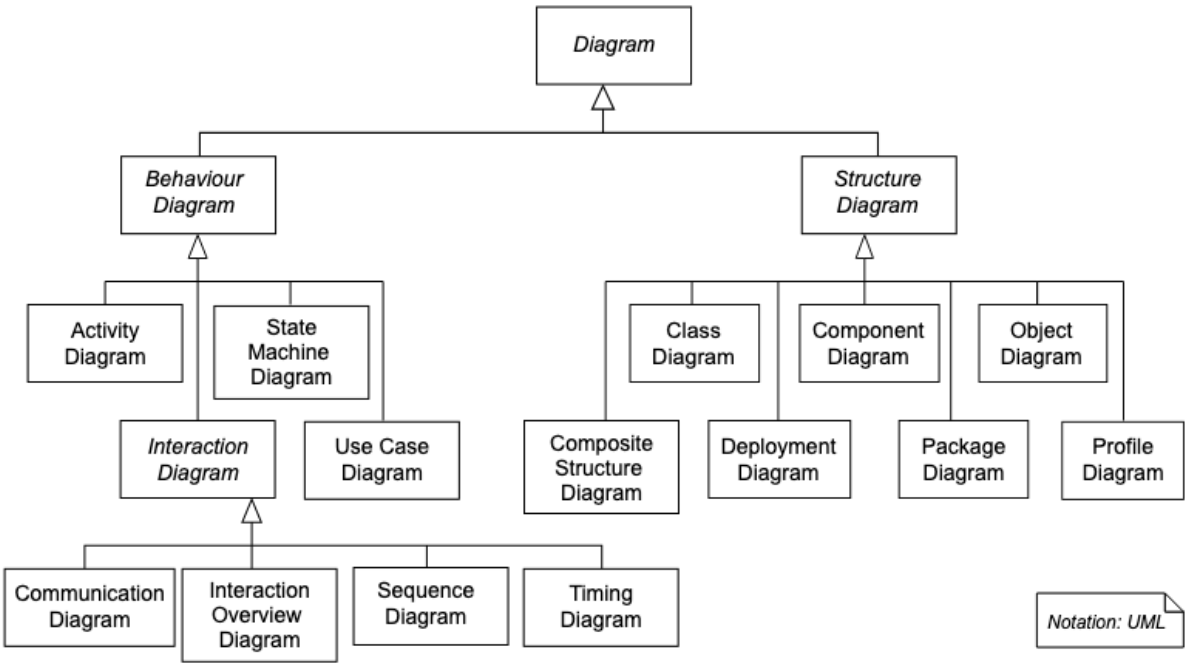
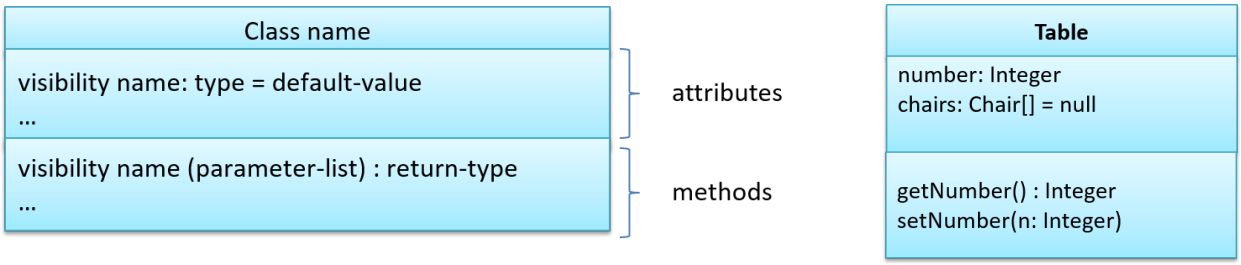
CS2103T Notes

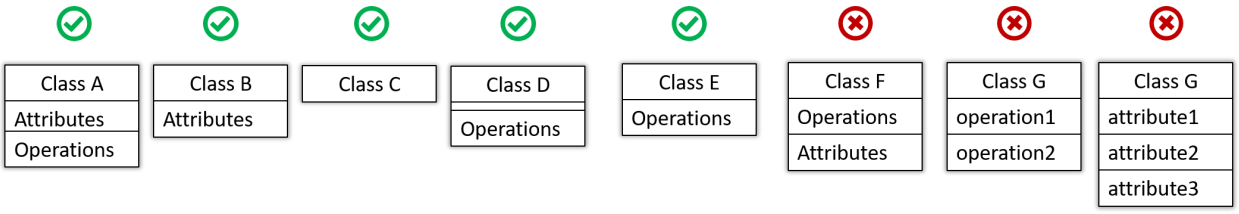
UML



Class Diagrams

- describe the structure (but not the behavior) of an OOP solution

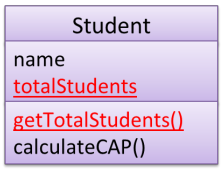




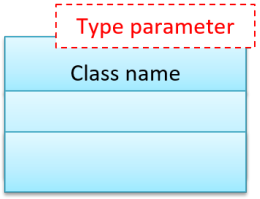
- 'Operations' compartment and/or the 'Attributes' compartment may be omitted (optional)
  - 'Attributes' always appear above the 'Operations' compartment.

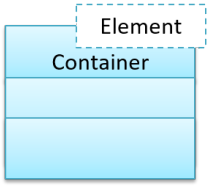
- All operations should be in one compartment rather than each operation in a separate compartment. Same goes for attributes.
- constructors are optionals
- built-in classes are usually shown in the attribute box rather than separate box (e.g. String)

- Class-level attributes and variables (static): underline



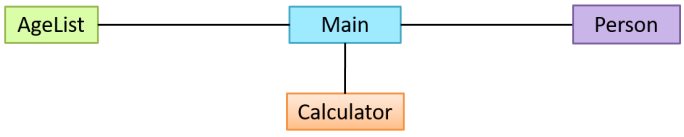
- Visibility: level of access
  - `+`: public
  - `-`: private
  - `#`: protected
  - `~`: package private
- Generic classes:





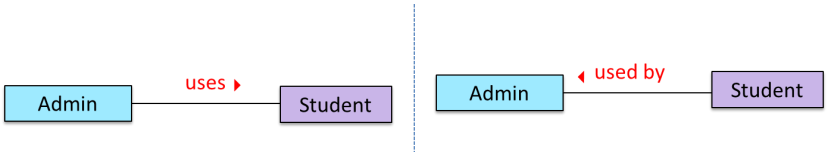
Associations

- main connections among the classes in a class diagram
  - e.g. student and courses are associated
- solid lines to represent association



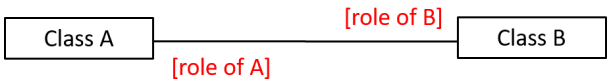
- **Association labels:**

- describe the meaning of the association
- arrow head indicates the direction in which the label is to be read.



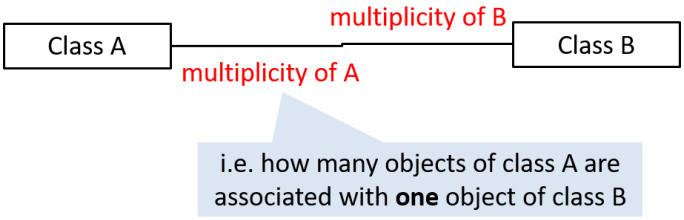
- Diagram on the left: `Admin` class is associated with `Student` class because an `Admin` object *uses* a `Student` object.
- Diagram on the right: `Admin` class is associated with `Student` class because a `Student` object is *used by* an `Admin` object.

- **Association roles:**

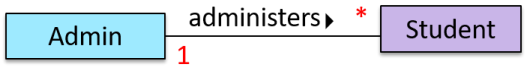


- This association represents a marriage between a `Man` object and a `Woman` object. The respective roles played by objects of these two classes are `husband` and `wife`.

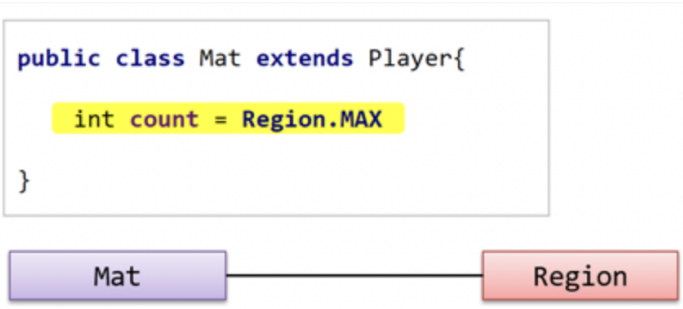
- **Association multiplicity:**



- dictates how many objects take part in each association.
- `0..1` : *optional association*, can be linked to 0 or 1 objects.
- `1` : *compulsory association*, must be linked to one object at all times.
- `*` : can be linked to 0 or more objects.
- `n..m` : the number of linked objects must be within `n` to `m` inclusive.
- bidirectional associations: require matching variables in both classes.
- other multiplicities require suitable data structure e.g. `Array`

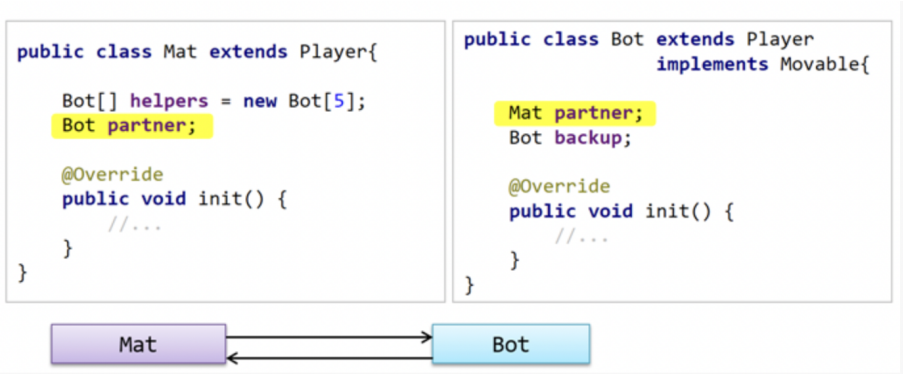


- an `Admin` object administers (is in charge of) any number of students but a `Student` object must always be under the charge of exactly one `Admin` object.



incorrect: not association

- **Navigability:**
  - The concept of which object in the association knows about the other object
  - **Unidirectional**: If the navigability is from `Box` to `Rope`, `b` will have a reference to `r` but `r` will not have a reference to `b`. Similarly, if the navigability is in the other direction, `r` will have a reference to `b` but `b` will not have a reference to `r`.
  - **Bidirectional**: `b` will have a reference to `r` and `r` will have a reference to `b` i.e., the two objects will be pointing to each other.

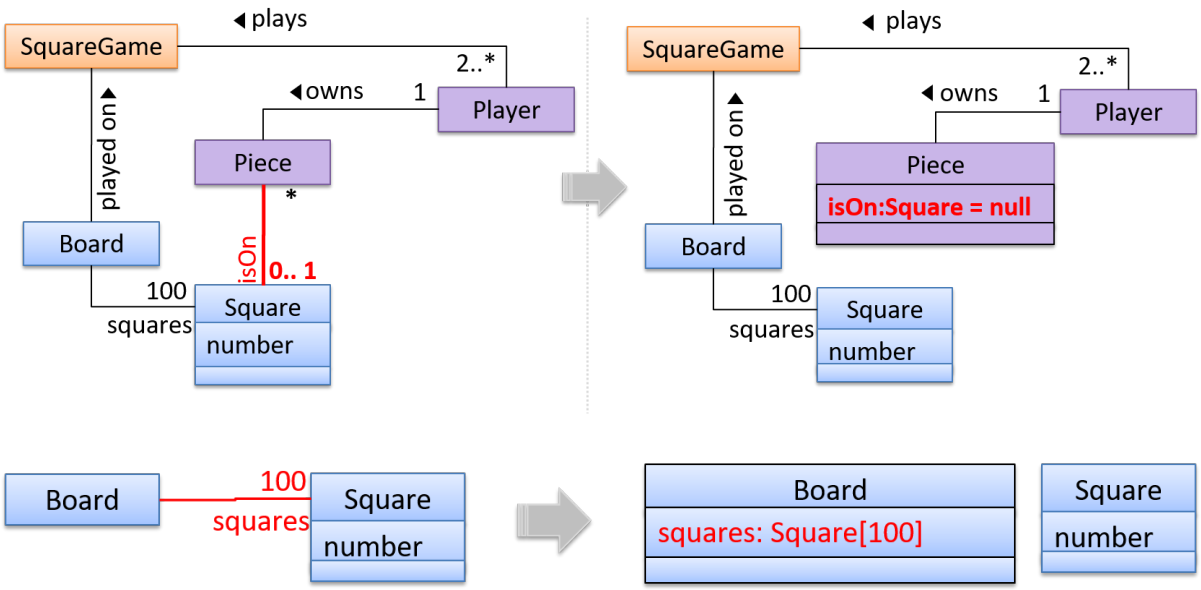


incorrect representation: Although there are two variables, it is a single association. Each variable keeps track of the the same associational (i.e., bi-directional)

```java
public class Mat extends Player{

    Bot[] helpers = new Bot[5];
    Bot partner;

    @Override
    public void init() {
        //...
    }
}
```

```java
public class Bot extends Player
                 implements Movable{

    Mat partner;
    Bot backup;

    @Override
    public void init() {
        //...
    }
}
```

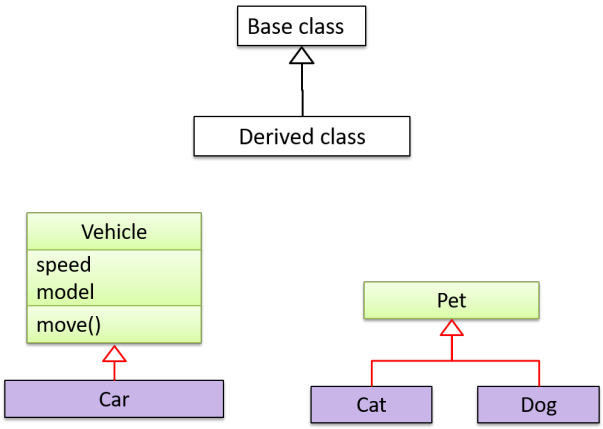
- An association can be shown as an attribute instead of a line, using the following notation:
  - `name: type [multiplicity] = default value`
  - A `Piece` may or may not be on a `Square`. Note how that association can be replaced by an `isOn` attribute of the `Piece` class. The `isOn` attribute can either be `null` or hold a reference to a `Square` object, matching the `0..1` multiplicity of the association it replaces. The default value is `null`.



- Show each association as either an attribute or a line but not both. A line is preferred as it is easier to spot.

Inheritance

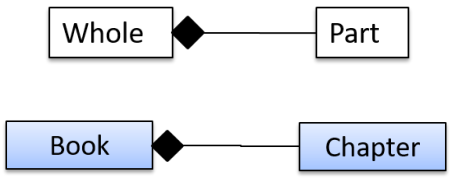
- use a triangle and a solid line



- The `Car` class inherits from the `Vehicle` class.
- The `Cat` and `Dog` classes inherit from the `Pet` class.
- No need show all parent methods in child class, only show the overridden methods

Composition

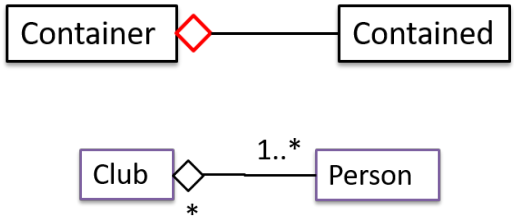
- Whole-Part relationship
- solid diamond symbol



- A `Book` consists of `Chapter` objects. When the `Book` object is destroyed, its `Chapter` objects are destroyed too.

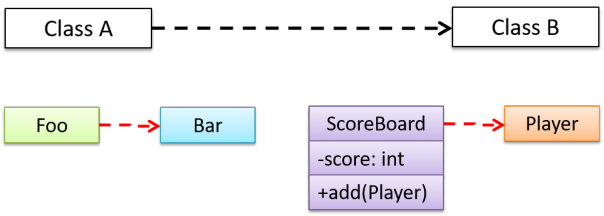
Aggregation

- Container-Containee relationship
- hollow diamond symbol



Dependency

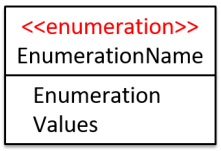
- dashed arrow

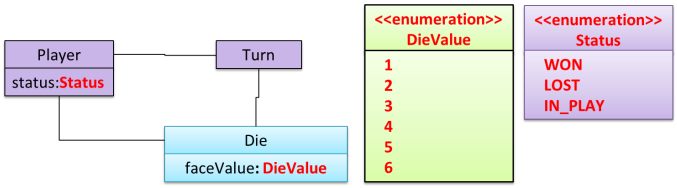


Dependencies vs Associations

- An association is a relationship resulting from one object keeping a reference to another object (i.e., storing an object in an instance variable).
  - While such a relationship forms a *dependency*, we need not show that as a dependency arrow in the class diagram if the association is already indicated in the diagram.
  - That is, showing a dependency arrow does not add any value to the diagram.
  - Similarly, an inheritance results in a dependency from the child class to the parent class but we don't show it as a dependency arrow either, for the same reason as above.
- **Use a dependency arrow to indicate a dependency only if that dependency is not already captured by the diagram in another way**
  - e.g., class `Foo` accessing a constant in `Bar` but there is no association/inheritance from `Foo` to `Bar`.

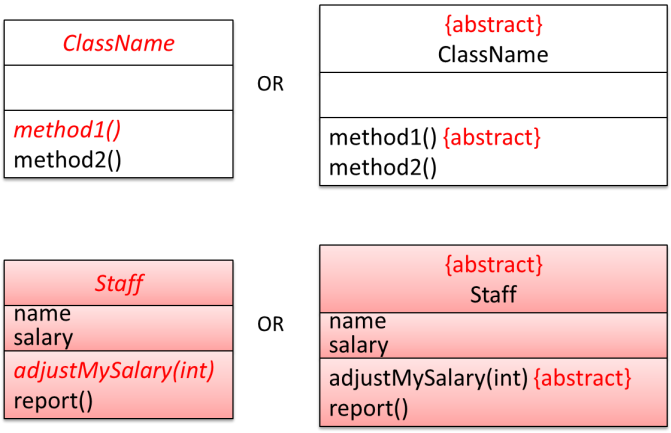
Enumeration





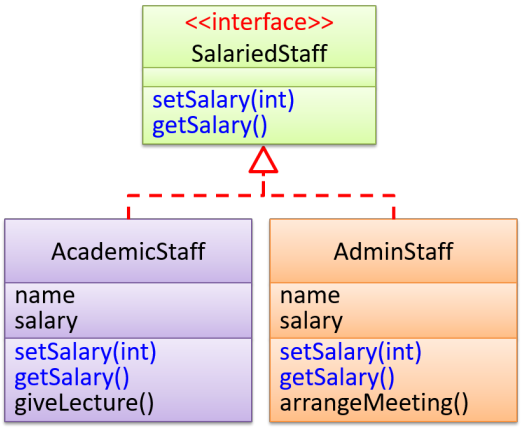
Abstract Classes

- use *italics* or `{abstract}` (preferred) keyword to denote abstract classes/methods.



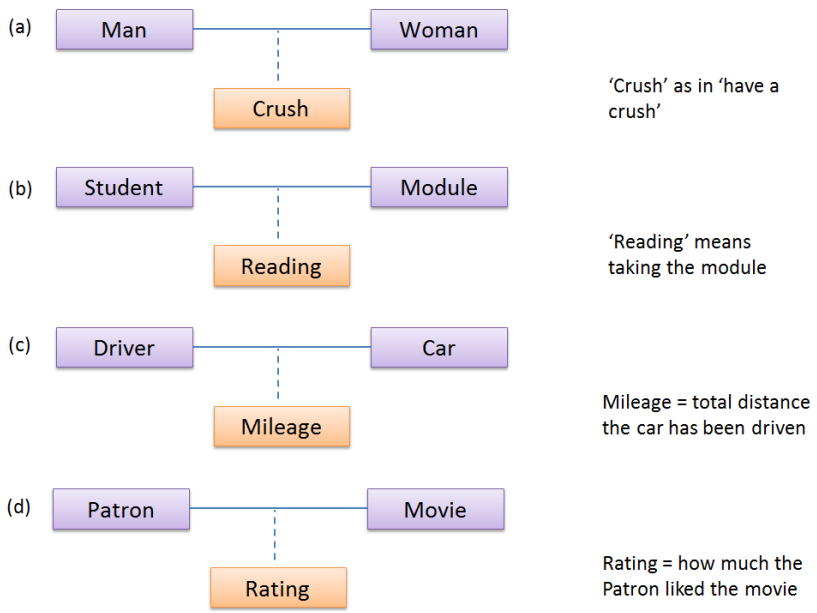
Interface

- additional keyword `<<interface>>`
- dashed line
- triangle can be solid or empty
- methods are `public` by default: `+`



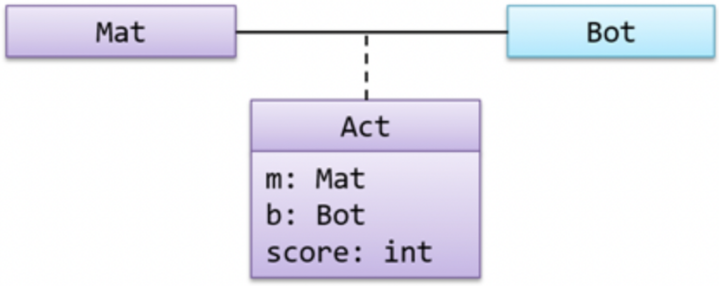
Association Class

- additional information about an association.
- A `Man` class and a `Woman` class are linked with a 'married to' association and there is a need to store the date of marriage.
  - However, that data is related to the association rather than specifically owned by either the `Man` object or the `Woman` object.
  - an additional association class can be introduced, e.g. a `Marriage` class, to store such information.



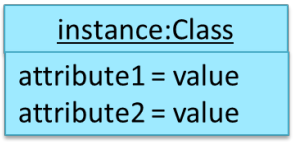
(c) is not an appropriate association class: mileage is a property of car

```
public class Act {
    Mat m;
    Bot b;
    int score;
    //...
}
```



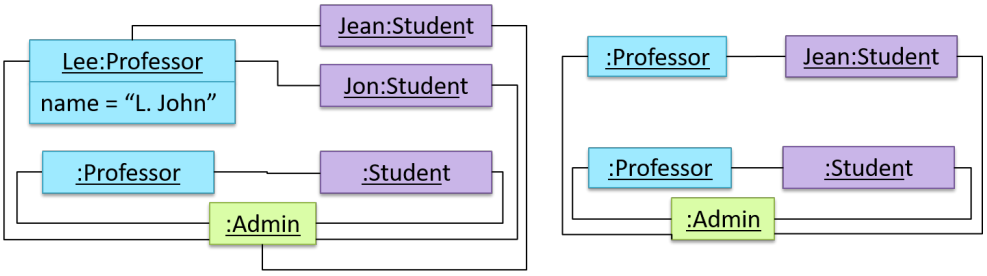
incorrect: Mat and Bot are already represented in the diagram as a line

Object Diagram



- The class name and object name e.g. `car1:Car` are underlined.
- `objectName:ClassName` is meant to say 'an instance of `ClassName` identified as `objectName`.
- Unlike classes, there is no compartment for methods.
- Attributes and object names are optional
  - e.g. `:Car` which is meant to say 'an *unnamed* instance of a Car object'.

Differences between class and object diagrams

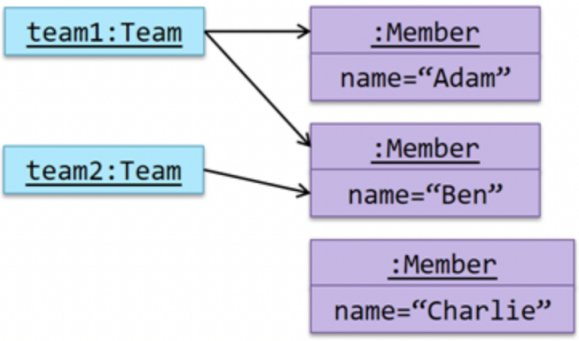


Object diagrams:

- Show objects instead of classes:
  - Instance name may be shown
  - There is a `:` before the class name
  - Instance and class names are underlined
  - dependencies are NOT shown
- Methods are omitted
- Multiplicities are omitted
- multiple object diagrams can correspond to a single class diagram.

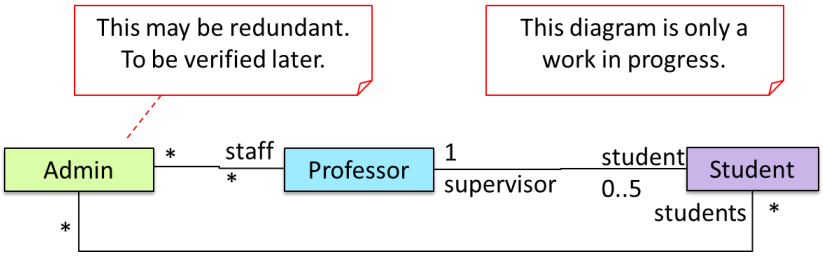
```
Member adam = new Member("Adam");
Member ben = new Member("Ben");
Member charlie = new Member("Charlie");
Team team1 = new Team(new Member[]{adam, ben})
Team team2 = new Team(new Member[]{ben})
```

```
class Team{
    Member[] members;

    Team(Member... members){
        this.members = members;
    }
}

class Member{
    String name;

    Member(String name){
        this.name = name;
    }
}
```



correct rep

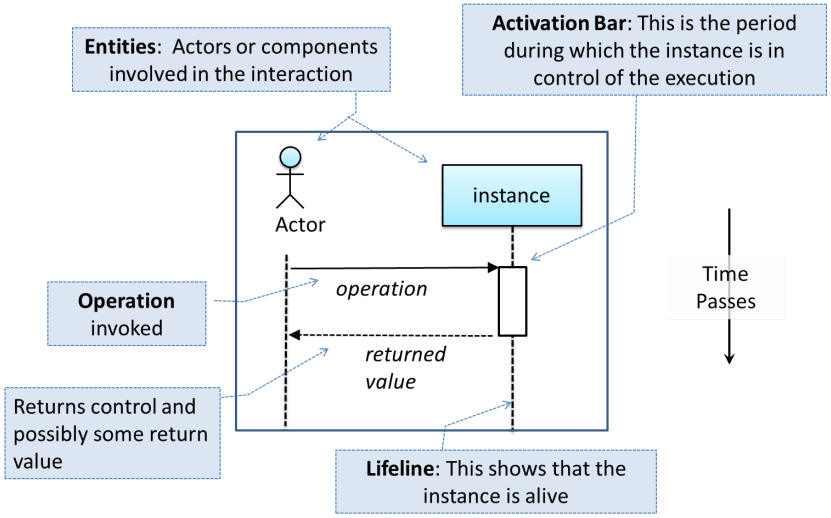
Notes

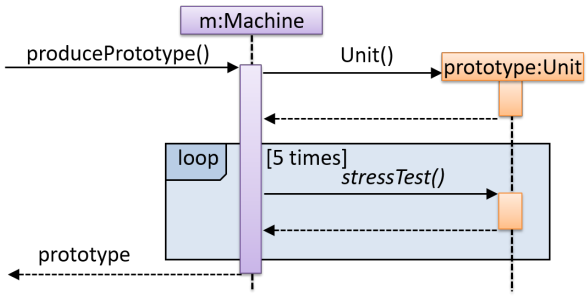


- additional info
- can be shown with or without connection

Sequence Diagram

- model the interactions between various entities in a system, in a specific scenario
- name, activation bars and return arrows are optional
- no underline for name



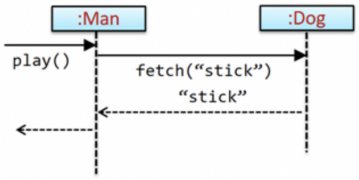


model the interactions for the method call produceProtoype() on a Machine object.

```
class Machine {

    Unit produceProtoype() {
        Unit prototype = new Unit();
        for (int i = 0; i < 5; i++) {
            prototype.stressTest();
        }
        return prototype;
    }
}

class Unit {

    public void stressTest() {

    }
}
```

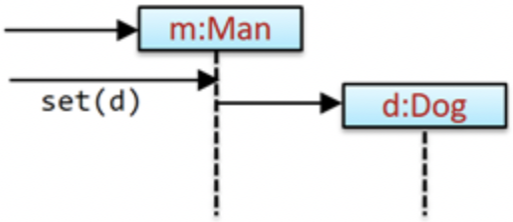
```
m.play(); //m is a Man object
```

```
class Man {
    Dog d;

    //...

    void play() {
        d.fetch("stick");
    }
}

class Dog {

    String fetch(String object) {
        //...
        return object;
    }
}
```



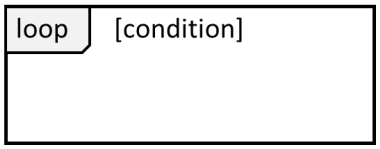
```
Man m = new Man();
m.set(new Dog());
```

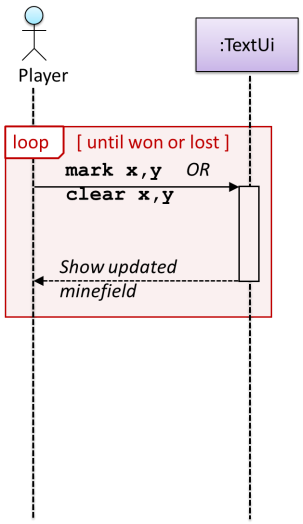
```
class Man {
    Dog d;

    void set(Dog d){
        this.d = d;
    }

    //...
}

class Dog {
    //...
}
```



incorrect: d cannot be used before it is created.

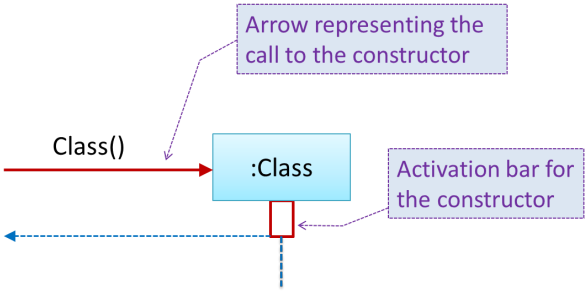
Loops



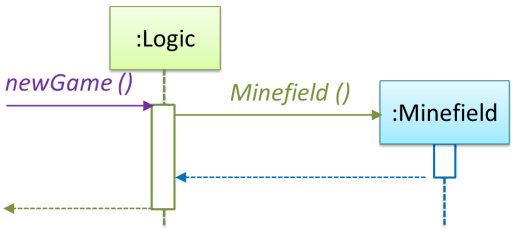


- The Player calls the `mark x, y` command or `clear x y` command repeatedly until the game is won or lost.

Object Creation



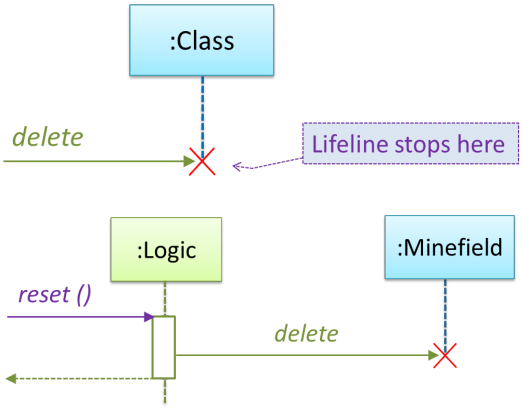
- The arrow that represents the constructor arrives at the side of the box representing the instance.
- The activation bar represents the period the constructor is active.



The Logic object creates a Minefield object.

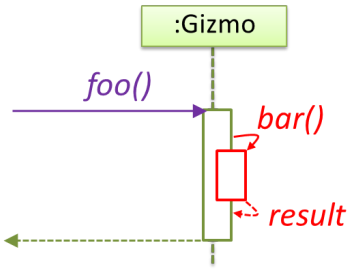
Objection Deletion

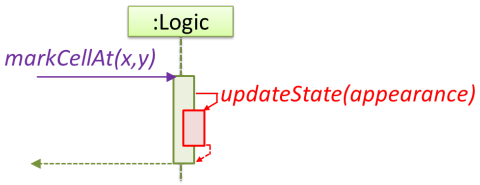
- `X` at the end of the lifeline of an object to show its deletion.



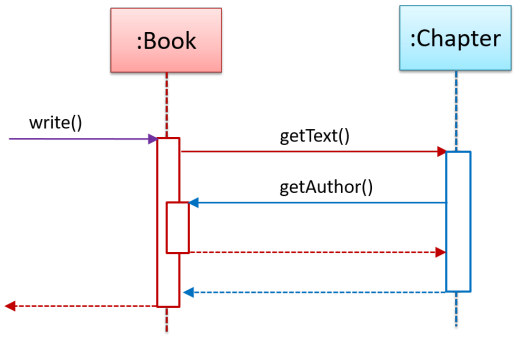
Self-Invocation

- a method of object calling another of its own methods





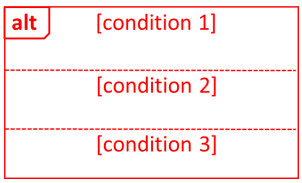
The markCellAt(...) method of a Logic object is calling its own updateState(...) method.

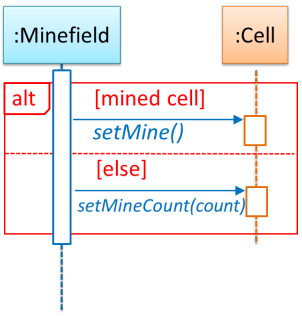


- In this variation, the `Book#write()` method is calling the `Chapter#getText()` method which in turn does a call back by calling the `getAuthor()` method of the calling object.

Alternative Paths

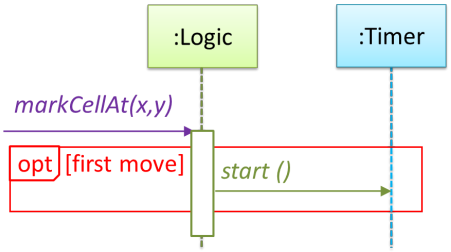
- ≤ 1 alternative partition can be executed





- `Minefield` calls the `Cell#setMine` method if the cell is supposed to be a mined cell, and calls the `Cell:setMineCount(...)` method otherwise.

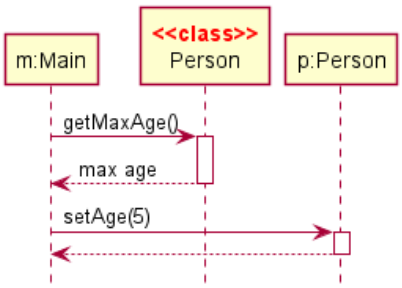
Optional Paths

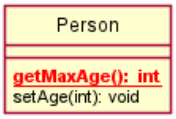


- `Logic#markCellAt(...)` calls `Timer#start()` only if it is the first move of the player.

Calls to Static Methods

- use `<<class>>` to show that a participant is the class itself.

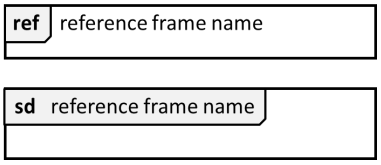


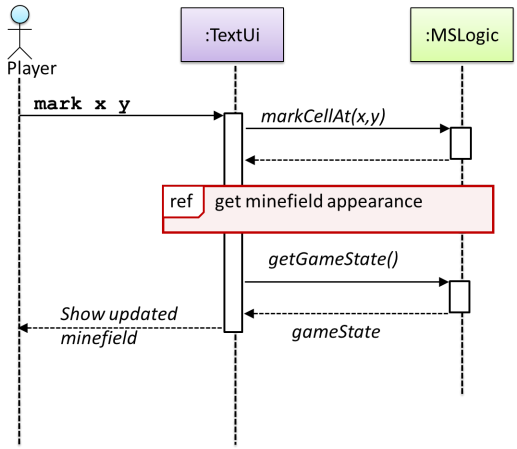


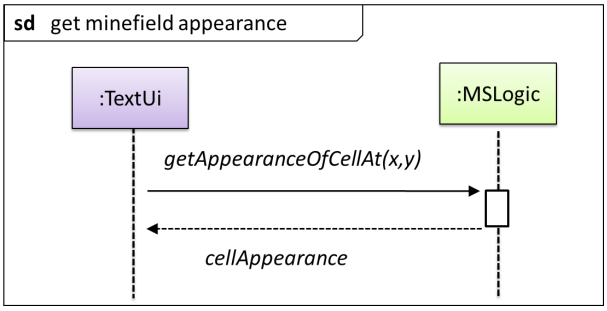
- `m` calls the static method `Person.getMaxAge()` and also the `setAge()` method of a `Person` object `p`.

Reference Frames

- allow a segment of the interaction to be omitted and shown as a separate sequence diagram

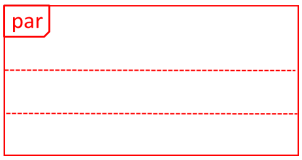


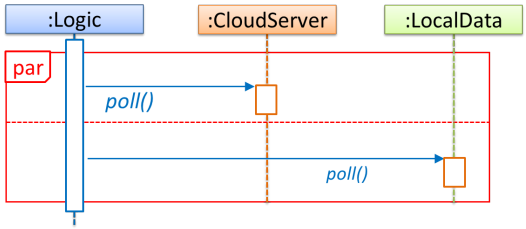




Those details are shown in a separate sequence diagram given above.

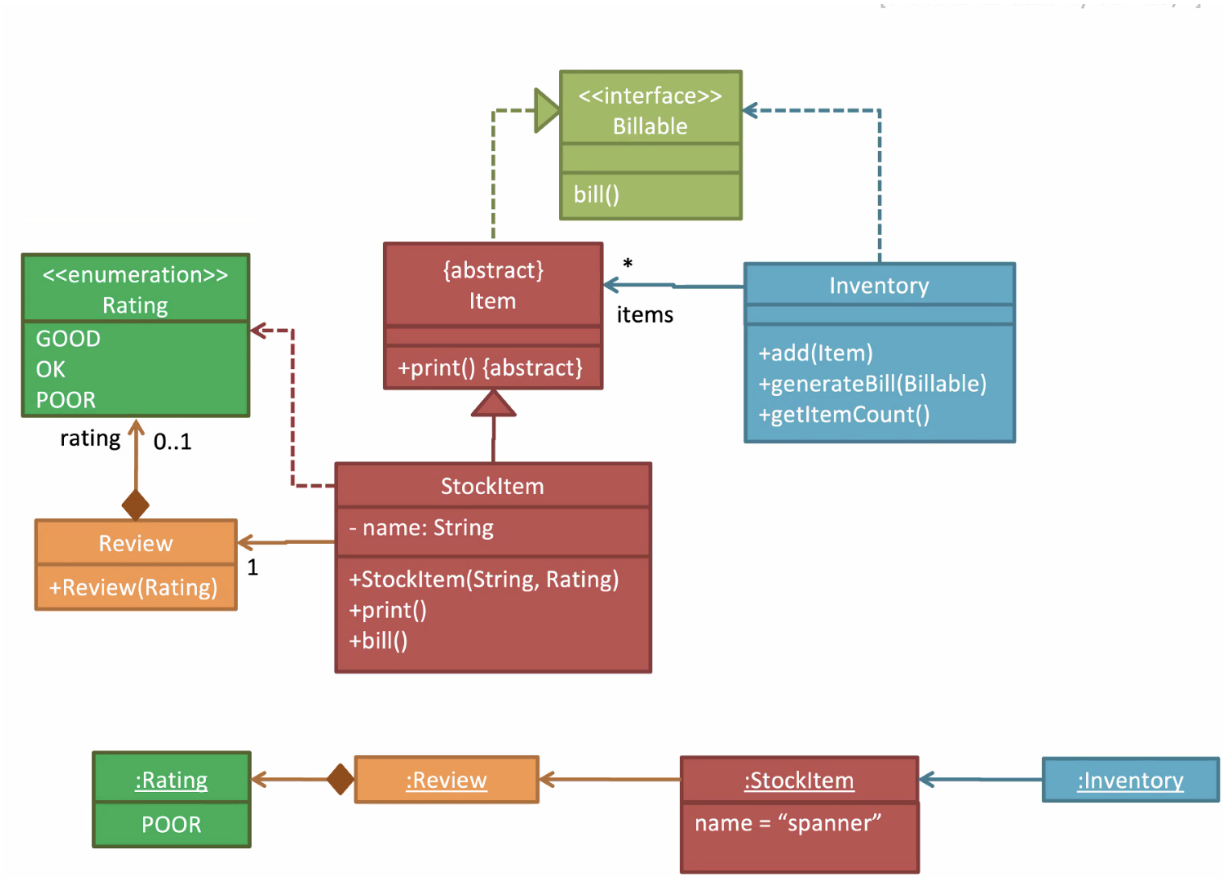
Parallel Paths





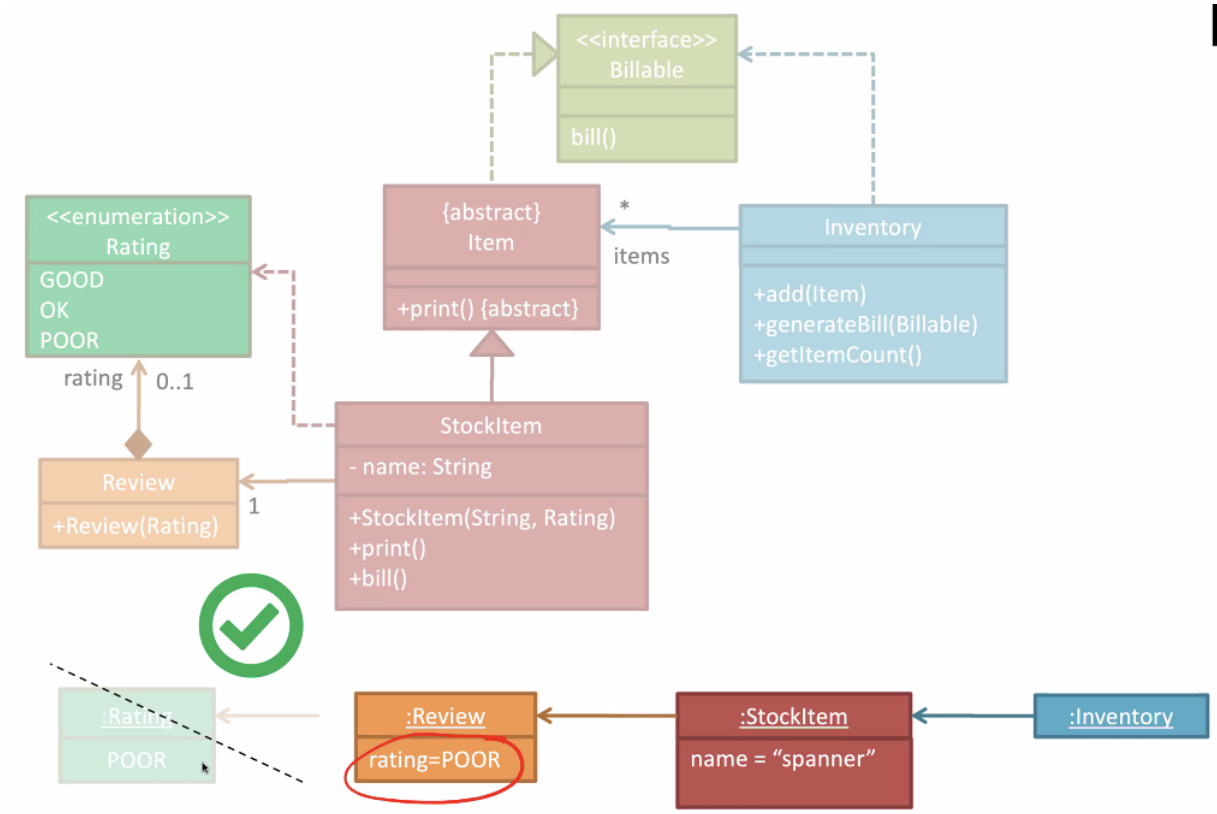
Logic is calling methods CloudServer#poll() and LocalServer#poll() in parallel.

W8 Tutorial



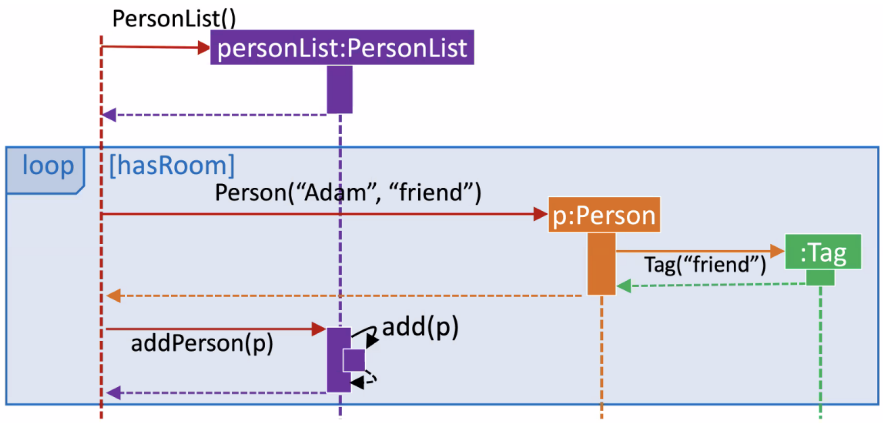
composition is not compulsory

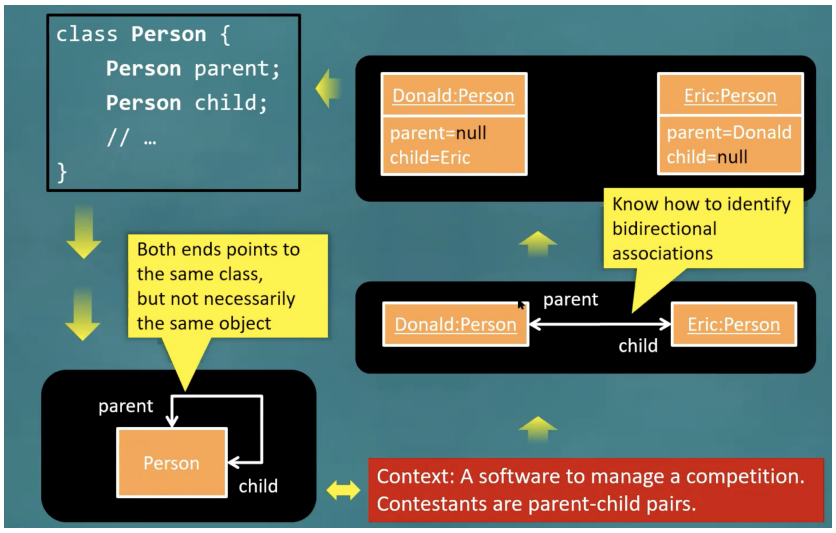
19

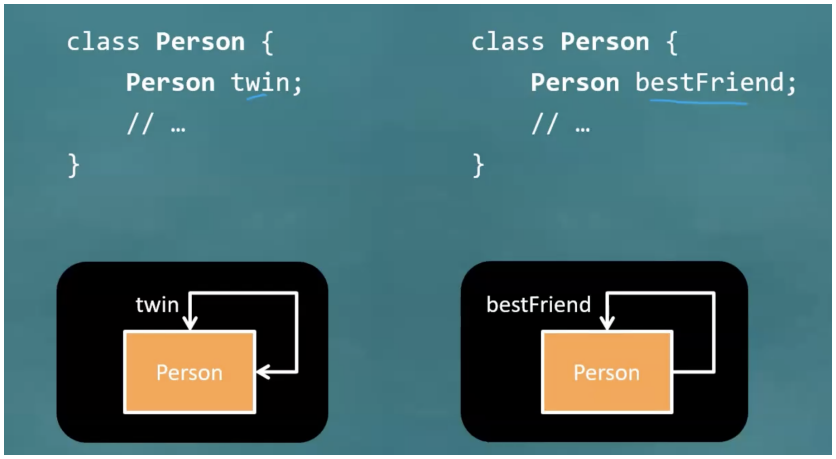


(b) Draw an object diagram to represent the situation where the inventory has one item with a name `spanner` and a review of `POOR` rating

i.e. `new Inventory().add(new StockItem("spanner", Rating.POOR))`

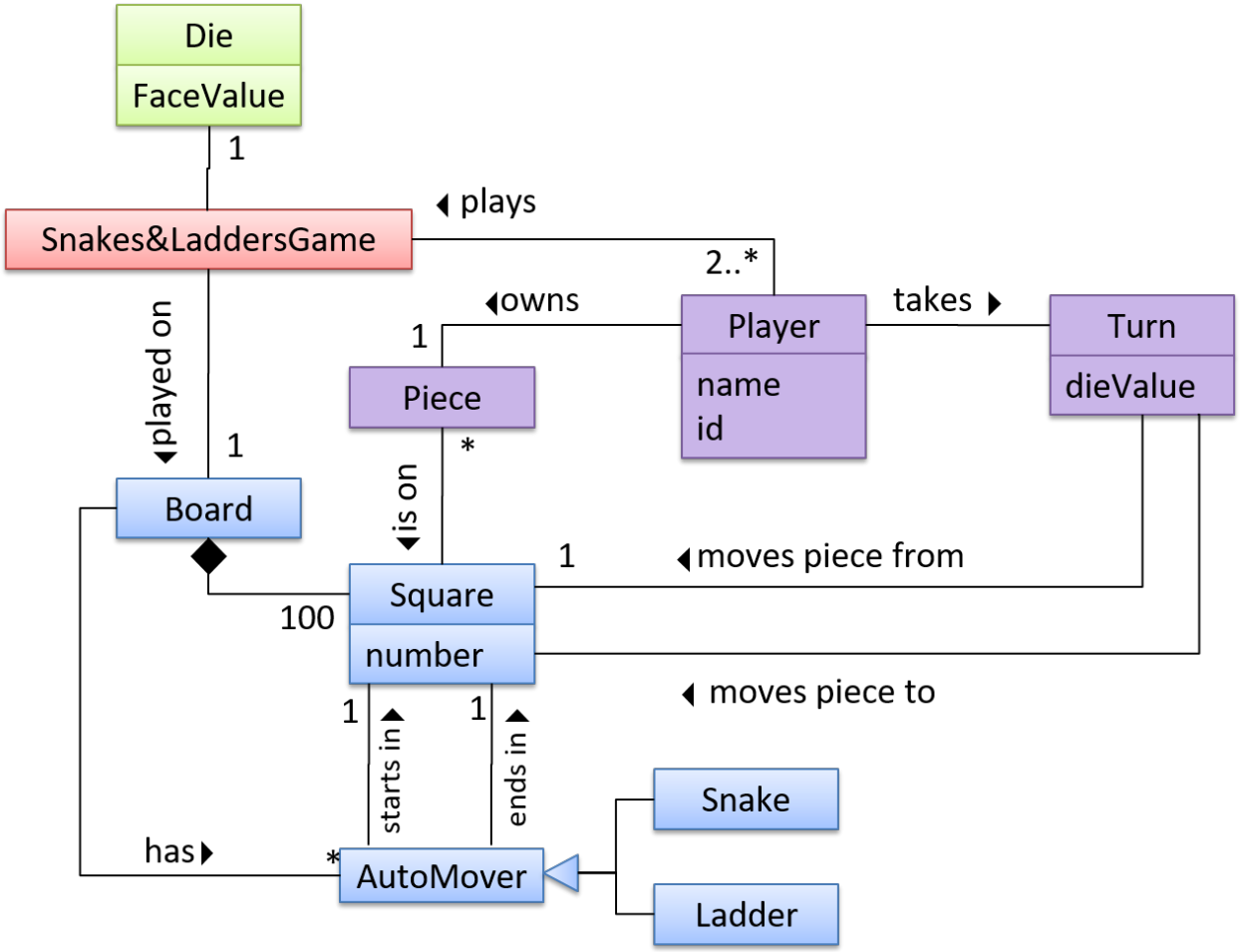






Object-Oriented Domain Models (OODMs) / Conceptual Class Diagrams

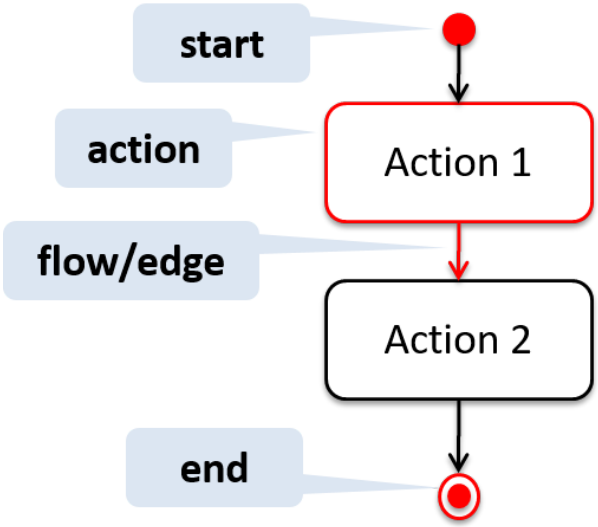
- OODM model objects in the problem domain, class diagram is about solution domain
- OODMs do not contain solution-specific classes
  - e.g. a class called `DatabaseConnection` could appear in a class diagram but not usually in an OODM because `DatabaseConnection` is something related to a software solution but not an entity in the problem domain.
- OODMs represents the class structure of the problem domain and not their behaviour
  - behaviours shown with sequence diagrams
- **DO NOT** have methods and navigability, unlike class diagrams



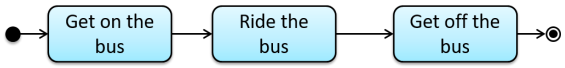
The snakes and ladders game is played by two or more players using a board and a die. The board has 100 squares marked 1 to 100. Each player owns one piece. Players take turns to throw the die and advance their piece by the number of squares they earned from the die throw. The board has a number of snakes. If a player's piece lands on a square with a snake head, the piece is automatically moved to the square containing the snake's tail. Similarly, a piece can automatically move from a ladder foot to the ladder top. The player whose piece is the first to reach the 100th square wins.

Activity Diagrams (AD)

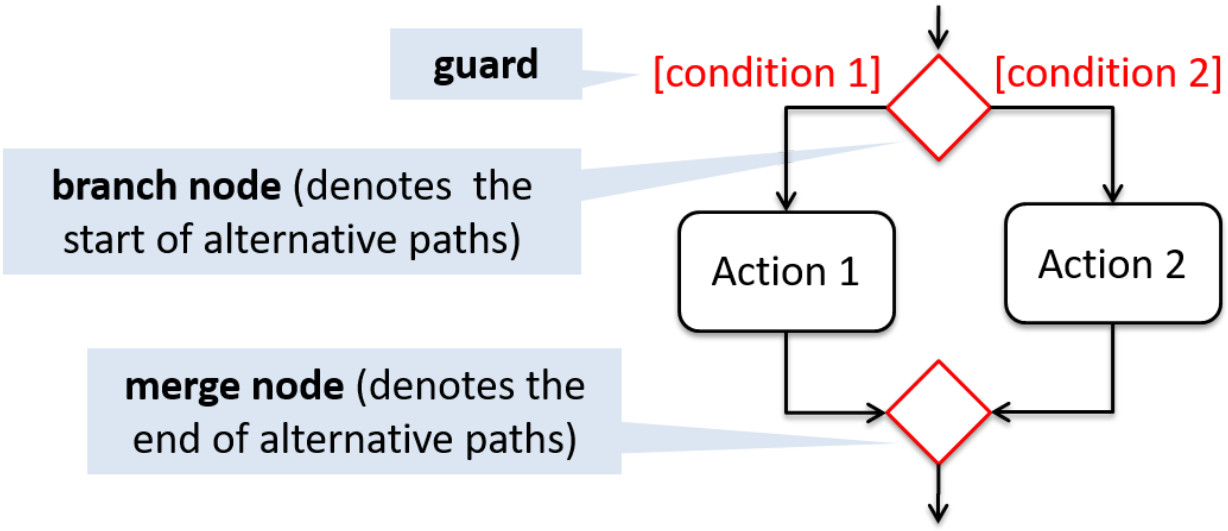
- model workflows
  - >1 endpoints are ok
- captures an activity through the actions and control flows that make up the activity.
  - An action is a single step in an activity → shown as a rectangle with rounded corners.
  - A control flow shows the flow of control from one action to the next → shown by drawing a line with an arrow-head to show the direction of the flow.
    - cannot have double-headed arrows



Activity: A passenger rides on a bus

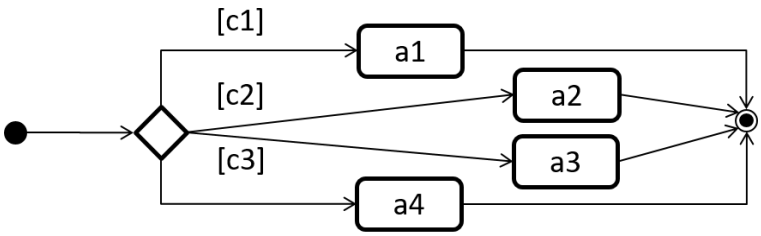


Alternate Paths

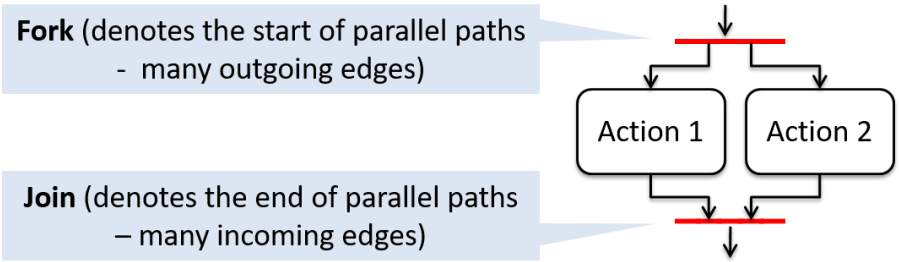


Both branch nodes and merge nodes are diamond shapes. Guard conditions must be in square brackets.

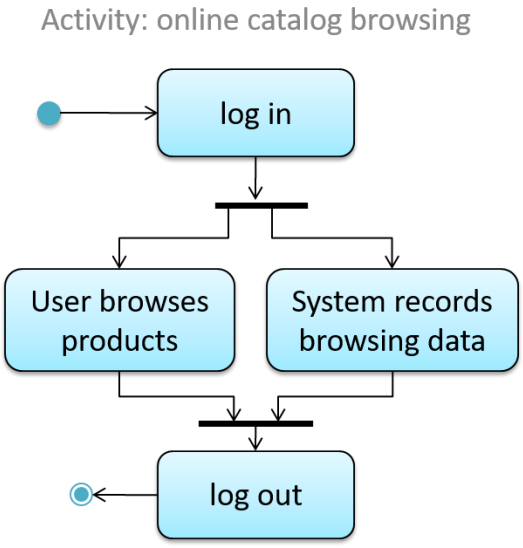
- Acceptable modifications
  - merge node is optional (if does not cause ambiguities)
  - `Else` condition is optional
  - Multiple arrows can starting from the same corner of a branch node.



Parallel Paths

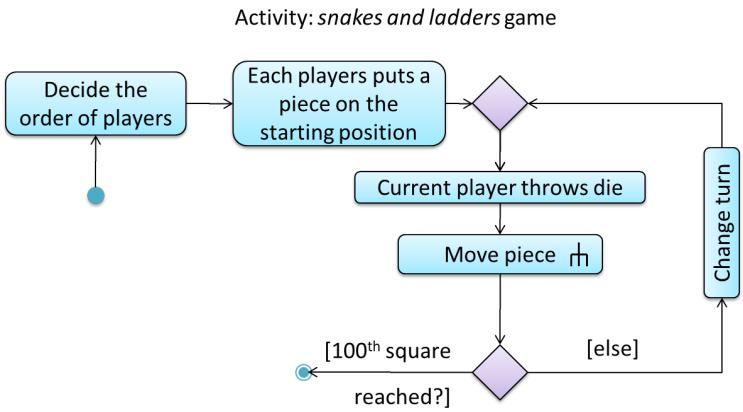


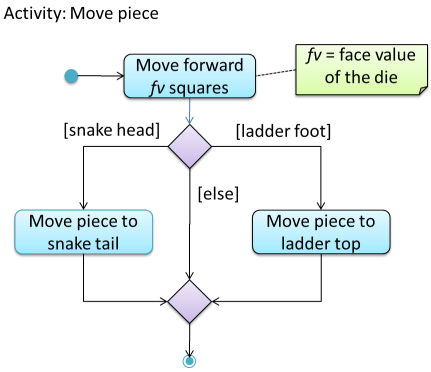
- execution along all parallel paths should be complete before the execution can start on the outgoing control flow of the join.



Rake

- indicate that a part of the activity is given as a separate diagram.

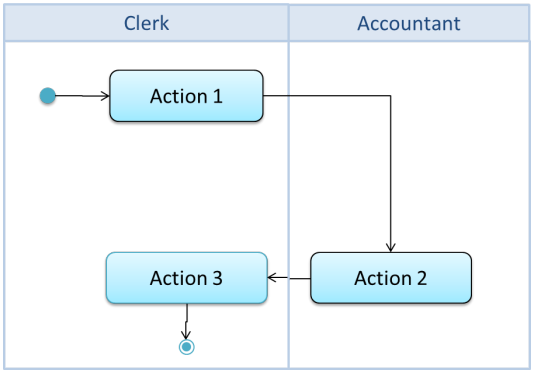


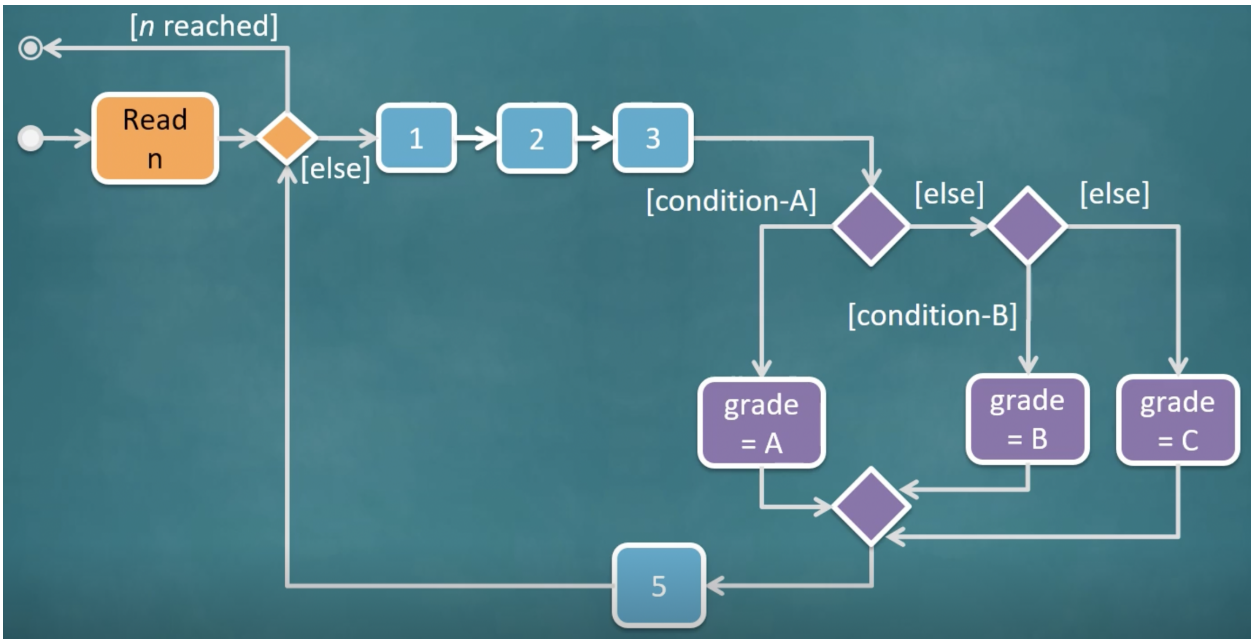


The rake symbol (in the Move piece action above) is used to show that the action is described in another subsidiary activity diagram elsewhere. That diagram is given below.

Swim Lanes

- partitioned activity diagrams to show different parties doing different actions



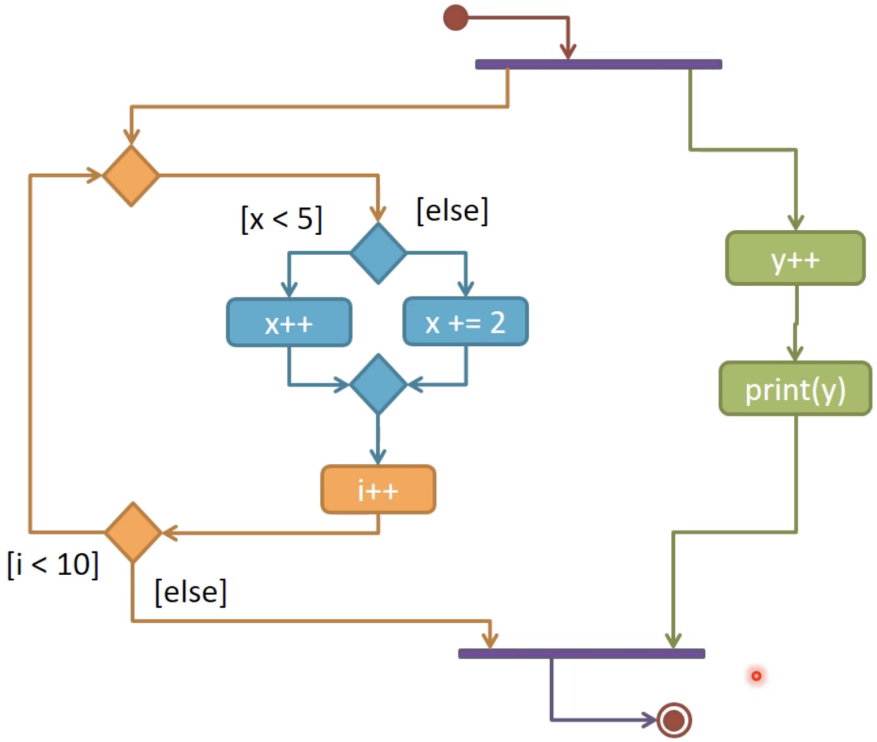


dn `increment count` (no need include implementation details)

Draw an activity diagram to illustrate that the following two code snippets are executed in parallel.

```
do {
    x < 5 ? x++ : x += 2;
    i++;
} while (i < 10);
```

```
y++;
print(y);
```

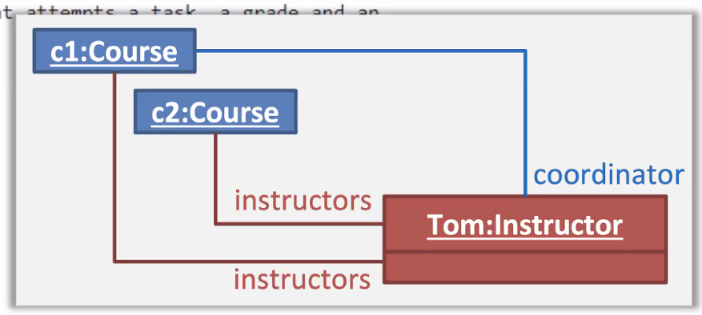


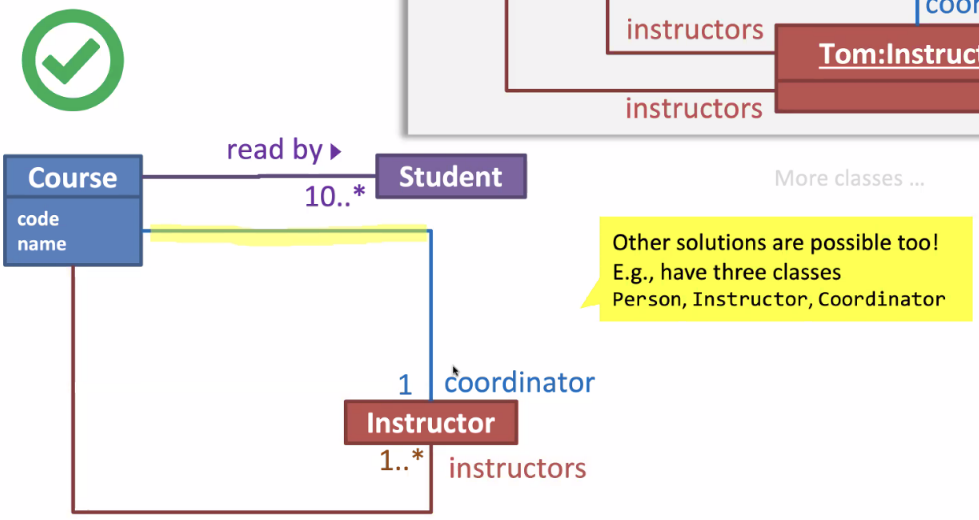
Tutorial 9

13

(i) Draw an OODM for the description below, about how *courses* work in a certain university:

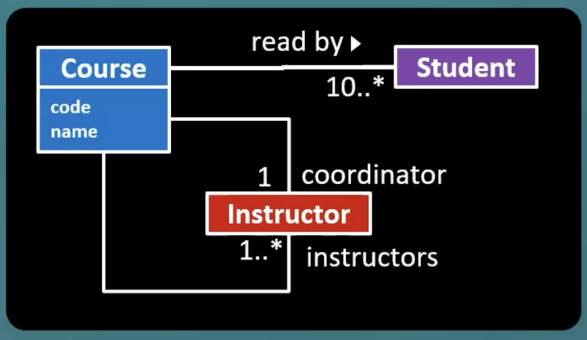
A course has a name and a code. A course is read by 10 or more students, and taught by one or more instructors one of whom is the coordinator. A course can have a number of tasks which can be assignments or tests. Some assignments are compulsory. When a student attempts a task, a grade and an optional feedback is given.





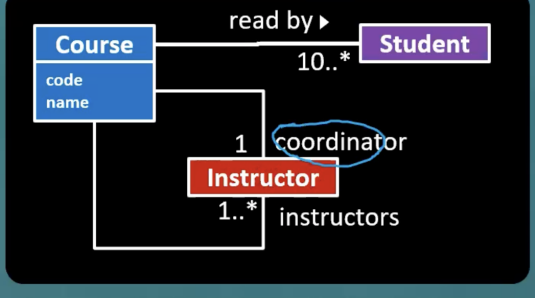
Caution: avoid implementation-specific terms such as design, variable, method when talking about domain models

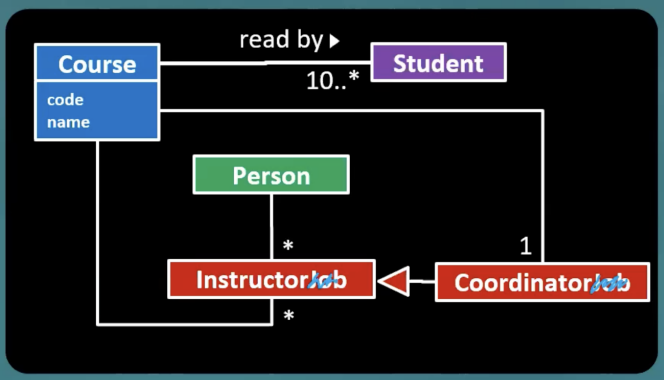
Which statements about this OODM are correct?

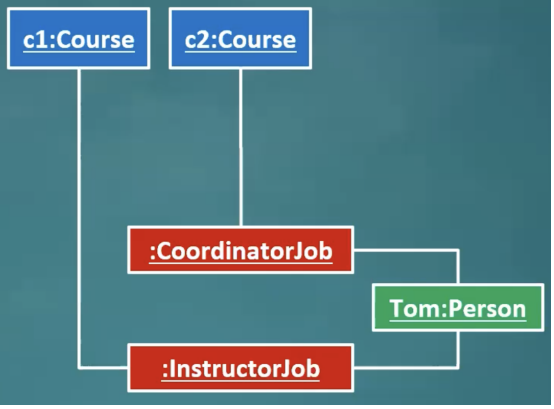


OODM is a model of the real world. The design and the implementation of the solution can be guided by it, but it's not the same.
→ **avoid implementation-specific terminology** when discussing the problem domain.

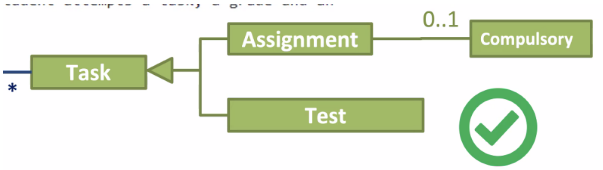
- ☐ A. The `instructors` variable in the `Course` class can be of type `ArrayList<Instructor>`
- ☐ B. The `Course` class can have methods to add students to a specific course.
- ☐ C. There are alternative designs that can be more efficient to implement.
- ☑ D. Although not shown in the diagram, it is likely the `Student` class has attributes such as name, student number, gender etc.
- ☑ E. Instructor objects can play the *role* of coordinator.

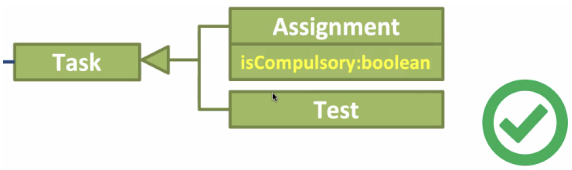




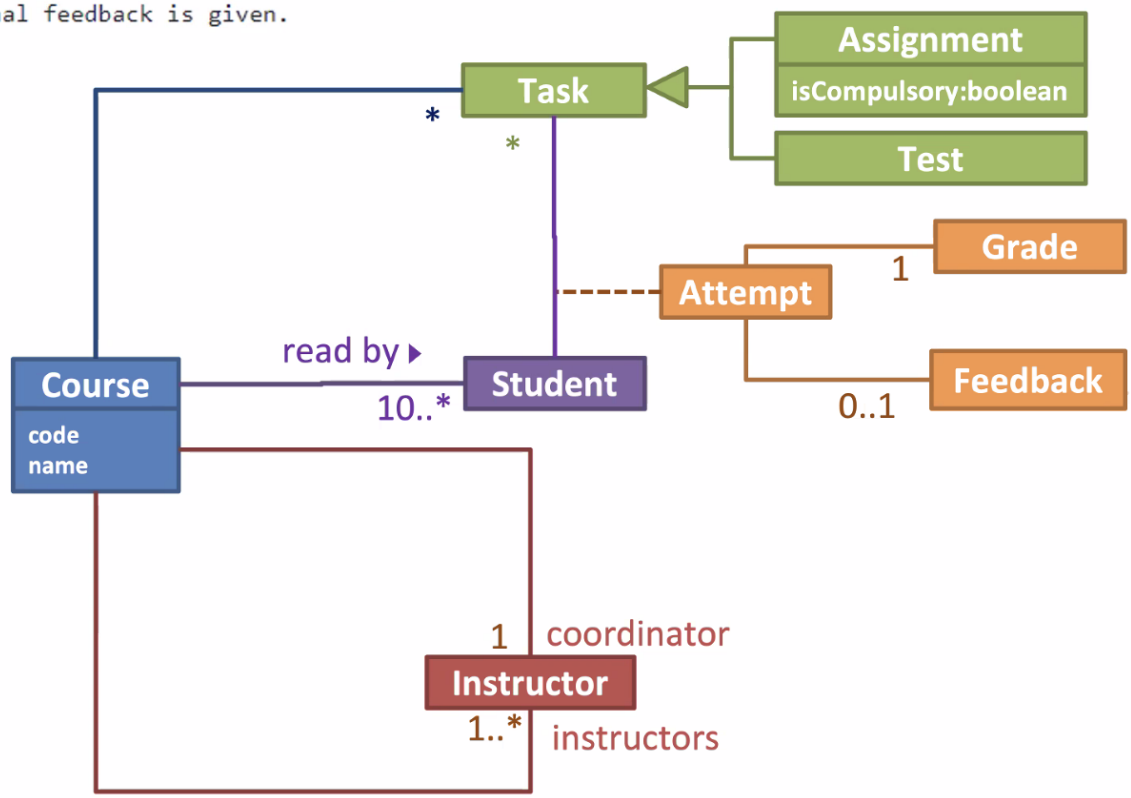


alternate ans



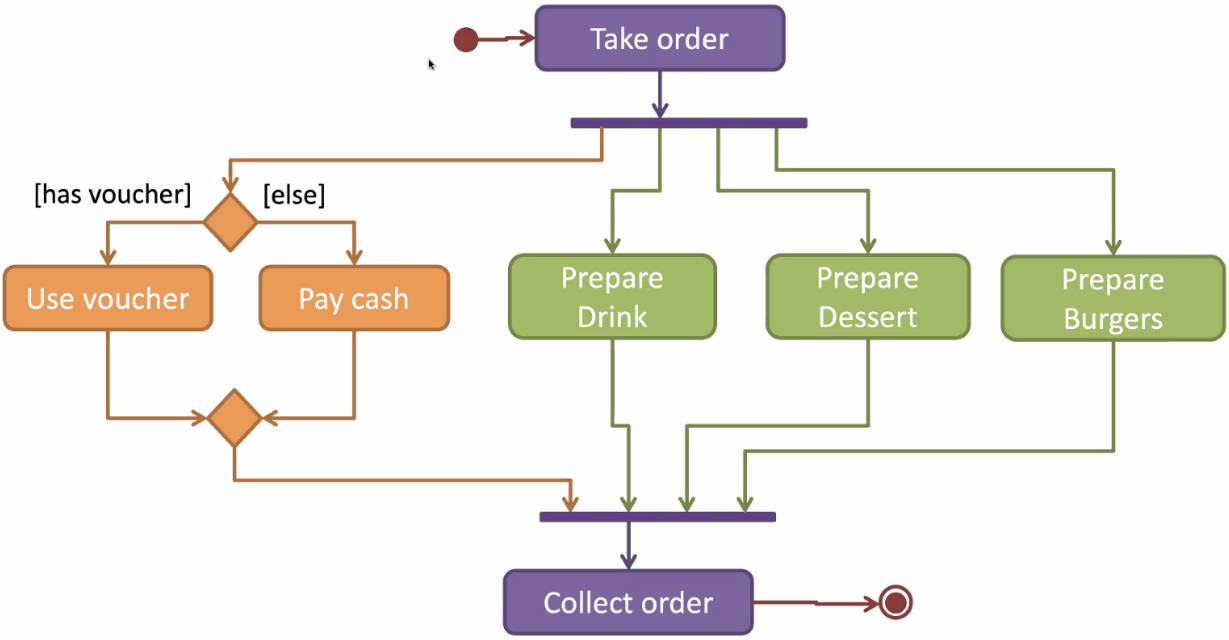


A course has a name and a code. A course is read by 10 or more students, and taught by one or more instructors one of whom is the coordinator. A course can have a number of tasks which can be assignments or tests. Some assignments are compulsory. When a student attempts a task, a grade and an optional feedback is given.



Draw an activity diagram to represent the following workflow a burger shop uses when processing an order by a customer.

- First, a cashier takes the order.
- Then, three workers start preparing the order at the same time; one prepares the drinks, one prepares the burgers, and one prepares the desserts.
- In the meantime, the customer pays for the order. If the customer has a voucher, she pays using the voucher; otherwise she pays using cash.
- After paying, the customer collects the food after all three parts of the order are ready.

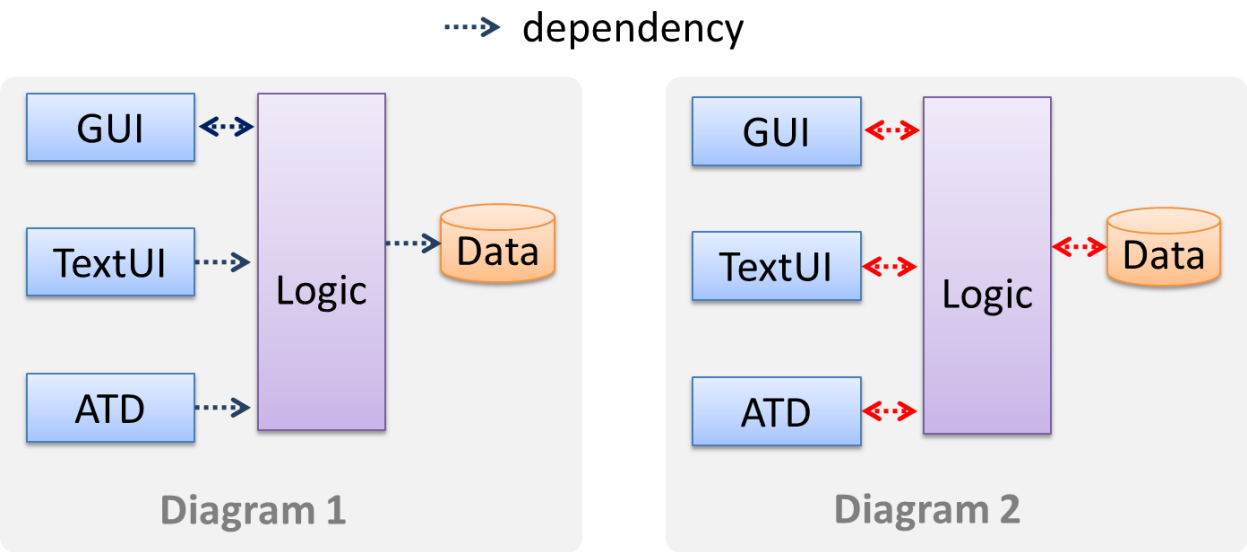


Architecture Diagrams

- shows the overall organisation of the system and can be viewed as a very high-level design.
- no standard conventions

General Guidelines

- Minimize the variety of symbols.
  - If the symbols you choose do not have widely-understood meanings, explain their meaning.
  - e.g. A drum symbol is widely-understood as representing a database → dn explain meaning
- Avoid the indiscriminate use of double-headed arrows to show interactions between components.

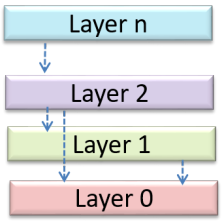


Consider the two architecture diagrams of the same software given below. Because `Diagram 2` uses double-headed arrows, the important fact that GUI has a bidirectional dependency with the Logic component is no longer captured.

Architectural Styles

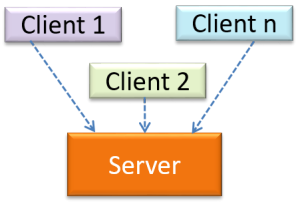
n-tier (multi-layered, layered)

- In the n-tier style, higher layers make use of services provided by lower layers.
- Lower layers are independent of higher layers.



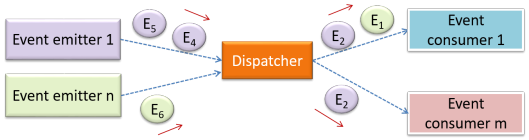
Client-Server

- The 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.



Event-Driven

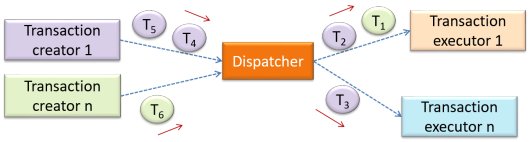
- Event-driven style controls the flow of the application by detecting events from event emitters and communicating those events to interested event consumers.



often used in GUIs.

Transaction Processing

- The 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 Style

- The service-oriented architecture (SOA) style builds applications by combining functionalities packaged as programmatically accessible services.
- SOA 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.

Use Case Diagrams

- illustrate use cases of a system visually, providing a visual ‘table of contents’ of the use cases of a system.
- Use cases can be specified at various levels of detail.
  - start with high level use cases and progressively work towards lower level use cases
- A use case describes only the externally visible behaviour, not internal details, of a system
  - i.e. should minimise details that are not part of the interaction between the user and the system.
  - e.g. `LMS saves the file into the cache and indicates success.`
- UI details are usually omitted
  - 👍 User clears input.
  - 👎 User right-clicks the text box and chooses ‘clear’

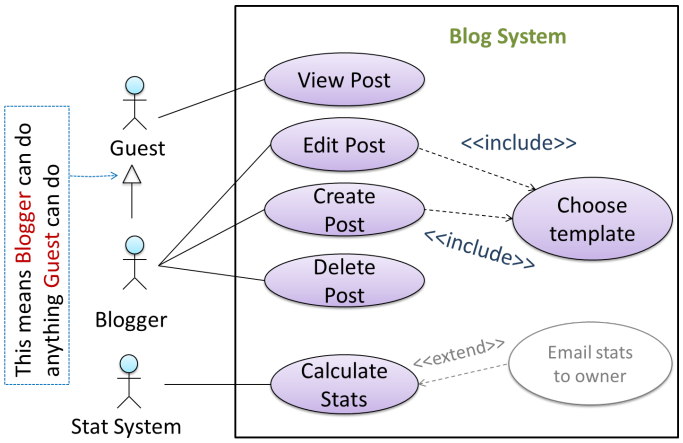
```
Software System: SquareGame
Use case: UC02 - Play a Game
Actors: Player (multiple players)

1. A Player starts the game.
2. SquareGame asks for player names.
3. Each Player enters his own name.
4. SquareGame shows the order of play.
5. SquareGame prompts for the current Player to throw a die.
6. Current Player adjusts the throw speed.
7. Current Player triggers the die throw.
8. SquareGame shows the face value of the die.
9. SquareGame moves the Player's piece accordingly.
   Steps 5-9 are repeated for each Player, and for as many rounds as required until a Player reaches the 100th square.
10. SquareGame shows the Winner.

Use case ends.
```

Actors (Roles)

- An actor (in a use case) is a role played by a user.
- An actor can be a human or another system.
- Actors are not part of the system; they reside outside the system.
- A use case can involve multiple actors.
- An actor can be involved in many use cases.
- A single person/system can play many roles.
    - a student can be a student, guest or tutor
- Many persons/systems can play a single role.
    - e.g. undergrad, grad, part-time students are all students
- Although system may be represented as an actor, but it is not recommended

Actor Generalisation

- similar to class inheritance



actor Blogger can do all the use cases the actor Guest can do, as a result of the actor generalisation relationship given in the diagram.

Main Success Scenario (MSS)

- describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong.
- aka: Basic Course of Action or the Main Flow of Events of a use case.

Extensions

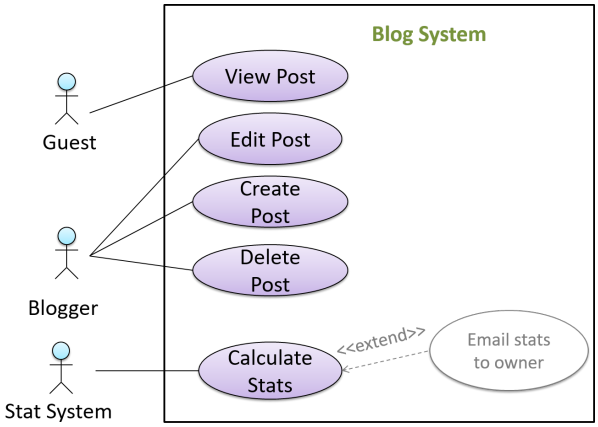
- "add-on"s to the MSS that describe exceptional/alternative flow of events
- Extensions appear below the MSS.

```
System: Online Banking System (OBS)
Use case: UC23 - Transfer Money
Actor: User
MSS:
  1. User chooses to transfer money.
  2. OBS requests for details of the transfer.
  3. User enters the requested details.
  4. OBS requests for confirmation.
  5. User confirms.
  6. OBS transfers the money and displays the new account balance.
  Use case ends.
```

```
Extensions:
  3a. OBS detects an error in the entered data.
      3a1. OBS requests for the correct data.
      3a2. User enters new data.
      Steps 3a1-3a2 are repeated until the data entered are correct.
      Use case resumes from step 4.

  3b. User requests to effect the transfer in a future date.
      3b1. OBS requests for confirmation.
      3b2. User confirms future transfer.
      Use case ends.

  *a. At any time, User chooses to cancel the transfer.
```

```
      *a1. OBS requests to confirm the cancellation.
      *a2. User confirms the cancellation.
      Use case ends.

  *b. At any time, 120 seconds lapse without any input from the User.
      *b1. OBS cancels the transfer.
      *b2. OBS informs the User of the cancellation.
      Use case ends.
```

- either of the extensions marked `3a.` and `3b.` can happen just after step `3` of the MSS.
- the extension marked as `a.` can happen at any step (hence, the `*`).



direction of the arrow is from the extension to the use case it extends and the arrow uses a dashed line.

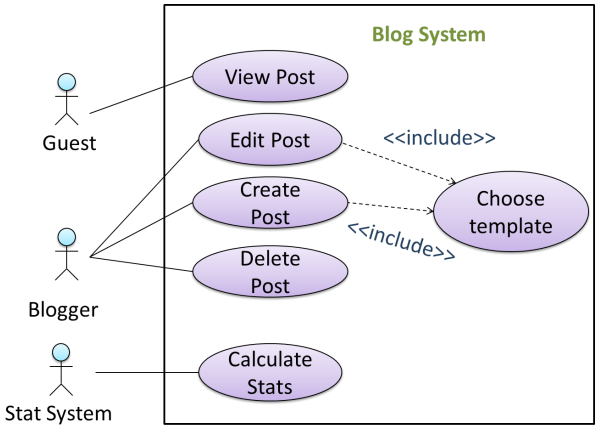
Inclusion

- a use case can include another use case
- Underlined text is commonly used to show an inclusion of a use case.

```
Software System: LearnSys
Use case: UC01 - Conduct Survey
Actors: Staff, Student
MSS:
1. Staff creates the survey (UC44).
2. Student completes the survey (UC50).
3. Staff views the survey results.
   Use case ends.
```

Inclusions are useful,

- when you don't want to clutter a use case with too many low-level steps.
- when a set of steps is repeated in multiple use cases.



dashed arrow from use case to inclusion

Preconditions

- specify the specific state you 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.

```
Software System: Online Banking System
Use case: UC23 - Transfer Money
Actor: User
Preconditions: User is logged in.
Guarantees:
- Money will be deducted from the source account only if the transfer to the destination account is successful.
- The transfer will not result in the account balance going below the minimum balance required.

MSS:

User chooses to transfer money.
OBS requests for details for the transfer.
...
```

Design

Design Approaches

Top-Down

- Design the high-level design first and flesh out the lower levels later.

- This is especially 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.
- This is not usually scalable for bigger systems.
- One instance where this approach might work is when designing a variation of an existing system or re-purposing existing components to build a new system.

Mix

- Design the top levels using the top-down approach but switch to a bottom-up approach when designing the bottom levels.

Agile

- emergent, not defined up front
- code before documentation
- multiple iterations

Design Fundamentals

Abstraction

- only details that are relevant to the current perspective or the task at hand need to be considered

Data abstraction

- abstracting away the lower level data items and thinking in terms of bigger entities
- e.g. Within a certain software component, you might deal with a user data type, while ignoring the details contained in the user data item such as name, and date of birth. These details have been ‘abstracted away’ as they do not affect the task of that software component.

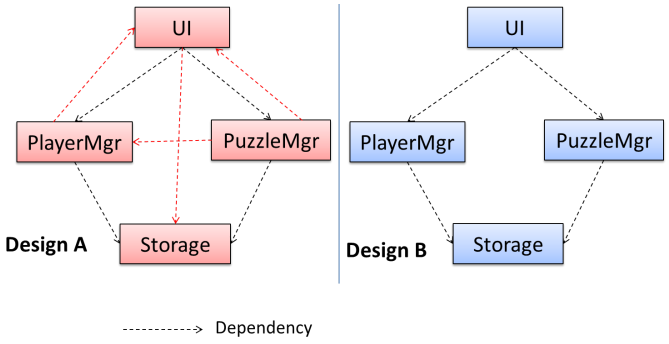
Control abstraction

- abstracting away details of the actual control flow to focus on tasks at a higher level
- e.g. `print(“Hello”)` is an abstraction of the actual output mechanism within the computer.

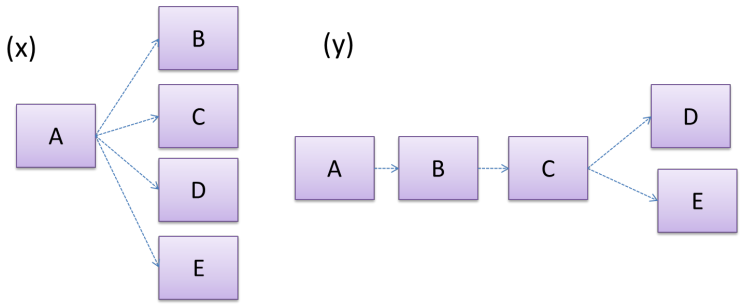
Coupling

- a measure of the degree of dependence between components, classes, methods, etc.
- Low coupling indicates that a component is less dependent on other components.
- High coupling (aka tight coupling or strong coupling) is discouraged:
  - harder maintenance (ripple effect of changes)
  - harder integration (multiple component have to be integrated at the same time)
  - harder testing and reusability of the module (lower testability and reusability)
  - value of automated regression testing increases
  - higher risk of regression

- X is coupled to Y if a change to Y can potentially require a change in X.
- If the `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 the `Foo` class
- e.g. if the signature of `Bar#read()` is changed, `Foo` needs to change as well, but a change to the `Bar#write()` method may not require a change in the `Foo` class because `Foo` does not call `Bar#write()`.



A has more coupling



- Overall coupling levels in x and y seem to be similar (neither has more dependencies than the other).
  - Note that the number of dependency links is not a definitive measure of the level of coupling. Some links may be stronger than the others.

- However, in x, `A` is highly-coupled to the rest of the system while `B`, `C`, `D`, and `E` are standalone (do not depend on anything else).
- In y, no component is as highly-coupled as `A` of x. However, only `D` and `E` are standalone.

Examples of coupling:

`A` is coupled to `B` if,

- `A` has access to the internal structure of `B` (this results in a very high level of coupling)
- `A` and `B` depend on the same global variable
- `A` calls `B`
- `A` receives an object of `B` as a parameter or a return value
- `A` inherits from `B`
- `A` and `B` are required to follow the same data format or communication protocol

Cohesion

- a measure of how strongly-related and focused the various responsibilities of a component are.
- A highly-cohesive component keeps related functionalities together while keeping out all other unrelated things.
- higher cohesion is better

Disadvantages of low/weak cohesion

- Lowers the understandability of modules as it is difficult to express module functionalities at a higher level.
- Lowers maintainability because a module can be modified due to unrelated causes (reason: the module contains code unrelated to each other) or
  - many modules may need to be modified to achieve a small change in behaviour (reason: because the code related to that change is not localised to a single module).
- Lowers reusability of modules because they do not represent logical units of functionality.

Example of cohesion

- Code related to a single concept is kept together,
  - e.g. the `Student` component handles everything related to students.
- Code that is invoked close together in time is kept together,
  - e.g. all code related to initialising the system is kept together.
- Code that manipulates the same data structure is kept together,

- e.g. the `GameArchive` component handles everything related to the storage and retrieval of game sessions.

Integration Approaches

- Big-bang integration: integrate all components at the same time. (NOT recommended)
- Incremental/continuous integration: integrate a few components at a time. This approach is better than big-bang integration because it surfaces integration problems in a more manageable way.

Design Principles

Separation of Concerns (SoC)

- separate the code into distinct sections, such that each section addresses a separate concern.
- reduces functional overlaps among code sections and also limits the ripple effect
- lead to better modularity, higher cohesion and lower coupling.
- ~single responsibility principle

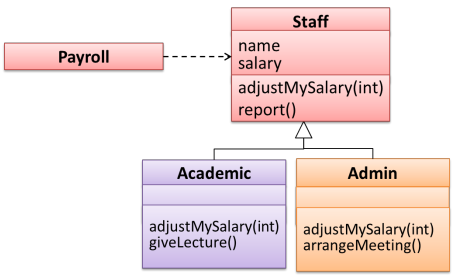
Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change (for only one responsibility)
- e.g. Consider a TextUi class that does parsing of the user commands as well as interacting with the user. That class needs to change when the formatting of the UI changes as well as when the syntax of the user command changes. Hence, such a class does not follow the SRP.
  - Separate into `TextUi` and `Formatter`

Liskov Substitution Principle (LSP)

- Derived classes must be substitutable for their base classes
- LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.

e.g. Suppose the `Payroll` class depends on the `adjustMySalary(int percent)` method of the `Staff` class. Furthermore, the `Staff` class states that the `adjustMySalary` method will work for all positive percent values. Both the `Admin` and `Academic` classes override the `adjustMySalary` method.

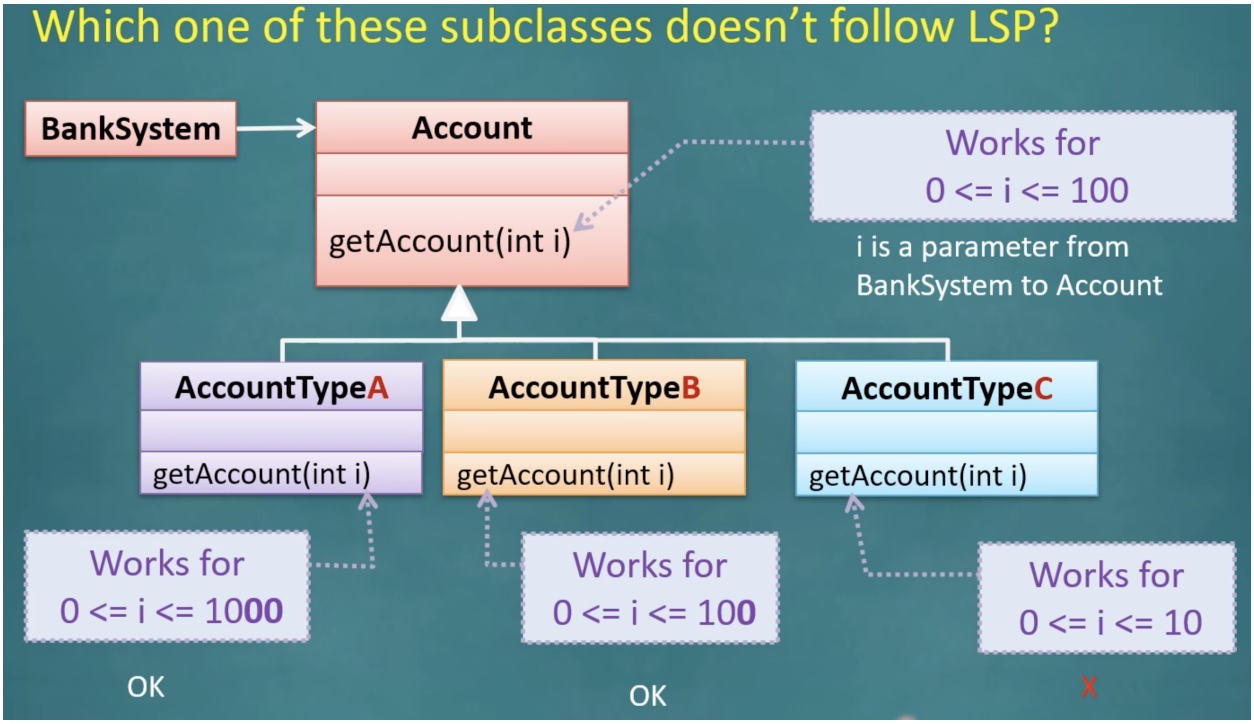


Now consider the following:

- The `Admin#adjustMySalary` method works for both negative and positive percent values.
- The `Academic#adjustMySalary` method works for percent values `1..100` only.

In the above scenario,

- The `Admin` class follows LSP because it fulfills `Payroll`'s expectation of `Staff` objects (i.e. it works for all positive values). Substituting `Admin` objects for `Staff` objects will not break the `Payroll` class functionality.
- The `Academic` class violates LSP because it will not work for percent values over `100` as expected by the `Payroll` class. Substituting `Academic` objects for `Staff` objects can potentially break the `Payroll` class functionality.

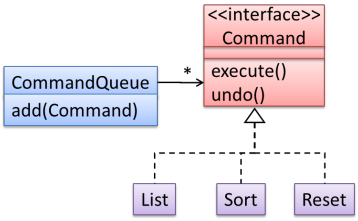


Open-Closed Principle (OCP)

- Def: A module should be open for extension but closed for modification.
  - modules should be written so that they can be extended, without requiring them to be modified.
  - able to change a software module's behavior without modifying its code.
- aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself.
- often requires separating the specification (i.e. interface) of a module from its implementation.
- e.g. the behaviour of the `CommandQueue` class can be altered by adding more concrete `Command` subclasses.
  - For example, by including a `Delete` class alongside `List`, `Sort`, and `Reset`, the `CommandQueue` can now perform delete commands

without modifying its code at all.

- That is, its behaviour was extended without having to modify its code. Hence, it is open to extensions, but closed to modification.

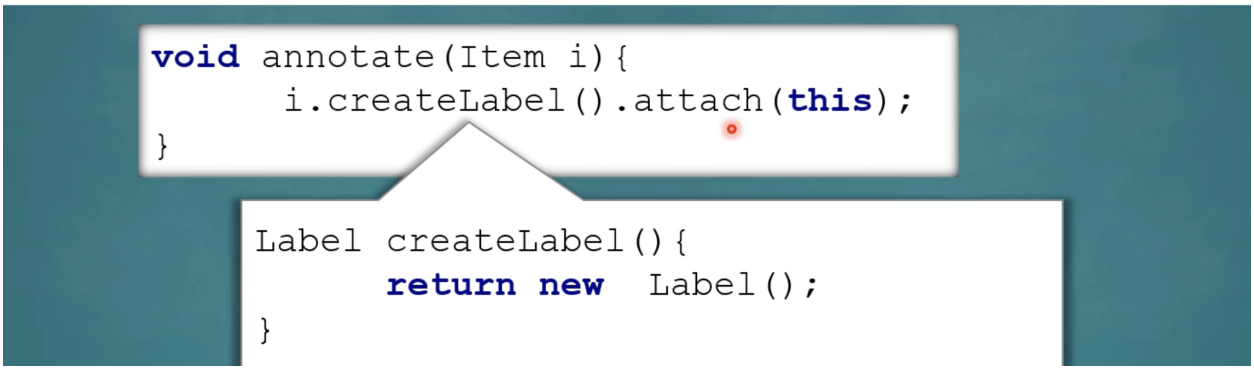


Law of Demeter (LoD)

- An object should have limited knowledge of another object.
- An object should only interact with objects that are closely related to it.
- reduce coupling
- aka
  - Don't talk to strangers.
  - Principle of least knowledge

More concretely, a method `m` of an object `o` should invoke only the methods of the following kinds of objects:

- The object `o` itself
- Objects passed as parameters of `m`
- Objects created/instantiated in `m`
- Objects from the direct association of `o`



e.g. Let us take the `Logic` class as an example. Assume that it has the following operation.

`setMinefield(Minefield mf): void`

Consider the following that can happen inside this operation:

- `mf.init();`: this does not violate LoD since LoD allows calling operations of parameters received.
- `mf.getCell(1, 3).clear();`: this violates LoD because `Logic` is handling `Cell` objects deep inside `Minefield`. Instead, it should be `mf.clearCellAt(1, 3);`.

- `timer.start();`: this does not violate LoD because `timer` appears to be an internal component (i.e. a variable) of `Logic` itself.
- `Cell c = new Cell();` `c.init();`: this does not violate LoD because `c` was created inside the operation.

SOLID:

- SRP
- OCP
- LSP
- ISP (Interface Segregation Principle)
- DIP (Dependency Inversion Principle)

Design Pattern

- def: An elegant reusable solution to a commonly recurring problem within a given context in software design.

The common format to describe a pattern consists of the following components:

- **Context**: The situation or scenario where the design problem is encountered.
- **Problem**: The main difficulty to be resolved.
- **Solution**: The core of the solution. It is important to note that the solution presented only includes the most general details, which may need further refinement for a specific context.
- **Anti-patterns** (optional): Commonly used solutions, which are usually incorrect and/or inferior to the Design Pattern.
- **Consequences** (optional): Identifying the pros and cons of applying the pattern.
- **Other useful information** (optional): Code examples, known uses, other related patterns, etc.

Singleton

Context

- Certain classes should have no more than one instance (e.g. the main controller class of the system).
- These single instances are commonly known as *singletons*.

Problem

- A normal class can be instantiated multiple times by invoking the constructor.

Solution

- Make the constructor of the singleton class `private`, because a `public` constructor will allow others to instantiate the class at will.

- Provide a `public` class-level method to access the *single instance*.

Pros and Cons of Singleton

Name	Tags
Pros	
Cons	

Facade Pattern

Context

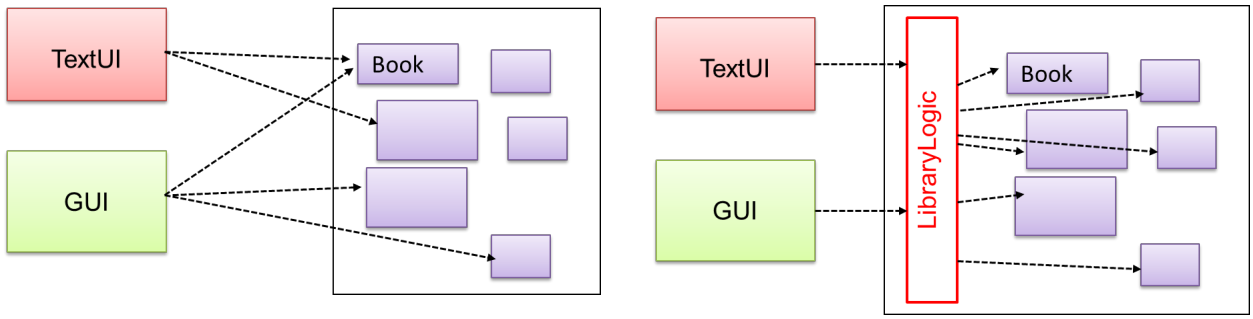
- Components need to access functionality deep inside other components.

Problem

- Access to the component should be allowed without exposing its internal details.
- e.g. 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.



ure | cls.nus.edu.sg/nusgerman/data/course_path/686-20200323163642.pdf

Command Pattern

Context

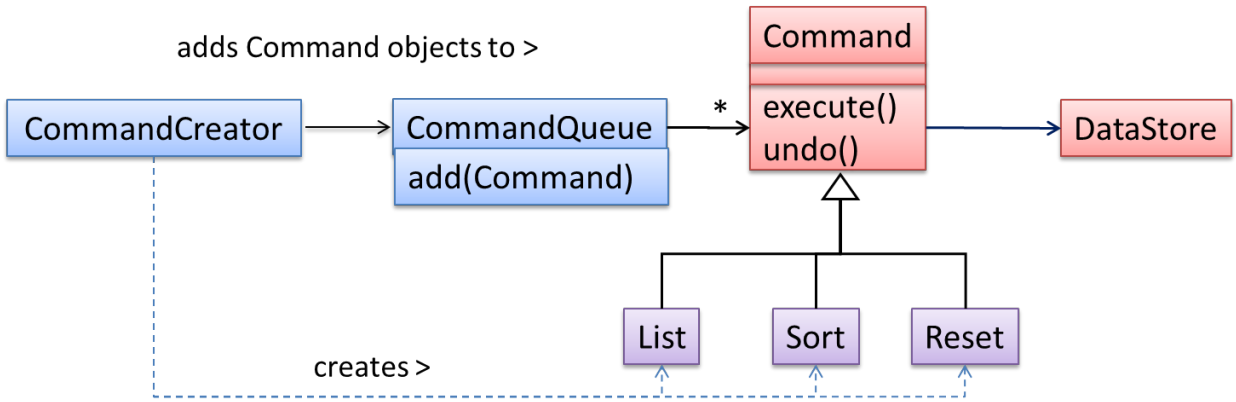
- A system is required to execute a number of commands, each doing a different task.
- e.g., a system might have to support `Sort`, `List`, `Reset` commands.

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).



Model-View-Controller Pattern (MVC)

Context

- 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

Solution

- Decouple data, presentation, and control logic of an application by separating them into three different components: *Model*, *View* and *Controller*.
  - *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.

Observer Pattern

Context

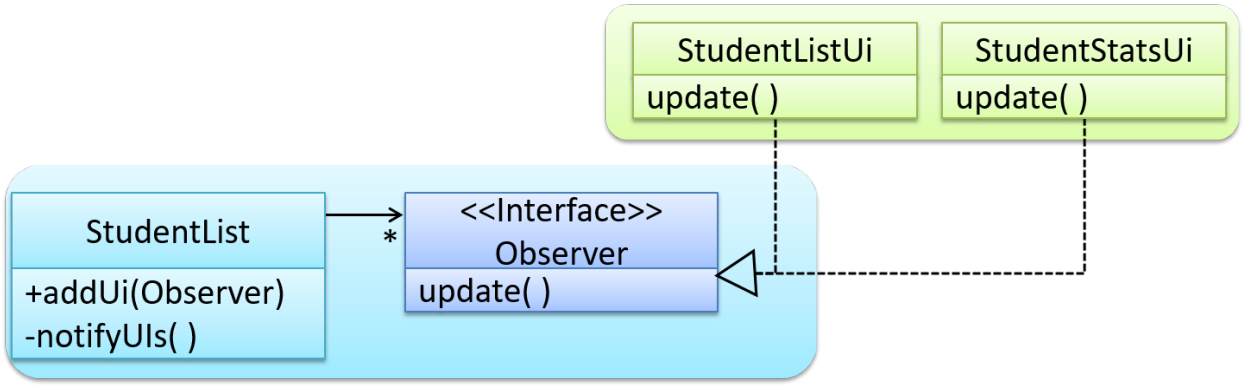
- An object (possibly more than one) wants to be notified when a change happens to another object.
- i.e., some objects want to 'observe' another object.

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.



During the initialization of the system,

1. First, create the relevant objects.

```
StudentList studentList = new StudentList();
StudentListUi listUi = new StudentListUi();
StudentStatsUi statsUi = new StudentStatsUi();
```

2. Next, the two UIs indicate to the `StudentList` that they are interested in being updated whenever `StudentList` changes. This is also known as ‘subscribing for updates’.

```
studentList.addUi(listUi);
studentList.addUi(statsUi);
```

3. Within the `addUi` operation of `StudentList`, all Observer object subscribers are added to an internal data structure called `observerList`.

```
// StudentList class
public void addUi(Observer o) {
    observerList.add(o);
}
```

Now, whenever the data in `StudentList` changes (e.g. when a new student is added to the `StudentList`),

1. All interested observers are updated by calling the `notifyUIs` operation.

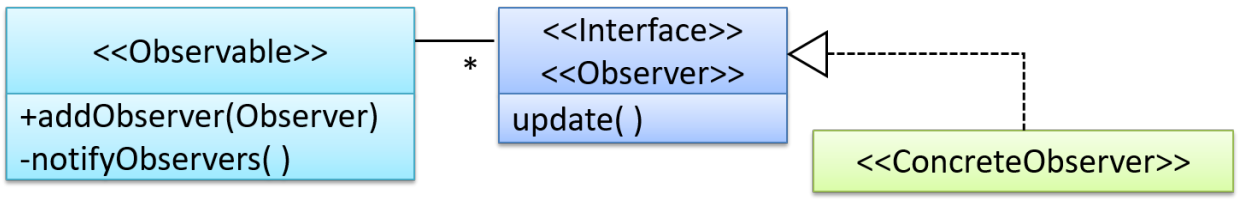
```
// StudentList class
public void notifyUIs() {
    // for each observer in the list
    for (Observer o: observerList) {
        o.update();
```

```
    }
}
```

2. UIs can then pull data from the `StudentList` whenever the `update` operation is called.

```
// StudentListUI class
public void update() {
    // refresh UI by pulling data from StudentList
}
```

Note that `StudentList` is unaware of the exact nature of the two UIs but still manages to communicate with them via an intermediary.



- `<<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.

Error Handling

Defensive Programming

- proactively tries to eliminate any room for things to go wrong.

```
class MainApp {
    Config config;

    /** Returns the config object */
    Config getConfig() {
        return config;
    }

    /** Returns a copy of the config object */
    Config getConfig() {
        return config.copy(); // return a defensive copy
    }
}
```

- enforce compulsory associations

```
class Account {
    Guarantor guarantor;

    void setGuarantor(Guarantor g) {
        guarantor = g;
    }
}

Account a = new Account();
a.setGuarantor(null); // results in an account w/o Guarantor
/*********************************************************/
// defensive programming way
class Account {
    private Guarantor guarantor;

    public Account(Guarantor g) {
        if (g == null) {
            stopSystemWithMessage(
                    "multiplicity violated. Null Guarantor");
        }
        guarantor = g;
    }
    public void setGuarantor(Guarantor g) {
        if (g == null) {
            stopSystemWithMessage(
                    "multiplicity violated. Null Guarantor");
        }
        guarantor = g;
    }
}
```

- defensive programming can
  - make program slower
  - make code longer and more complex
  - make code less susceptible to misuse
  - require extra efforts
- depends on
  - 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

Testing

Integration Testing

- Integration testing is not simply repeating the unit test cases using the actual dependencies (instead of the stubs used in unit testing).
- Integration tests are additional test cases that focus on the interactions between the parts.

- Pure integration tests:

a. First, unit test `Engine` and `Wheel`

b. Next, unit test `Car` in isolation of `Engine` and `Wheel`, using stubs for `Engine` and `Wheel`

c. After that, do an integration test for `Car` by using it together with `Engine` and `Wheel` classes to ensure that `Car` integrates properly with the `Engine` and the `Wheel`.

- In practice, developers often use a hybrid of unit+integration tests to minimise the need for stubs.

a. First, unit test `Engine` and `Wheel`

b. After that, do an integration test for `Car` by using it together with the `Engine` and `Wheel` classes to ensure that `Car` integrates properly with the `Engine` and the `Wheel`. This step should include test cases that are meant to unit test `Car` (i.e. test cases used in the step (b) of the example above) as well as test cases that are meant to test the integration of `Car` with `Wheel` and `Engine` (i.e. pure integration test cases used of the step (c) in the example above).

- no longer need stubs for `Engine` and `Wheel`
- downside is `Car` is never tested in isolation of its dependencies.

System Testing

- take the whole system and test it against the system specification.
- System test cases are based on the specified external behaviour of the system.
  - can go beyond the bounds defined in the specification
- includes testing against NFRs
  - *Performance testing* – to ensure the system responds quickly.
  - *Load testing* (also called *stress testing* or *scalability testing*) – to ensure the system can work under heavy load.
  - *Security testing* – to test how secure the system is.
  - *Compatibility testing, interoperability testing* – to check whether the system can work with other systems.
  - *Usability testing* – to test how easy it is to use the system.
  - *Portability testing* – to test whether the system works on different platforms.

Automated Testing of GUIs

- **TestFX** can do automated testing of JavaFX GUIs
- **Visual Studio** supports the ‘record replay’ type of GUI test automation.
- **Selenium** can be used to automate testing of web application UIs

(User) Acceptance Testing (UAT)

- test the system to ensure it meets the user requirements.
- give an assurance to the customer that the system does what it is intended to do.
- Acceptance test cases are often defined at the beginning of the project, usually based on the use case specification.
- negative test cases: cases where the SUT is not expected to work normally
  - e.g. incorrect inputs;
- positive test cases: cases where the SUT is expected to work normally
- in many cases one document serves as both a requirement specification and a system specification.

System Testing v.s. UAT

Aa 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 v.s. System specifications

Aa 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 solves 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)

Alpha / Beta Testing

Alpha testing is

- performed by the **users**,
- under controlled conditions set by the software development team.

Beta testing is

- performed by a selected subset of target users of the system
- in their natural work setting.

Exploratory v.s. Scripted Testing

1. **Scripted testing:** First write a set of test cases based on the expected behaviour of the SUT, and then perform testing based on that set of test cases.
    - “Scripted” means test cases are predetermined. They need not be an executable script.
    - more systematic, and hence, likely to discover more bugs given sufficient time
2. **Exploratory testing:** Devise test cases on-the-fly, creating new test cases based on the results of the past test cases.
    - aka reactive testing, error guessing technique, attack-based testing and bug hunting
    - usually manual
    - the nature of the follow-up test case is decided based on the behaviour of the previous test cases.
    - testing is driven by observations during testing
    - usually starts with areas identified as error-prone, based on the tester’s past experience with similar systems
    - success of exploratory testing depends on the tester’s prior experience and intuition
    - should be done by experienced testers, using a clear strategy/plan/framework
    - allows us to detect problems in a relatively short time but not prudent as the sole means of testing

Dependency Injection

- Dependency injection is the process of 'injecting' objects to replace current dependencies with a different object.
- This is often used to inject stubs to isolate the SUT (software under test) from its dependencies so that it can be tested in isolation.

Testability

- indication of how easy it is to test an SUT.
- higher testability, better quality

Test Coverage

- Test coverage is a metric used to measure the extent to which testing exercises the code i.e., how much of the code is 'covered' by the tests.
- **Function/method coverage** : based on functions executed e.g., testing executed 90 out of 100 functions.
- **Statement coverage** : based on the number of lines of code executed e.g., testing executed 23k out of 25k LOC.
- **Decision/branch coverage** : based on the decision points exercised e.g., an `if` statement evaluated to both `true` and `false` with separate test cases during testing is considered 'covered'.
- **Condition coverage** : based on the boolean sub-expressions, each evaluated to both true and false with different test cases. Condition coverage is not the same as the decision coverage.

`if(x > 2 && x < 44)` is considered one decision point but two conditions.

For 100% branch or decision coverage, two test cases are required:

- `(x > 2 && x < 44) == true` : [e.g. `x == 4`]
- `(x > 2 && x < 44) == false` : [e.g. `x == 100`]

For 100% condition coverage, three test cases are required:

- `(x > 2) == true`, `(x < 44) == true` : [e.g. `x == 4`]
- `(x < 44) == false` : [e.g. `x == 100`]
- `(x > 2) == false` : [e.g. `x == 0`]

- **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. A commonly used notation for path analysis is called the *Control Flow Graph (CFG)*.
  - highest intensity of testing
- **Entry/exit coverage** measures coverage in terms of possible *calls to* and *exits* from the operations in the SUT.

Test-Driven Development (TDD)

- advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments.
- define the precise behaviour of the SUT using test cases, and then write the SUT to match the specified behaviour.
- adv: guarantees the code is testable.

Test Input Combinations Strategies

- all combinations strategy: generates test cases for each unique combination of test inputs → too much, inefficient
- at least once strategy: includes each test input at least once → too few, may miss bugs
- all pairs strategy: creates test cases so that for any given pair of inputs, all combinations between them are tested
- random strategy: generates test cases using one of the other strategies and then picks a subset randomly
- Heuristics:
  - all valid inputs should be tested at least once in positive tests
  - should not have more than 1 invalid input in a test case
  - cannot give precise number of test cases, depends on one's judgement

Values	Explanation
Apple	Label format is round
Banana	Label format is oval
Cherry	Label format is square
Dog	Not a valid fruit

Values	Explanation
1	Only one digit
20	Two digits
0	Invalid because 0 is not a valid price
-1	Invalid because negative prices are not allowed

Negative Example

Case	fruitName	unitPrice	Expected
1	Apple	1	Print label
2	Banana	20	Print label
3	Cherry	0	Error message “invalid price”
4	Dog	-1	Error message “invalid fruit"

Cherry -- the only input that can produce a square-format label -- is in a negative test case which produces an error message instead of a label. If there is a bug in the code that prints labels in square-format, these tests cases will not trigger that bug.

Positive Example

Case	fruitName	unitPrice	Expected
1	Apple	1	Print round label
2	Banana	20	Print oval label
2.1	Cherry	VV	Print square label
3	VV	0	Error message “invalid price”
4	VV	-1	Error message “invalid price"
4.1	Dog	VV	Error message “invalid fruit"

VV/IV = Any Invalid or Valid Value VV = Any Valid Value

Formal Methods

- Formal verification uses mathematical techniques to prove the correctness of a program.

Advantages:

- **Formal verification can be used to prove the absence of errors**. In contrast, testing can only prove the presence of errors, not their absence.

Disadvantages:

- It only proves the compliance with the specification, but not the actual utility of the software.
- It 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**.

Test Case Design

Positive Test

- test is designed to produce an expected/valid behaviour.

Negative Test

- designed to produce a behaviour that indicates an invalid/unexpected situation, such as an error message.

Black Box

- aka specification-based or responsibility-based approach
- test cases are designed exclusively based on the SUT's specified external behaviour.

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.

Grey Box

- test case design uses some important information about the implementation.
- e.g., 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.

Equivalence Partition (aka equivalence class)

- A group of test inputs that are likely to be processed by the SUT in the same way.
- e.g. `isValidMonth(m)` : returns `true` if `m` (and `int`) is in the range `[1..12]`
  - [-MIN_INT, 0]: false
  - [1, 12]: true
  - [13, MAX_INT]: false
- Some inputs have only a small number of possible values and a potentially unique behavior for each value → consider each value as a partition.
  - e.g. Consider the method `showStatusMessage(GameStatus s)` : `String` that returns a unique `String` for each of the possible values of `s` (GameStatus is an `enum`).
  - In this case, each possible value of s will have to be considered as a partition.

By dividing possible inputs into equivalence partitions you can,

- **avoid testing too many inputs from one partition** → increases the efficiency of testing by reducing redundant test cases.
- **ensure all partitions are tested** → increases the effectiveness of testing by increasing the chance of finding bugs.

Boundary Value Analysis

- a test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.

- when picking test inputs from an equivalence partition, values near boundaries (i.e. boundary values) are more likely to find bugs.
- Boundary values are aka corner cases.
- Typically, you should choose three values around the boundary to test: one value from the boundary, one value just below the boundary, and one value just above the boundary.

Aa Equivalence partition	Some possible test values (boundaries are in bold)
[1-12]	0,**1**,2, 11,**12**,13
[MIN_INT, 0] (MIN_INT is the minimum possible integer value allowed by the environment)	**MIN_INT**, MIN_INT+1, -1, **0** , 1
[any non-null String](assuming string length is the aspect of interest)	**Empty String**, **a String of maximum possible length**
[prime numbers]["F"]["A", "D", "X"]	No specific boundary for all 3
[non-empty Stack](assuming a fixed size stack)	Stack with: no elements, **one element**, two elements, **no empty spaces**, only one empty space

Reuse

Application Programming Interface (API)

- specifies the interface through which other programs can interact with a software component
- A class has an API (e.g., API of the Java String class, API of the Python str class) which is a collection of public methods that you can invoke to make use of the class.

Libraries

- A library is a collection of modular code that is general and can be used by other programs.
- Java classes you get with the JDK (such as `String`, `ArrayList`, `HashMap`, etc.) are library classes that are provided in the default Java distribution.
- Natty is a Java library that can be used for parsing strings that represent dates e.g. `The 31st of April in the year 2008`
- Your code calls the library code

Frameworks

- a reusable implementation of a software (or part thereof) providing generic functionality that can be selectively customized to produce a specific application.
- come with full or partial implementations.
- more concrete than patterns or principles.

- often configurable (customisable).
- Framework code calls your code.
- Frameworks use a technique called inversion of control, aka the “Hollywood principle” (i.e. don’t call us, we’ll call you!)

Platforms

- A platform provides a runtime environment for applications.
- A platform is often bundled with various libraries, tools, frameworks, and technologies in addition to a runtime environment but the defining characteristic of a software platform is the presence of a runtime environment.

Analysis