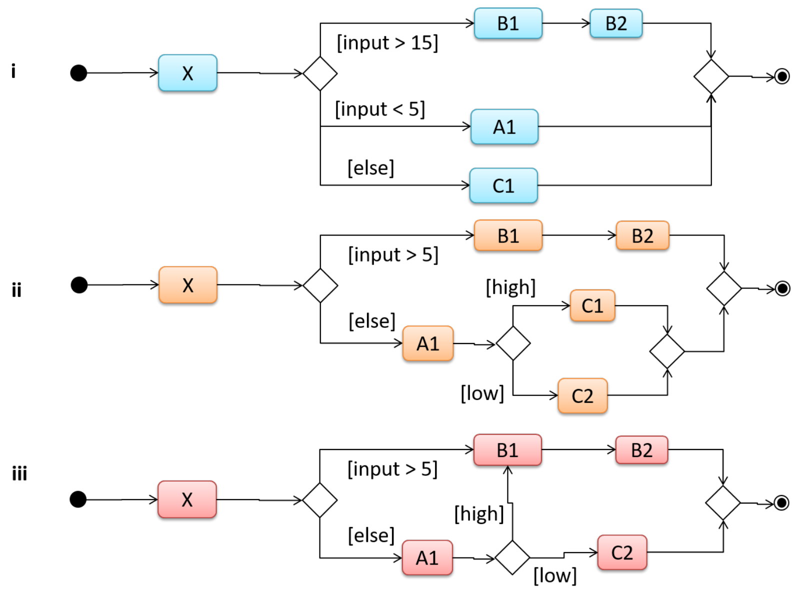
- **Action**: a single step in an activity, shown as a rectangle with rounded courners
- **Control flow**: shows the flow of control from one action to the next, show by drawing a line with an arrow-head to show direction of flow

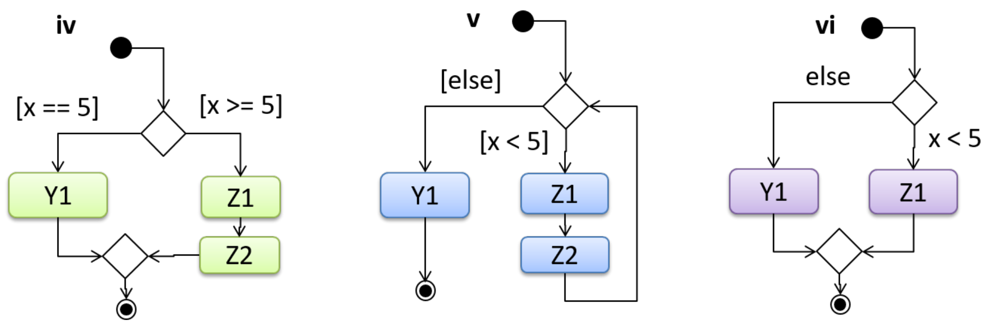


- Note the difference between the start node and the end node
- Only diagrams i and iv are correct
- Diagram **v is wrong**: there cannot be double-headed arrows

---

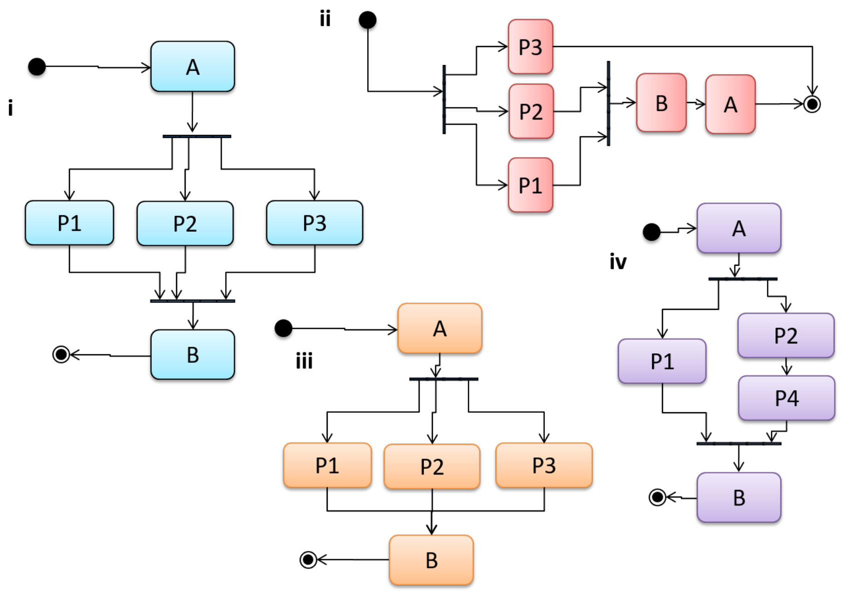
- **Branch node**: shows the start of alternate paths
  - Each control flow exiting a branch node has a guard condition (a boolean condition that should be true for execution to take that path)
  - Only one of the guard condition can be true at any time
- **Merge node**: shows the end of alternate paths.





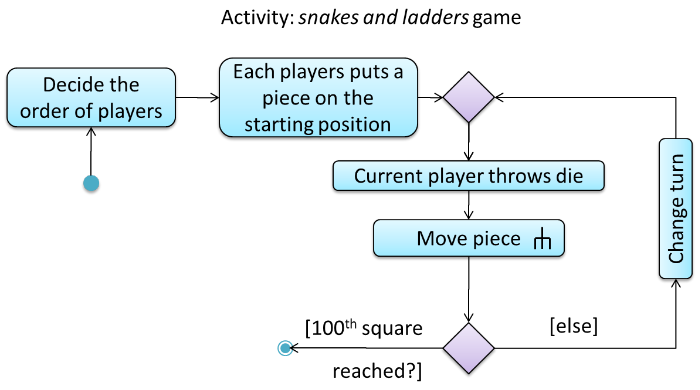
- Both branch nodes and merge nodes are diamond shapes
- Guard conditions must be in square brackets
- Diagram **iv is wrong**: at `x=5`, both guard conditions become true
- Diagram **vi is wrong**: guard conditions should be in square brackets
- All other diagrams are correct

- **Fork node**: indicates the start of concurrent flows of control
- **Join node**: indicates the end of parallel paths
- In a set of parallel paths, execution along all parallel paths should be complete before the execution can start on the outgoing control flow of the join



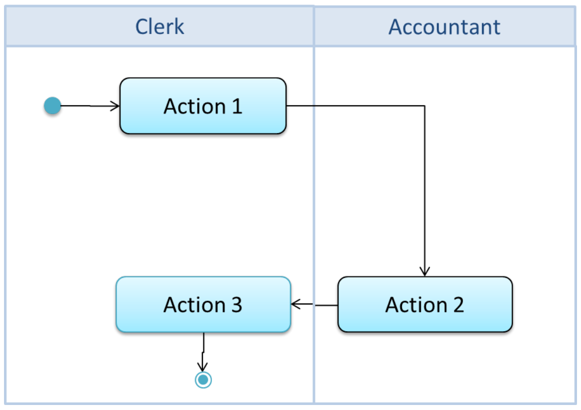
- Diagram **ii is wrong**: all parallel paths that starts from a fork should end in the same join node
- Diagram **iii is wrong**: all parallel paths must end with a join node
- All other diagrams are correct

- **Rake notation**: used to indicate that a part of the activity is given as a separate diagram



Activity: *snakes and ladders* game
Decide the order of players
Each players puts a piece on the starting position
Current player throws die
Move piece
Change turn
[100th square reached?]
[else]

- **Swimlane diagram**: when an AD is partitioned to show who is doing which action



Week 10

- **Design Pattern**: An elegant reusable solution to a commonly recurring problem within a given context in software design (popularised by the book *Design Patterns: Elements of Reusable Object-Oriented Software* by "Gang of Four")
  - **Context**: the situation or scenario where the design problem is encountered
  - **Problem**: main difficulty to be resolved
  - **Solution**: The core of the solution, only includes most general details, which may need further refinement for a specific context
  - **Anti-patterns** (optional): commonly used solutions, which are usually incorrect or inferior to the design pattern
  - **Consequences** (optional): identifying pros and cons of applying the pattern
  - **Other useful information** (optional): code examples, known uses, other related patterns, etc
- **Defensive programming**: code under the assumption "if we leave room for things to go wrong, they will go wrong"
  - Not necessary to be 100% defensive all the time, code may be more complicated and slower to run, depends on factors like:
    - How critical is the system?
    - Will the code be used by programmers other than the author?
    - The level of programming language support for defensive programming

        - The overhead of being defensive
- **Quality Assurance**: the process of ensuring that the software being built has the required levels of quality
    - **Quality Assurance = Validation + Verification**
        - **Validation**: are we building the right system i.e., are the requirements correct?
        - **Verification**: are we building the system right i.e., are the requirements implemented correctly?
        - It is **not** important to clearly distinguish between validation and verification; important to do both
        - A system crash is more likely to be a verification failure than a validation failure
    - Except for trivial SUTs, exhaustive testing is not practical
    - Every test case adds to the cost of testing
        - Test cases need to be designed to make the best use of testing resources
        - Testing should be effective (finds high percentage of existing bugs)
        - Testing should be efficient (high rate of success = bugs_found/test_cases)
        - Each new test should be targeting a potential fault that is not already targeted by existing test cases
    - **Positive test case**: designed to produce an expected/valid behavior
    - **Negative test case**: designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message
    - Test case design can be of 3 types:
        1. **Black-box** (aka specification-based or responsibility-based) approach: test cases are designed exclusively based on the SUT's specified external behavior
        2. **White-box** (aka glass-box or structured or implementation-based) approach: test cases are designed based on what is known about the SUT's implementation, i.e. the code
        3. **Gray-box approach**: test case design uses some important information about the implementation. For example, if the implementation of a sort operation uses different algorithms to sort lists shorter than 1000 items and lists longer than 1000 items, more meaningful test cases can then be added to verify the correctness of both algorithms
- **Testing based on Use Cases**: Use cases can be used for system testing and acceptance testing
    - Note that use cases do not specify the exact data entered into the system
    - Tester has to choose data by considering equivalence partitions and boundary values
    - High-priority use cases are given more attention
    - A scripted approach can be used to test high priority test cases, while an exploratory approach is used to test other areas of concern that could emerge during testing
- **Equivalence Partitioning (EP, aka equivalence class)**: A group of test inputs that are likely to be processed by the SUT in the same way
    - Most SUTs do not treat each input in a unique way, they process all possible inputs in a small number of distinct ways
    - A test case design technique that uses the above observation to improve the E&E of testing
    - Avoids testing too many inputs from one partition (>1 test case from each partition can still be tested)
    - Ensures all partitions are tested
    - Can help make tests more efficient and effective
    - Usually derived from the specifications of the SUT
    - State of target object should be considered
    - Merely a heuristic, applying EP under a glass-box or gray-box approach can yield more precise partitions
    - When deciding EPs of OOP methods, we need to identify EPs of all data participants that can influence the behaviour of the method

        - The target object of the method call
        - Input parameters of the method call
        - Other data/objects accessed by the method such as global variables (NA if black box)
- **Boundary Value Analysis (BVA)**: test case design heuristic based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions
    - When picking test inputs from an equivalence partition, testing boundary values/corner cases are more likely to find bugs
    - Choose three values around the boundary to test: *from*, *below*, and *above* boundary
        - There is NO hard limit like "one boundary value per partition"
    - It is possible that a SUT does not have clear boundary values
        - eg. prime numbers: there are infinitely many

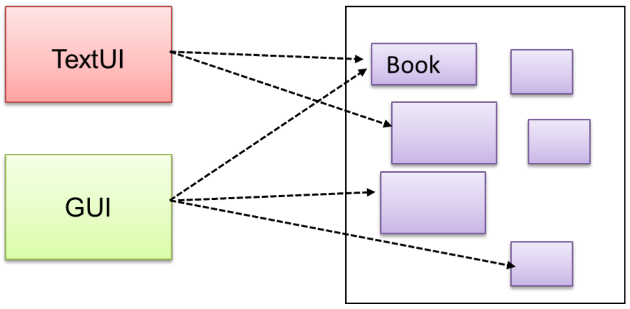
Singleton Pattern

- Context: certain classes should have no more than one instance (singletons)
- Problem: normal class can be instantiated multiple times by invoking constructor
- Solution: make constructor of the singleton class `private` and provide a `public static` class-level method to access the single `private static` instance
- Pros:
    - Easy to apply
    - Effective in achieving its goal with minimal extra work
    - Provides an easy way to access the singleton object from anywhere in the code base
- Cons:
    - The singleton object acts like a global variable that increases coupling across the code base
    - In testing, it is difficult to replace Singleton objects with stubs (static methods cannot be overridden)
    - In testing, singleton objects carry data from one test to another even when we want each test to be independent of the others
- Recommended to apply when, in addition to requiring only one instance of a class, there is a risk of creating multiple objects by mistake, and creating such multiple objects has real negative consequences

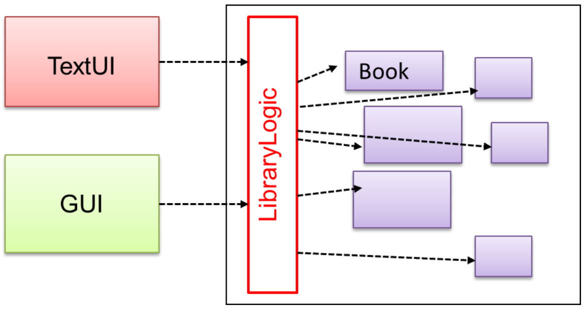
```
class Logic {
    private static Logic theOne = null;

    private Logic() {
        ...
    }

    public static Logic getInstance() {
        if (theOne == null) {
            theOne = new Logic();
        }
        return theOne;
    }
}
```

Facade Pattern

- Context: Components need to access functionality deep inside other components

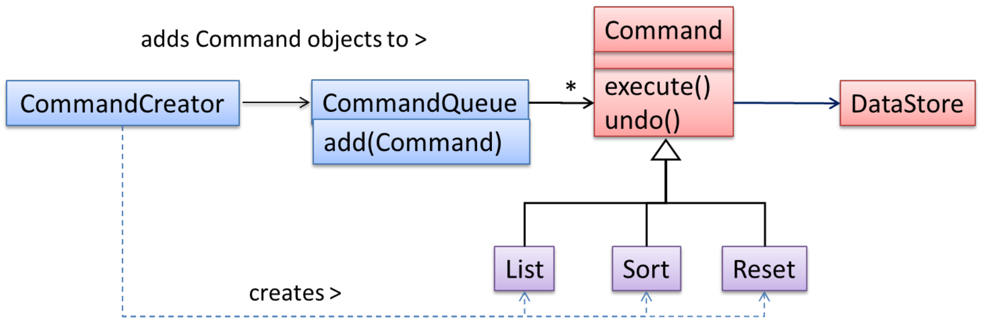


- Problem: Access to the component should be allowed without exposing its internal details
  - Eg. the `UI` component should access the functionality of the `Logic` component without knowing that it contains a `Book` class within it
- Solution: Include a Façade class that sits between the component internals and users of the component such that all access to the component happens through the Facade class
  - Eg. `LibraryLogic` is the Facade class



Command Pattern

- Context: A system is required to execute a number of commands, each doing a different task.
  - Eg. a system might have to support `Sort` , `List` , `Reset` commands
- Problem: It is preferable that some part of the code executes these commands without having to know each command type
  - e.g., there can be a `CommandQueue` object that is responsible for queuing commands and executing them without knowledge of what each command does
- Solution: The essential element of this pattern is to have a general `<<Command>>` object that can be passed around, stored, executed, etc without knowing the type of command (i.e. via polymorphism)

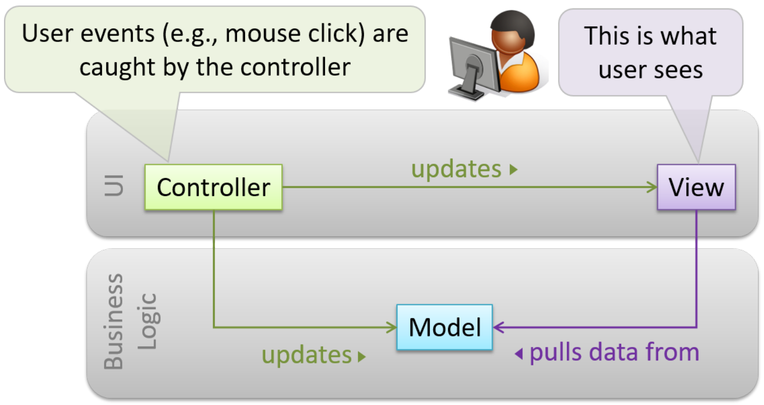


- In the example solution above, the `CommandCreator` creates `List` , `Sort` , and `Reset` `Command` objects and adds them to the `CommandQueue` object. The `CommandQueue` object treats them all as `Command` objects and performs the execute/undo operation on each of them without knowledge of the specific `Command` type. When executed, each

`Command` object will access the `DataStore` object to carry out its task. The `Command` class can also be an abstract class or an interface

MVC Pattern

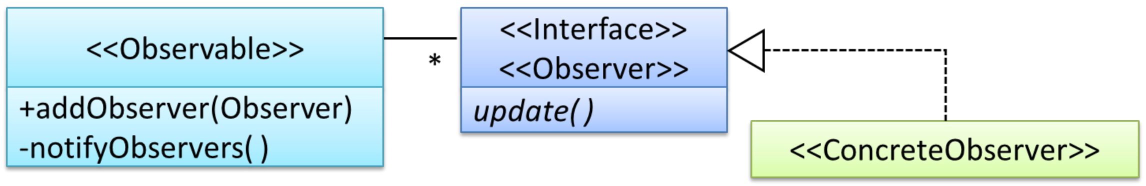
- Context: most applications support storage/retrieval of information, displaying of information to the user (often via multiple UIs having different formats), and changing stored information based on external inputs
- Problem: high coupling that can result from the interlinked nature of the features described above
- Solution: decouple data, presentation, and control logic of an application by separating them into three different components:
    - Model: stores and maintains data, updates views if necessary
    - View: displays data, interacts with the user, and pulls data from the model if necessary
    - Controller: detects UI events such as mouse clicks, button pushes and takes follow up action. Updates/changes the model/view when necessary



- Typically, the UI is the combination of view and controller
- In a simple UI where there's only one view, Controller and View can be combined as one class

Observer Pattern

- Context: An object (possibly, more than one) is interested to get notified when a change happens to another object. That is, some objects want to 'observe' another object
- Problem: The 'observed' object does not want to be coupled to objects that are 'observing' it
- Solution: Force the communication through an interface known to both parties



- `<<Observer>>` is an interface: any class that implements it can observe an `<<Observable>>`. Any number of `<<Observer>>` objects can observe (i.e. listen to changes of) the `<<Observable>>` object
- The `<<Observable>>` maintains a list of `<<Observer>>` objects. addObserver(Observer) operation adds a new `<<Observer>>` to the list of `<<Observer>>`'s
- Whenever there is a change in the `<<Observable>>`, the `notifyObservers()` operation is called that will call the `update()` operation of all `<<Observer>>`'s in the list

Week 11

- **Architectural Styles**: software architectures that follow various high-level styles
    - **N-tier/multi-layered/layered style**: higher layers make use of services provided by lower layers, lower layers are independent of higher layers
        - Used in operating systems and network communication software
    - **Client-server style**: has at least one component playing the role of a server and at least one client component accessing the services of the server
        - Used often in distributed applications
    - **Event-driven style**: controls the flow of the application by detecting events from event emitters and communicating those events to interested event consumers
        - Used often in GUIs
    - **Transaction processing style**: divides the workload of the system down to a number of transactions which are then given to a *dispatcher* that controls the execution of each transaction
        - Task queuing, ordering, undo etc. are handled by the dispatcher
    - **Service-oriented architecture (SOA) style**: builds applications by combining functionalities packaged as programmatically accessible services
        - Aims to achieve interoperability between distributed services, which may not even be implemented using the same programming language
        - A common way to implement SOA is through the use of XML web services where the web is used as the medium for the services to interact, and XML is used as the language of communication between service providers and service users
    - Most applications use a mix of these architectural styles
- **Architecture diagrams**: have no standard notation (ie. UML is not the standard), but try to follow these basic guidelines
    - Minimise the variety of symbols and explain them if they do not have widely-understoof meanings
    - Avoid the indiscriminate use of double-headed arrows to show interactions between components
- **Test Input Combination Strategies**
    - **All combinations**: generates test cases for each unique combination of test inputs
        - Effective but not efficient
    - **At least once**: includes each test input at least once
    - **All pairs**: creates test cases so that for any given pair of inputs, all combinations between them are tested
        - Based on the observations that a bug is rarely the result of more than two interacting factors
        - Number of test cases is lower than the all combinations strategy, but higher than the at least once approach
    - **Random strategy**: generates test cases using one of the other strategies and then pick a subset randomly
- **Test Input Heuristic**
    - Each valid input must appear at least once in a positive test case (to test that an input is valid, all other inputs must be valid)
    - No more than one invalid input in a test case
- **Formal Verification**: uses mathematical techniques to prove the correctness of a program
    - Can be used to prove the *absence* of errors (unlike testing)
    - Only proves the compliance with the specification, but not the actual utility
    - Requires highly specialized notations and knowledge which makes it an expensive technique to administer

    - Therefore, formal verifications are more commonly used in safety-critical software such as flight control systems
- **Reuse**: robustness of a new software system can be enhanced while reducing the manpower and time requirement
    - Reused code may be an overkill (think using a sledgehammer to crack a nut) increasing the size of, or/and degrading the performance of, your software
    - Reused software may not be mature/stable enough to be used in an important product, software can change drastically and rapidly, possibly in ways that break your software
    - Non-mature software has the risk of dying off as fast as they emerged, leaving you with a dependency that is no longer maintained
    - The license of the reused software (or its dependencies) restrict how you can use/develop your software
    - The reused software might have bugs, missing features, or security vulnerabilities that are important to your product but not so important to the maintainers of that software, which means those flaws will not get fixed as fast as you need them to
    - Malicious code can sneak into your product via compromised dependencies
- **Library**: a collection of modular code that is general and can be used by other programs
- **Application Programming Interface (API)**: specifies the interface through which other programs can interact with a software component
    - eg. GitHub API, API of the Java `String` class
- **Software framework**: reusable implementation of a software (or part thereof) providing generic functionality that can be selectively customised to produce a specific application
    - Some provide a complete implementation of a default behavior which makes them immediately usable (ie. Eclipse)
    - Facilitates the adaptation and customisation of some desired functionality
    - Some cover only a specific components or an aspect
    - eg. Eclipse, JavaFX, Spring, JUnit, Jest
- **Frameworks vs Libraries**
    - Libraries are meant to be used 'as is' while frameworks are meant to be customised/extended
    - Your code calls the library code while the framework code calls your code
    - **Hollywood Principle (aka Inversion of Control)**: used by frameworks, where the coder writes code that will be called by the framework (eg. writing test code that will be called by the JUnit framework)
- **Platform**: provides a **runtime environment** for applications
    - Often also bundled with various libraries, tools, frameworks, and technologies
    - Technically, an OS can be called a platform. For example, Windows PC is a platform for desktop applications while iOS is a platform for mobile apps
    - Two well-known examples of platforms are *JavaEE* and *.NET*, both of which sit above Operating systems layer, and are used to develop enterprise applications