Command Line Interface (CLI)

is the primary mode of input.

- GUI - Give visual feedback to the user rather than to collect input
- Command-driven (as little use of mouse as possible)
    - Mouse actions should have keyboard alternatives
    - Typing is preferred over key combinations
    - One-shot commands

SE Pros and Cons

SE: Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" -- IEEE Standard Glossary of Software Engineering Terminology

Pros	Cons
● First is the sheer joy of making things. ● Second is the pleasure of making things that are useful to other people. ● Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning. ● Fourth is the joy of always learning, which springs from the nonrepeating nature of the task. ● Finally, there is the delight of working in such a tractable medium.	● First, one must perform perfectly. ● Next, other people set one's objectives, provide one's resources, and furnish one's information. ● The dependence upon others' programs. ● Bugs ● The product over which one has labored so long appears to be obsolete upon (or before) completion.

E1

Compare Software Engineering with Civil Engineering in terms of how work products in CE (i.e. buildings) differ from those of SE (i.e. software).

Buildings	Software
Visible, tangible	Invisible, intangible
Wears out over time	Does not wear out
Change is limited by physical restrictions (e.g. difficult to remove a floor from a high rise building)	Change is not limited by such restrictions. Just change the code and recompile.
Creating an exact copy of a building is impossible. Creating a near copy is almost as costly as creating the original.	Any number of exact copies can be made with near zero cost.
Difficult to move.	Easily delivered from one place to another.

Many low-skilled workers following tried-and-tested procedures.	No low-skilled workers involved. Workers have more freedom to follow their own procedures.
Easier to assure quality (just follow accepted procedure).	Not easy to assure quality.
Majority of the work force has to be on location.	Can be built by people who are not even in the same country.
Raw materials are costly, costly equipment required.	Almost free raw materials and relatively cheap equipment.
Once construction is started, it is hard to do drastic changes to the design.	Building process is very flexible. Drastic design changes can be done, although costly
A lot of manual and menial labor involved.	Most work involves highly-skilled labor.
Generally robust. E.g. removing a single brick is unlikely to destroy a building.	More fragile than buildings. A single misplaced semicolon can render the whole system useless.

E2

Justify this statement: Coding is still a ‘design’ activity, not a ‘manufacturing’ activity. You may use a comparison (or an analogy) of Software engineering versus Civil Engineering to argue this point.

Arguments to support this statement:

- If coding is a manufacturing activity, we should be able to do it using robotic machines (just like in the car industry) or low-skilled laborers (like in the construction industry).
- If coding is a manufacturing activity, we wouldn’t be changing it so much after we code software. But if the code is in fact a ‘design’, yes, we would fiddle with it until we get it right.
- Manufacturing is the process of building a finished product based on the design. Code is the design. Manufacturing is what is done by the compiler (fully automated).

However, the type of ‘design’ that occurs during coding is at a much lower level than the ‘design’ that occurs before coding.

Integrated Development Environments (IDEs)

source code editor	syntax coloring, auto-completion, easy code navigation, error highlighting, code-snippet generation
compiler &/ interpreter	facilitates the compilation/linking/running/deployment of a program
debugger	allows the developer to execute the program one step at a time to observe the run-time behavior in order to locate bugs

others	support for automated testing, drag-and-drop construction of UI components, version management support, simulation of the target runtime platform, and modeling support

Testing

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.

We execute a set of test cases.

test case: specifies how to perform a test - input to SUT and expected behavior

Test cases can be determined based on:

- Specification
- Reviewing similar existing systems
- Comparing to the past behavior of the SUT

A test case failure is a mismatch between the expected behavior and the actual behavior. A failure is caused by a defect (or a bug).

Regression: Unintended and undesirable effects on the system, break other code

Regression testing: re-testing the software to detect regressions - more practical when it is automated

Automated test case: be run programmatically and the result of the test case is determined programmatically

Automated regression testing of CLI apps

1. Put all input in a file
2. Put all output in a file
3. $FILENAME <input.txt >actual.txt => will create an actual.txt file
4. FC expected.txt actual.txt => compare two files

Revision Control

Revision Control: the process of managing multiple versions of a piece of information

track the history and evolution of your project
makes it easier for you to collaborate
can help you to recover from mistakes
help you to work simultaneously on, and manage the drift between, multiple versions of your project

RCS: Revision Control Software are the software tools that automate the process of Revision Control i.e. managing revisions of software artifacts.

RCS tools store the history of the working directory as a series of commits.

To see what changed between two points of the history, you can ask the RCS tool to diff the two commits in concern.

To restore the state of the working directory at a point in the past, you can checkout the commit in concern.

DRCS vs CRCS

CRCS - centralized RCS

Do not support the notion of a local repo

DRCS - decentralized RCS

Core characteristic of RCS: can store snapshots of files and in an efficient way

Revision: A revision (some seem to use it interchangeably with version while others seem to distinguish the two -- here, let us treat them as the same, for simplicity) is a state of a piece of information at a specific time that is a result of some changes to it.
Repository: The database of the history of a directory being tracked by an RCS software (e.g. Git).
Remote Repositories: Copies of a repo that are hosted on remote computers.
You can clone a remote repo onto your computer which will create a copy of a remote repo on your computer, including the version history as the remote repo.
You can push new commits in your clone to the remote repo which will copy the new commits onto the remote repo.
You can pull from the remote repos to receive new commits in the remote repo.
A fork is a remote copy of a remote repo.
A pull request is mechanism for contributing code to a remote repo.

Refactoring

Refactoring: This process of improving a program's internal structure in small steps without modifying its external behavior is called refactoring.

- Not rewriting - needs to be done in small steps
- Not bug fixing - alter the external behavior

Should be followed by regression testing

Debugging

Debugging: The process of discovering defects in the program.
By inserting temporary print statements:

- Incurs extra effort when inserting and removing the print statements.
- Unnecessary program modifications increases the risk of introducing errors into the program.
- These print statements, if not promptly removed, may even appear unexpectedly in the production version.

By manually tracing through the code:

- It is difficult, time consuming, and error-prone technique.
- If you didn't spot the error while writing code, you might not spot the error when reading code too.

Using a debugger

Javadoc

Javadoc: A tool for generating API documentation in HTML format from doc comments in source.
Markdown
ASCiiDoc

Branching

Branching: The process of evolving multiple versions of the software in parallel.
A branch can be merged into another branch. -> result in a new commit
Merge conflicts happen when you try to merge two branches that had changed the same part of the code and the RCS software cannot decide which changes to keep.
Merge Types
Fast-forward merge - default
Move branch commits to master branch

Merge without fast-forward merge

Create a new commit to master branch

git merge --no-ff <branch>

Git does a fast forward merge if possible. -> Seeing that the master branch has not changed since you started the add-countries branch, Git has decided it is simpler to just put the commits of the add-countries branch in front of the master branch, without going into the trouble of creating an extra merge commit.

Base fork - where changes should be applied.

Head fork - changes you would like to be applied.

Switch to master branch before creating a new branch

Can fix by rebase/cherry-pick from the wrong branch to the correct one

Requirements

Requirement: specifies a need to be fulfilled by the software product.

brown-field project: replace/update

green-field project: totally new

Requirements come form stakeholders.

Stakeholder: A party that is potentially affected by the software project. e.g. users, sponsors, developers, interest groups, government agencies, etc.

Functional requirements

What the system should do.

Non-functional requirements

The constraints under which system is developed and operated.

1. Data requirements - size, volatility (how often do data change), persistency (saving data permanently)
2. Environment requirements
3. ……

Prioritizing requirements

Based the importance and urgency.

To gather requirements	
Brainstorming	A group activity designed to generate a large number of diverse and creative ideas for the solution of a problem.
	No bad ideas. The aim is to generate not to validate
Product Surveys	Studying existing products can unearth shortcomings of existing solutions that can be addressed by a new product.
Observation	Observing users in their natural work environment can uncover product requirements.
	Usage data of an existing system can also be used to gather information.
User Surveys	Surveys can be used to solicit responses and opinions from a large number of stakeholders
Interviews	Interviewing stakeholders and domain experts can produce useful information that project requirements.
	An expert of a discipline to which to product is connected.
	Focus groups are a kind of informal interview within an interactive group setting.

To gather requirements	
Focus groups	A group of people (e.g. potential users, beta testers) are asked about their understanding of a specific issue, process, product, advertisement, etc. - Lions meeting
Prototyping	Prototyping can uncover requirements, in particular, those related to how users interact with the system.
	UI prototypes are often used in brainstorming sessions, or in meetings with the users to get quick feedback from them.
	Can be used for discovering as well as specifying requirements

Prototype: A prototype is a mock up, a scaled down version, or a partial system constructed

- to get users' feedback.
- to validate a technical concept (a "proof-of-concept" prototype).
- to give a preview of what is to come, or to compare multiple alternatives on a small scale before committing fully to one alternative.
- for early field-testing under controlled conditions.

Techniques for Specifying Requirements

Prose: A textual description can be used to describe requirements. Especially useful when describing abstract ideas such as the vision of a product.

Feature List: A list of features of a product grouped according to some criteria such as aspect, priority, order of delivery, etc.

User story: User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

User story format: As a {user type/role} I can {function} so that {benefit}
The {benefit} can be omitted if it is obvious.
You can write user stories at various levels.
Epic (or Theme): high-level user stories.
You can add conditions of satisfaction to a user story.
Other useful info that can be added to a user story includes (but not limited to): priority, size, urgency

Usage
User stories capture user requirements in a way that is convenient for scoping(which features to include) , estimation(how much effort each feature will take) and scheduling(when to deliver).
User stories differ from traditional requirements specifications(prose) mainly in the level of detail. - less
User stories can capture non-functional requirements too.

Usage
User stories are quite handy for recording requirements during early stages of requirements gathering.

1. Define the target user: profile, work patterns
2. Define the problem scope: the exact problem
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

---

User case: A description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.

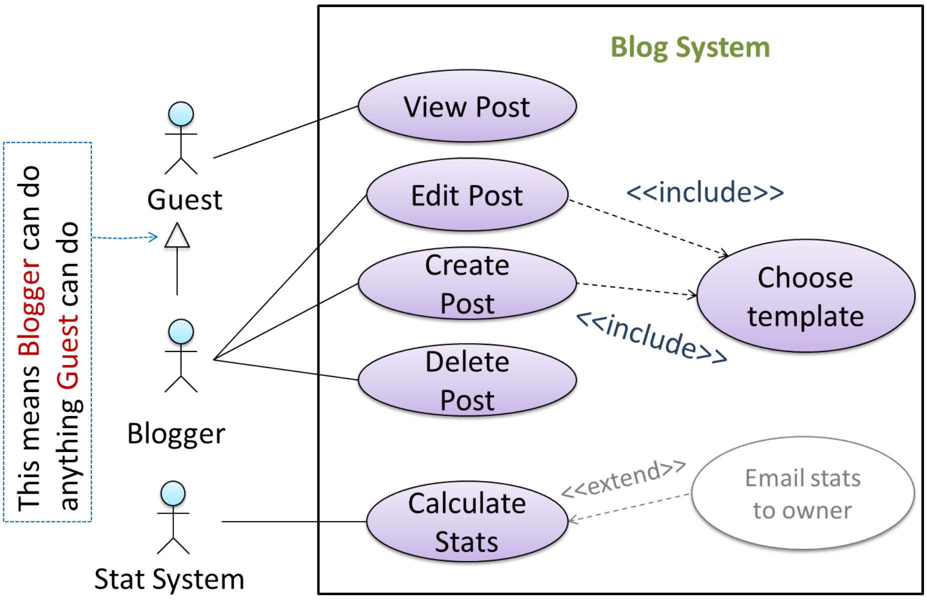
A use case describes an interaction between the user and the system for a specific functionality of the system.
UML includes a diagram type called use case diagrams that can illustrate use cases of a system visually.
Use cases capture the functional requirements of a system.
actor: An actor (in a use 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.
Many persons/systems can play a single role.
Use cases can be specified at various levels of detail.

Writing use case steps
The main body of the use case is the sequence of steps that describes the interaction between the system and the actors.
A use case describes only the externally visible behavior, not internal details, of a system.
A step gives the intention of the actor (not the mechanics).
The Main Success Scenario (MSS) describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong.
Extensions are "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 of the MSS. • the extension marked as *a. can happen at any step (hence, the *).
A use case can include another use case. Underlined text is commonly used.

In use case diagrams you can use the <<extend>> arrows to show extensions. Note the direction of the arrow is from the extension to the use case it extends and the arrow uses a dashed line.
We use a dotted arrow and a <<include>> annotation to show use case inclusions in a use case diagram. Note how the arrow direction is different from the <<extend>> arrows.

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.
You can use actor generalization in use case diagrams using a symbol similar to that of UML notation for inheritance.



Main Advantages
Because they use a simple notation and plain English descriptions, they are easy for users to understand and give feedback.
They decouple user intention from mechanism (note that use cases should not include UI-specific details), allowing the system designers more freedom to optimize how a functionality is provided to a user.
Identifying all possible extensions encourages us to consider all situations that a software product might face during its operation.
Separating typical scenarios from special cases encourages us to optimize the typical scenarios.

Main Disadvantages
Are not good for capturing requirements that does not involve a user interacting with the system. Use case are not very suitable for capturing non-functional requirements.

Glossary: A glossary serves to ensure that all stakeholders have a common understanding of the noteworthy terms, abbreviation, acronyms etc.

Supplementary Requirements: A supplementary requirements section can be used to capture requirements that do not fit elsewhere.
Typically, this is where most Non Functional Requirements will be listed.

JUnit Test

Delaying testing until the full product is complete has a number of disadvantages:
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
The delivery may have to be delayed

It’s better to do early testing.

Pros of developer testing	Cons of developer testing
• Can be done early (the earlier we find a bug, the cheaper it is to fix).	• A developer may subconsciously test only situations that he knows to work (i.e. test it too 'gently').
• Can be done at lower levels, for examples, at operation and class level (testers usually test the system at UI level).	• A developer may be blind to his own mistakes (if he did not consider a certain combination of input while writing code, it is possible for him to miss it again during testing).
• It is possible to do more thorough testing because developers know the expected external behavior as well as the internal structure of the component.	• A developer may have misunderstood what the SUT is supposed to do in the first place.
• It forces developers to take responsibility for their own work (they cannot claim that "testing is the job of the testers").	• A developer may lack the testing expertise.

Test Driver: the code that ‘drives’ the SUT for the purpose of testing.
JUnit is a toll for automated testing of Java programs.
whatIsBeingTested_descriptionOfTestInputs_expectedOutcome

Product Design

1. Value to users
2. Minimize work for users
3. Match user intent
4. Less is more (can give choice but not default, shoule provide good default and allow change)
5. Don't force to RTFM (read the fucking manual)
6. Don't make users feel stupid
7. Benefits, not features (only beneficial features are needed)
8. Be everything to somebody (not something to everybody - narrow down)
9. Visualize usage
10. Care

Models

Models: A representation of something else.

Abstractions <- A model provides a simpler view of a complex entity because a model captures only a selected aspect.
Multiple models of the same entity may be needed to capture it fully.

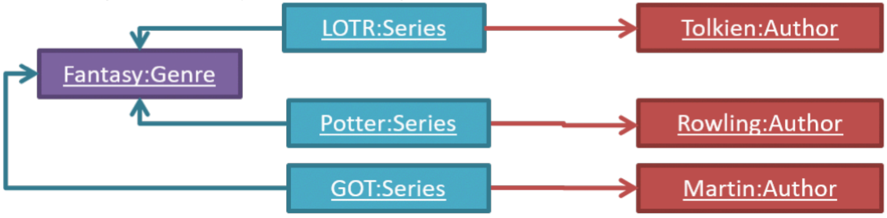
In software development, models are useful in several ways:	
For analysis	To analyze a complex entity related to software development
For communication - common	To communicate information among stakeholders.
As a blueprint	model-driven development

OO Structures

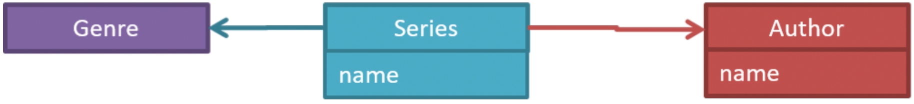
OO solutions - a network of objects interacting with each other.
Object structures within the same software can change over time.

- based on a set of rules, which was decided by the designer of that software.
    - Be illustrated as a class structure

UML Object Diagrams - object structures

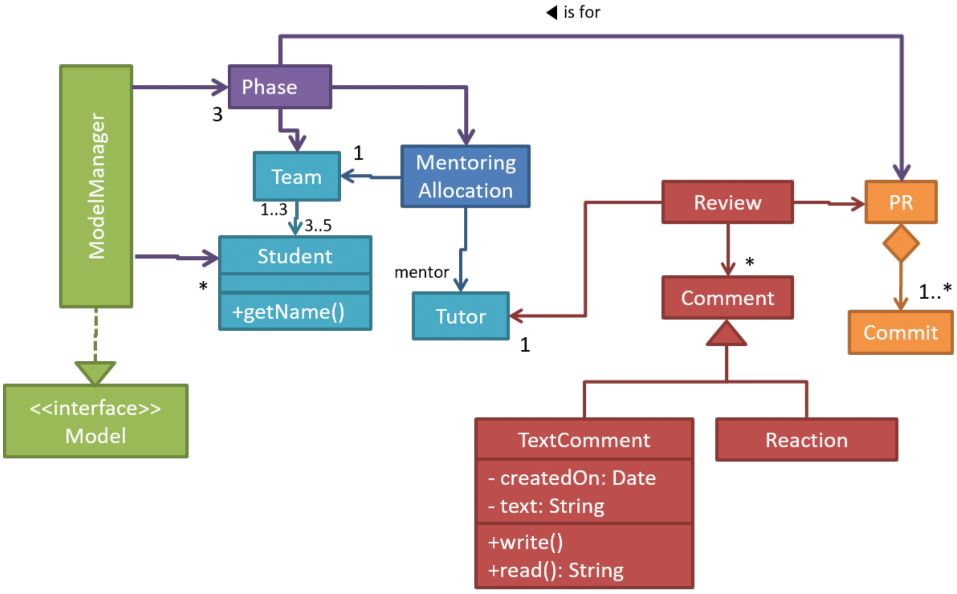


UML Class Diagrams - class structures

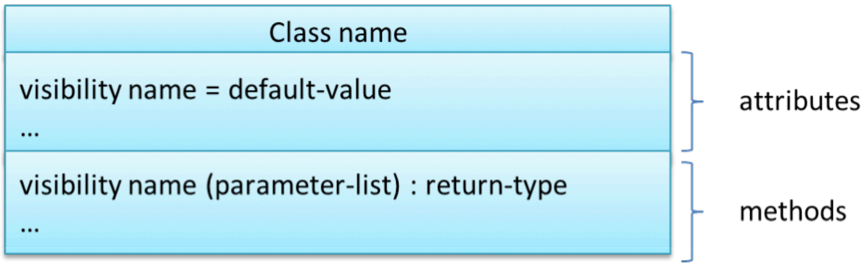


Class Diagram

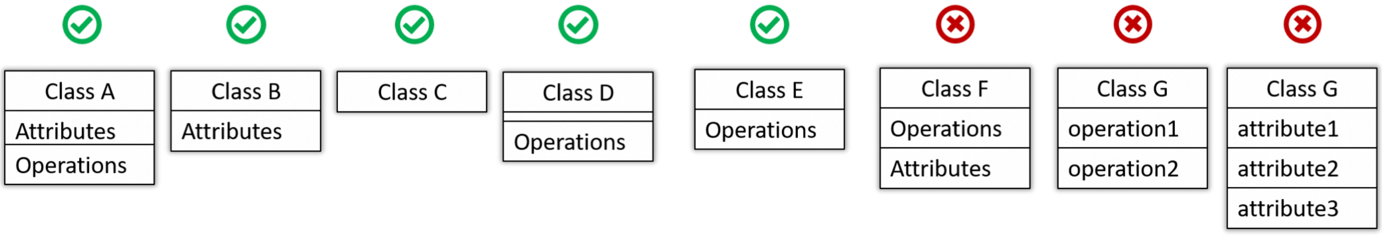
UML class diagrams describe the structure (not the behavior) of an OOP solution.
A model that represents a software design



Classes: UML notions used to present a class

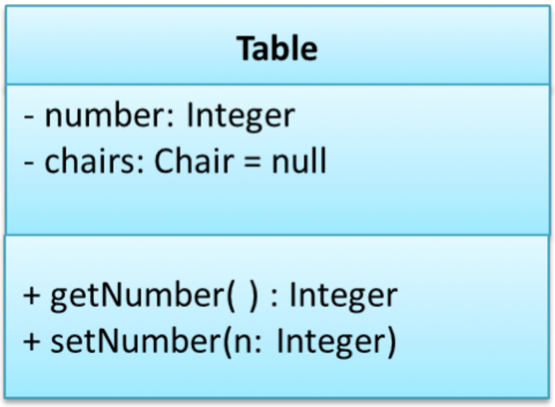

Class name
visibility name = default-value
...
visibility name (parameter-list) : return-type
...
attributes
methods

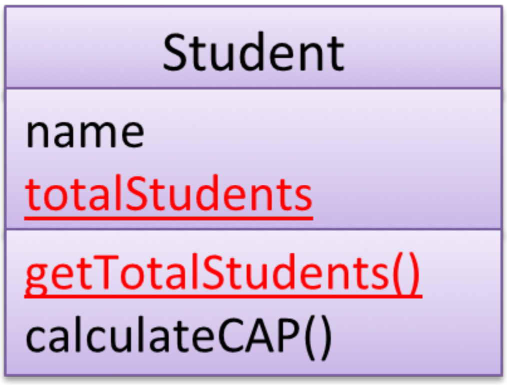
The 'Operations' compartment and/or the 'Attributes' compartment may be omitted if such details are not important for the task at hand.



Visibility: The visibility of attributes and operations is used to indicate the level of access allowed for each attribute or operation.

visibility	Java	Python
- private	private	at least two leading underscores (and at most one trailing underscores) in the name
#protected	protected	one leading underscore in the name
+ public	public	all other cases
~package private	default visibility	not applicable



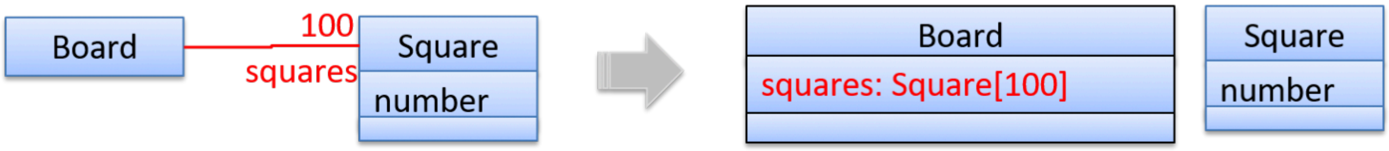


Underlines denote class-level attributes and variables.

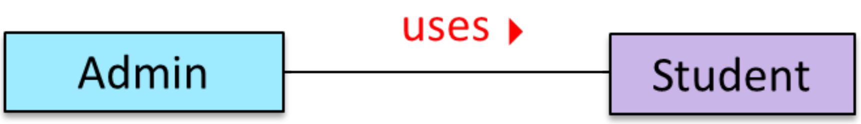
Associations: The main connections among the classes in a class diagram

Solid line - association
Dashed line - dependency
We use instance level variables to implement associations.

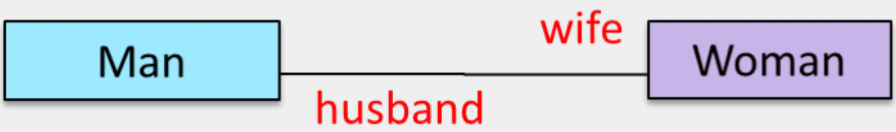
An association can be shown as an attribute instead of a line.
name: type [multiplicity] = default value



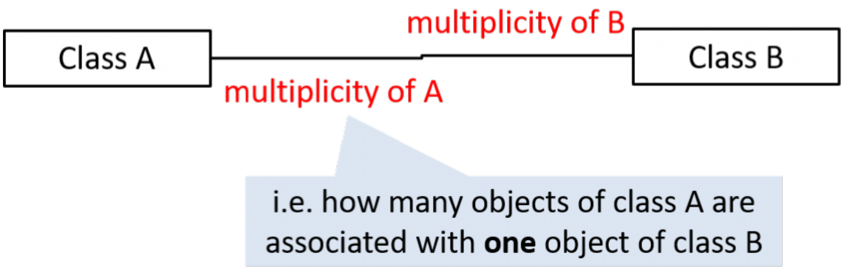
Association labels: describe the meaning of the association.



Association Role: indicate the role played by the classes in the association.



Multiplicity: dictates how many objects take part in each association.

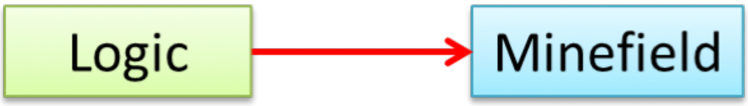


i.e. how many objects of class A are associated with **one** object of class B

- 0..1 : optional, can be linked to 0 or 1 objects.
- 1 : compulsory, 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 n to m inclusive

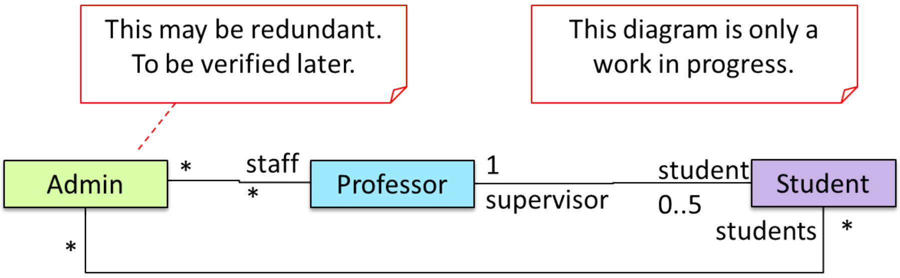
Navigability: The concept of which class in the association knows about the other.

Arrow head - navigability
Logic is aware of Minefield, but Minefield is not aware of Logic



UML notes: augment UML diagrams with additional information.

with/without connection - notes

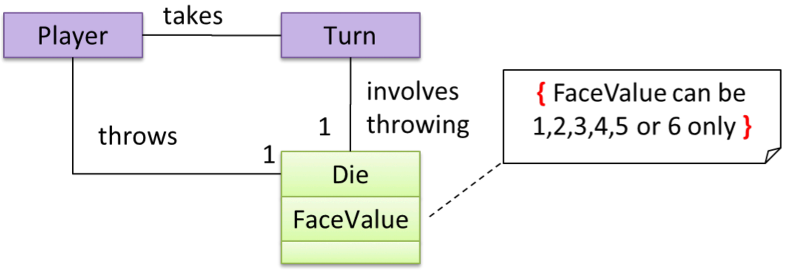


This may be redundant. To be verified later.

This diagram is only a work in progress.

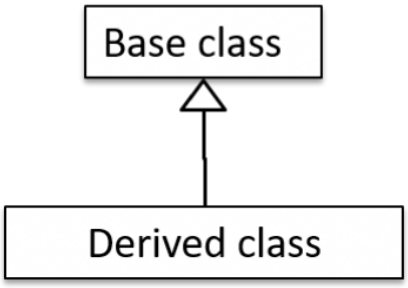
Constraints

inside a note, curly braces - constraints



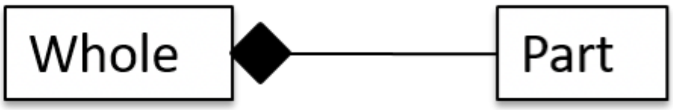
Inheritance: allows you to define a new class based on an existing class.

A triangle and a solid line
A superclass is said to be more general than the subclass.



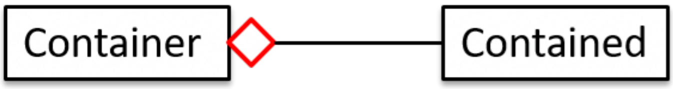
Composition: represents a strong whole-part relationship

A solid diamond symbol - composition - cannot exist without it
If whole is deleted, parts cannot exist
Don't use cyclical links

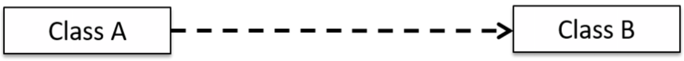


Aggregation: represents a container-contained relationship

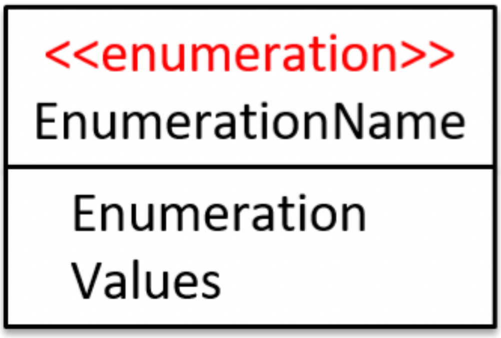
A hollow diamond symbol - ~~aggregation~~ - contain but don't depend on it
If container is deleted, contained items can exist as well



Dependency: a need for one class to depend on another without having a direction association with it

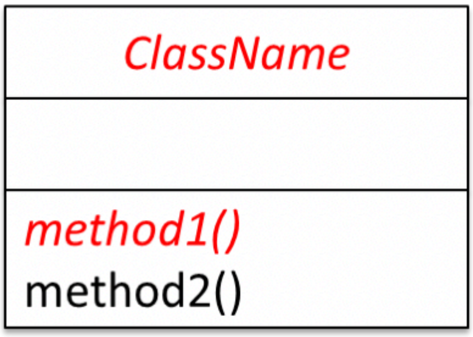


Enumeration: a fixed set of values that can be considered as a data type

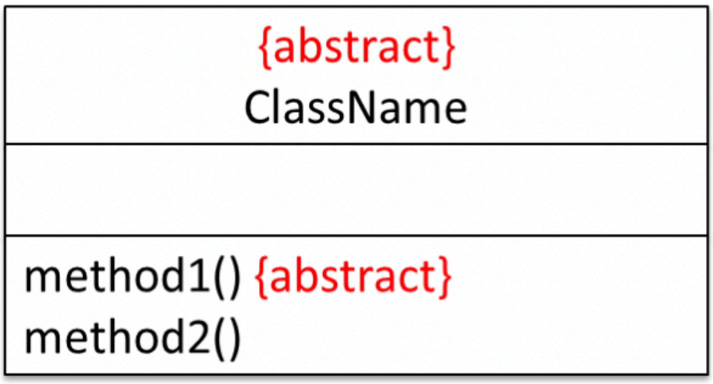


Abstract: a class is merely a representation of commonalities among its subclasses

Italics or {abstract} - abstract classes/methods

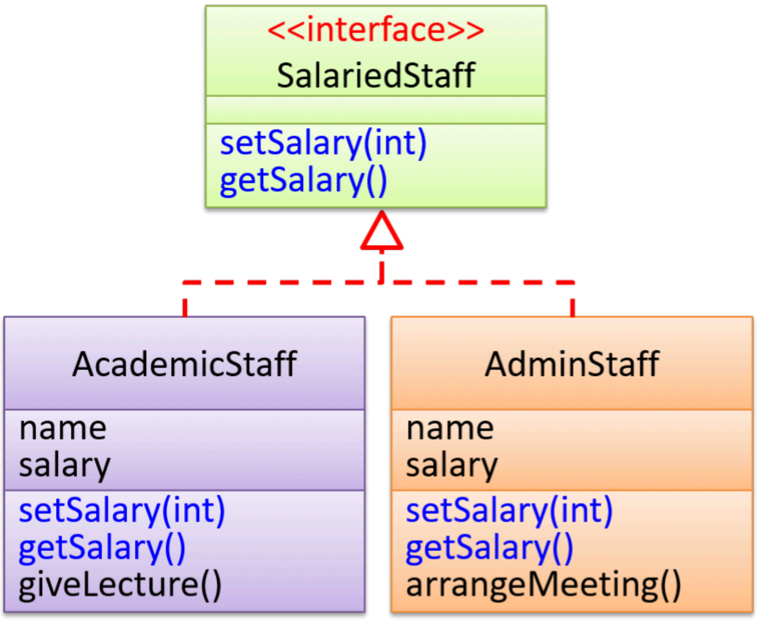


OR



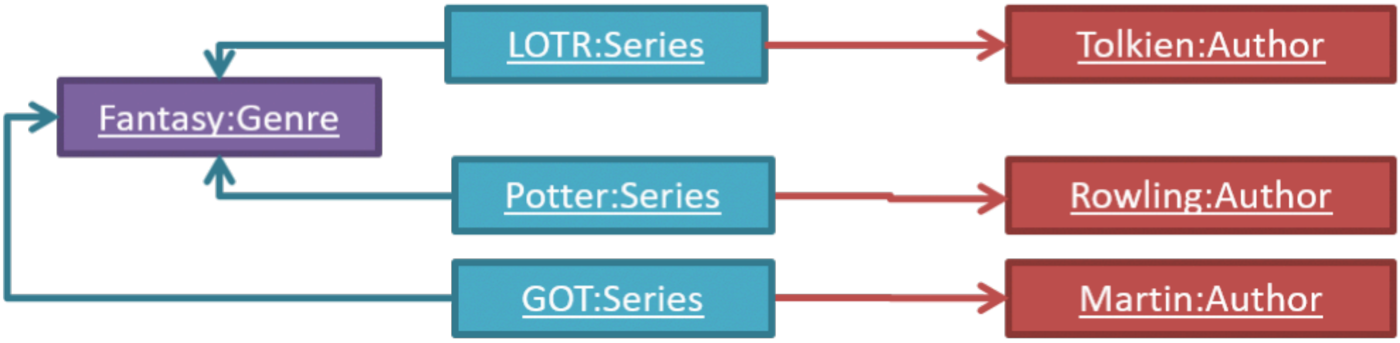
Interface: a behavior specification

Similar to class inheritance except a dashed line is used instead of a solid line
Java allows multiple inheritance between interface
Java allows implementing different interfaces



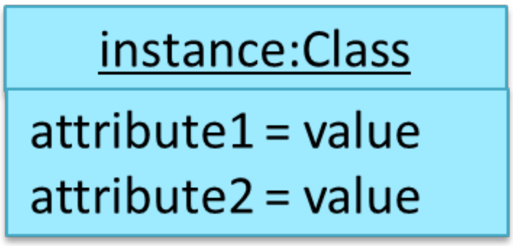
Object Diagrams

Object Diagram: Shows an object structure at a given point of time.

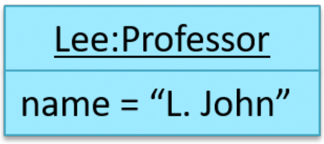


Need to follow the class diagrams

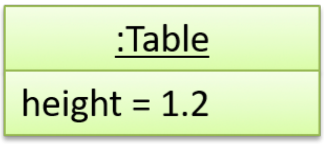
Objects



- 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'.
- No compartment for methods, multiplicities are omitted.
- Attributes compartment can be omitted.
- Object name can be omitted



Jon:Student



Sequence Diagrams

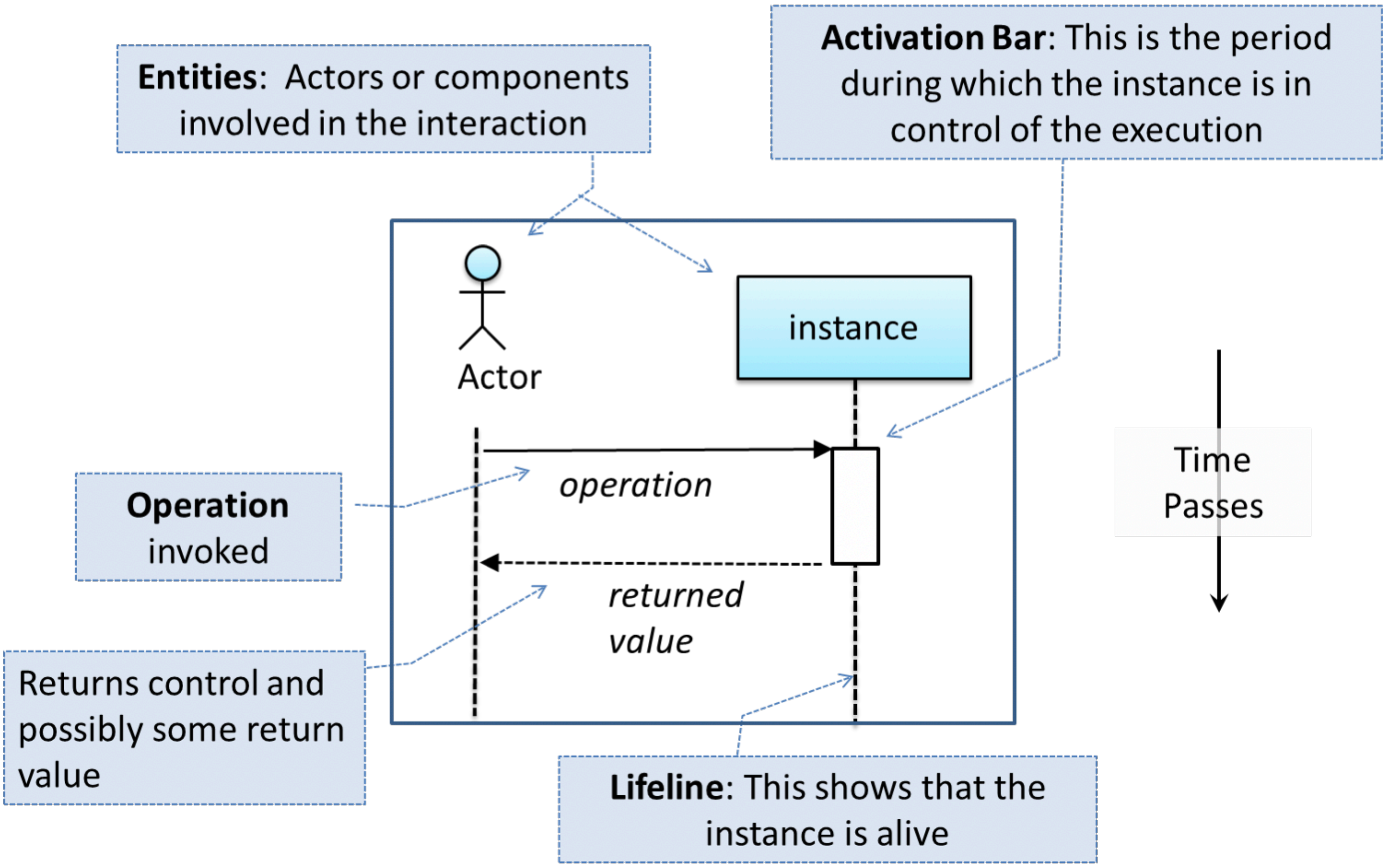
Sequence Diagram: A UML sequence diagram captures the interactions between multiple objects for a given scenario.

Method calls - solid arrows

Method returns - dashed arrows

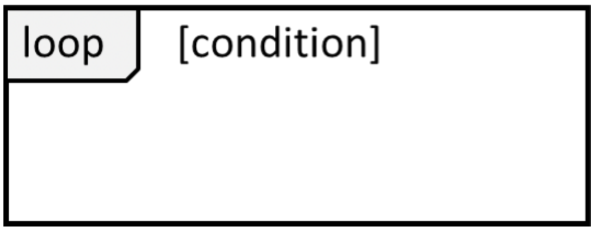
Class/object name is not underlined

Activation bars and return arrows may be omitted

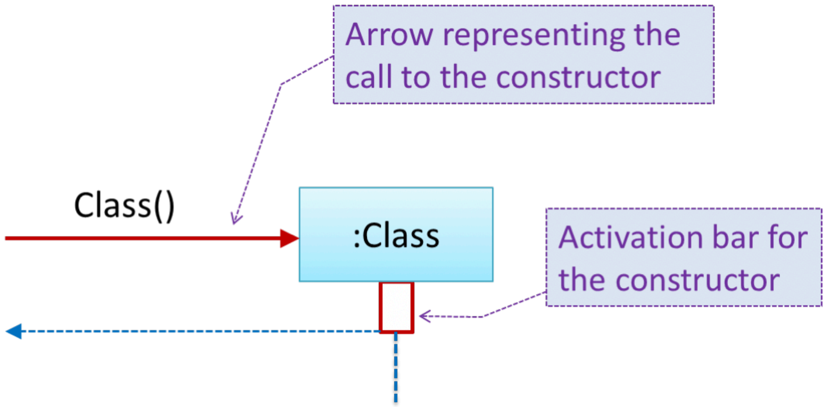


Entities: Actors or components involved in the interaction
Activation Bar: This is the period during which the instance is in control of the execution
Actor
instance
Time Passes
Operation invoked
operation
returned value
Returns control and possibly some return value
Lifeline: This shows that the instance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

Loop

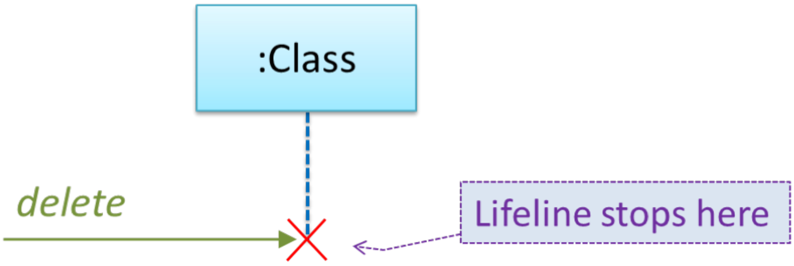


Object creation



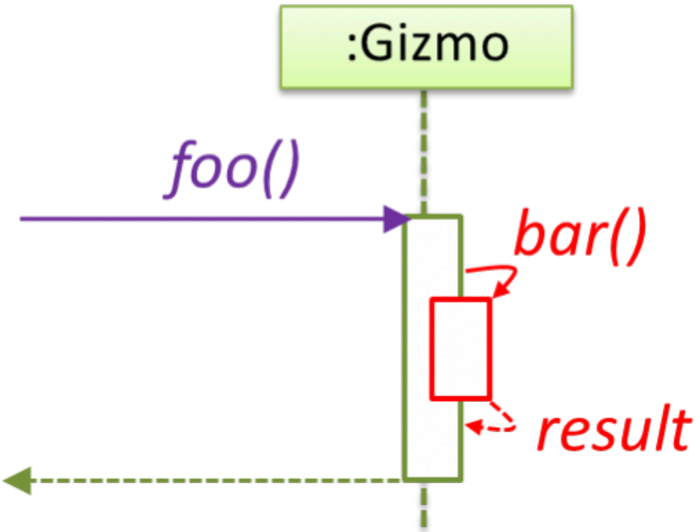
Arrow representing the call to the constructor
Class()
:Class
Activation bar for the constructor

Deletion - X

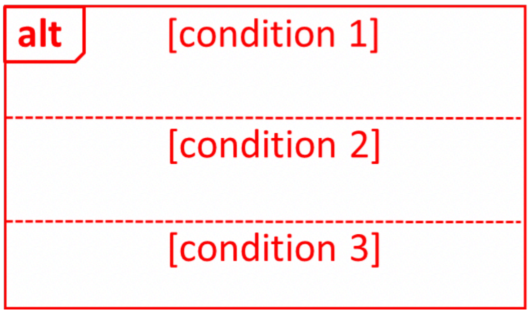


Self-invocation: a method of an object calling another of its own methods

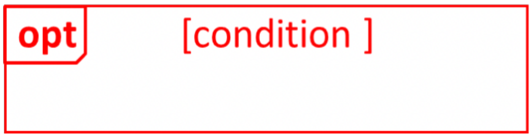
An object calling another of its own methods



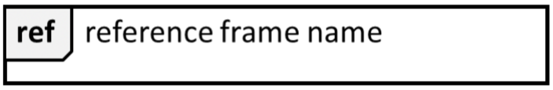
Alternative paths - alt frame

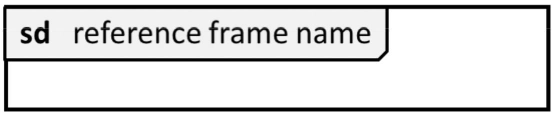


Optional paths - opt frame

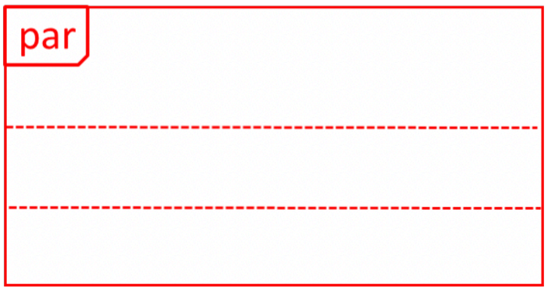


Reference frames - ref frame: Allow a segment of the interaction to be omitted and shown as a separate sequence diagram





Parallel path - par frames



Architecture Diagram

Design: in the creative process of transforming the problem into a solution; the solution is also called design.

Software design has two main aspects:	
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.

Architecture Diagram - free-form diagrams

Logging

Logging: the deliberate recording of certain information during a program execution for future reference.

Typically written to a log file but other ways are possible as well.

Don't prevent problems but can be helpful in understanding.

```
import java.util.logging.*;
private static Logger logger = Logger.getLogger("Foo");
```

When running the code, the logging level can be set to WARNING so that log messages specified as INFO level (which is a lower level than WARNING) will not be written to the log file at all.

```
// log a message at INFO level
logger.log(Level.INFO, "going to start processing");
//... processInput();
if(error){
//log a message at WARNING level logger.log(Level.WARNING, "processing error",
ex);
}
//...
logger.log(Level.INFO, "end of processing");
```

Assertions

Assertions: define assumptions about the program state so that the runtime can verify them.
Exceptions - user issue
Assertions - code bug
Take drastic action, e.g. terminating the execution with an error message
This assertion will fail with the message x should be 0 if x is not 0 at this point.

```
    x = getX();
    assert x == 0 : "x should be 0";
```

`java -enableassertions HelloWorld` (or `java -ea HelloWorld)` will run `HelloWorld` with assertions enabled while `java -disableassertions HelloWorld` will run it without verifying assertions.
Java disables assertions by default.
Java `assert` vs JUnit assertions: They are similar in purpose but JUnit assertions are more powerful and customized for testing.
JUnit assertions are not disabled by default.
Assertions are suitable for verifying assumptions about Internal Invariants, Control-Flow Invariants, Preconditions, Postconditions, and Class Invariants.

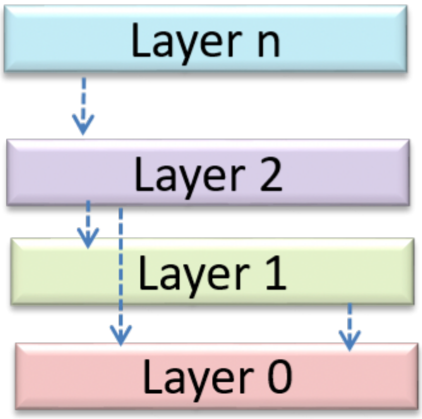
Continuous Integration and Continuous Deployment

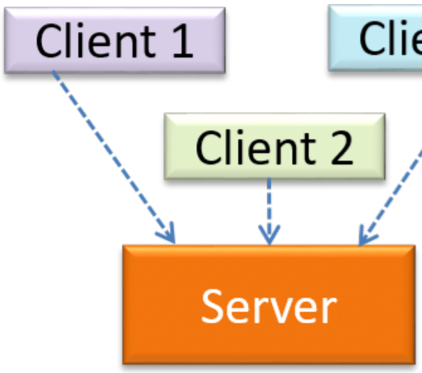
Integration: Combining parts of a software product to form a whole.
Build automation tools automate the steps of the build process, usually by means of build scripts.
Some build tools also serve as dependency management tools.
Continuous Integration: An extreme application of build automation is called continuous integration (CI) in which integration, building, and testing happens automatically after each code change.
Continuous Deployment: A natural extension of CI is Continuous Deployment (CD) where the changes are not only integrated continuously, but also deployed to end-users at the same time.

Architectural Styles

What are used by AB-4: Event-driven, Client-server

N-tier architectural style: Higher layers make use of services provided by lower layers. Lower layers are independent of higher layers

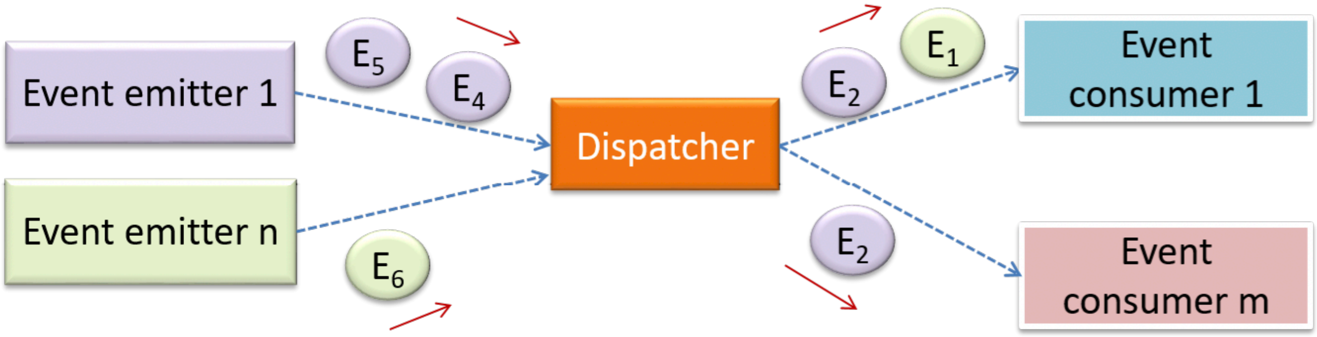




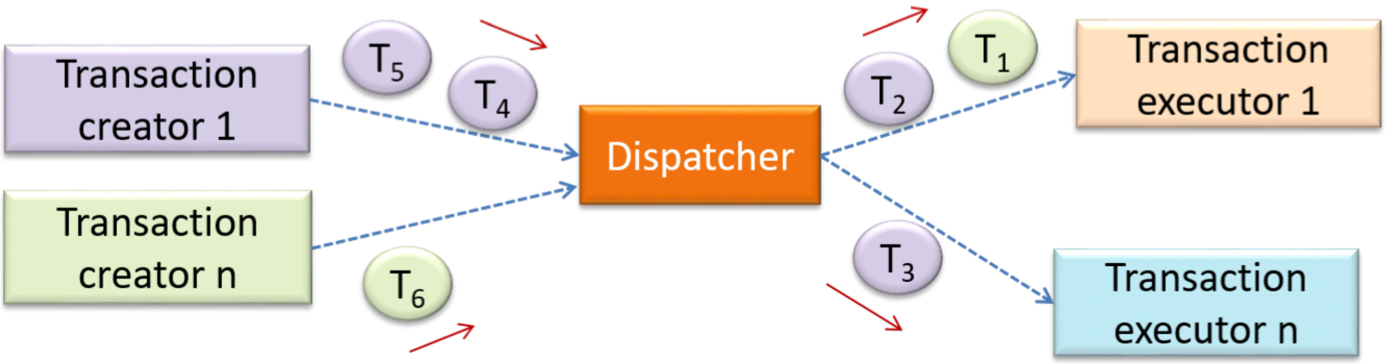
Client-server architectural style: has at least one component playing the role of a server and at least one client component accessing the services of the server.
e.g. Google

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

e.g. GUI



Transaction processing architectural 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.



Service-oriented architectural (SOA) style: Builds applications by combining functionalities packaged as programmatically accessible services.

Achieve interoperability between distributed services (no need be implemented using the same programming language)

Pipes-and-filters

Message-driven

Broker style

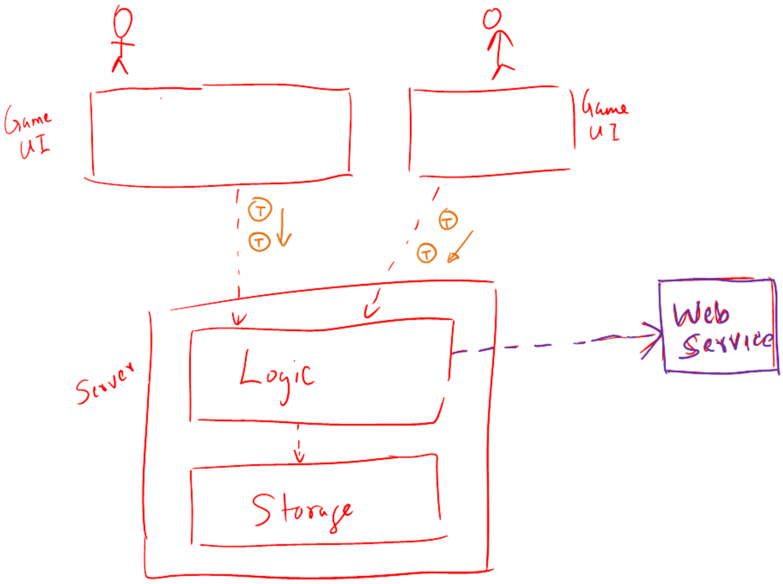
P2P

E1

Assume you are designing a multiplayer version of the Minesweeper game where any number of players can play the same Minefield. Players use their own PCs to play the game. A player scores by deducing a cell correctly before any of the other players do. Once a cell is correctly deduced, it appears as either marked or cleared for all players.

Comment on how each of the following architectural styles could be potentially useful when designing the architecture for this game.

1. Client-server – Clients can be the game UI running on player PCs. The server can be the game logic running on one machine.
2. Transaction-processing – Each player action can be packaged as transactions (by the client component running on the player PC) and sent to the server. Server processes them in the order they are received.
3. SOA – The game can access a remote web services for things such as getting new puzzles, validating puzzles, charging players subscription fees, etc.
4. Multi-layer – The server component can have two layers: logic layer and the storage layer.



APIs

API: An Application Programming Interface (API) specifies the interface through which other programs can interact with a software component.

A collection of public methods / a collection of Web request formats Github server accepts and the corresponding responses

Need not known implementation

How is it used in AB-4: javadoc is

Code Readability

Code quality: run-time efficiency, security, robustness, understandability

- Avoid long methods
- Avoid deep nesting - ≤ 3 levels of indentation
- Avoid complicated expressions - do in steps
- Avoid magic numbers - named constant

unused parameters in the method signature
similar things look different
different things that look similar
multiple statements in the same line
data flow anomalies such as, pre-assigning values to variables and modifying it without any use of the pre-assigned value

- Make the code obvious - do explicit instead of implicit

We may not know which parts are the real performance bottlenecks.
Optimizing can complicate the code
Hand-optimized code can be harder for the compiler to optimize

- Structure code logically
- Don't trip up reader
- Practice KISSing "keep it simple, stupid"
- Avoid premature optimizations
- SLAP hard - avoid varying the level of abstraction within a code fragment
- Make the Happy Path Prominent - guard clauses

Good naming:

- nouns - things, verbs - actions
- Use standard words
- Use name to explain the named entity accurately and at a sufficient level of detail
- Not too long, not too short - explain abbreviate or acronyms obviously
- Avoid misleading or ambiguous names - related things should be named similarly, avoid multiple meanings, similar sounding names, hard-to-pronounce ones, LI100

Avoid Unsafe Coding Practices

- Use the default branch - else means everything else
- Don't recycle variables or parameters
- Avoid empty catch blocks
- Delete dead code
- Minimise scope of variable - global variables do create implicit links between code segments that use the global variable
- Minimise code duplication

Software Design Principles

Abstraction: a technique for dealing with complexity. It works by establishing a level of complexity we are interested in, and suppressing the more complex details below that level.

Guiding principle: only details that are relevant to the current perspective or the tasks at hand needs to be considered
Data abstraction: ignoring lower level data items and thinking in terms of bigger entities
Control abstraction: abstracts away details of the actual control flow to focus on tasks at a simplified level.

- An OOP class is an abstraction over related data and behaviors.
- An architecture is a higher-level abstraction of the design of a software.
- Models (e.g., UML models) are abstractions of some aspect of reality.

Coupling: a measure of the degree of dependence between components, classes, methods, etc.

High coupling is discouraged:
Maintenance is harder: a change in one module could cause changes in other modules
Integration is harder: multiple components coupled with each other have to be integrated at the same time
Testing and reuse of the module is harder: due to its dependence on other modules

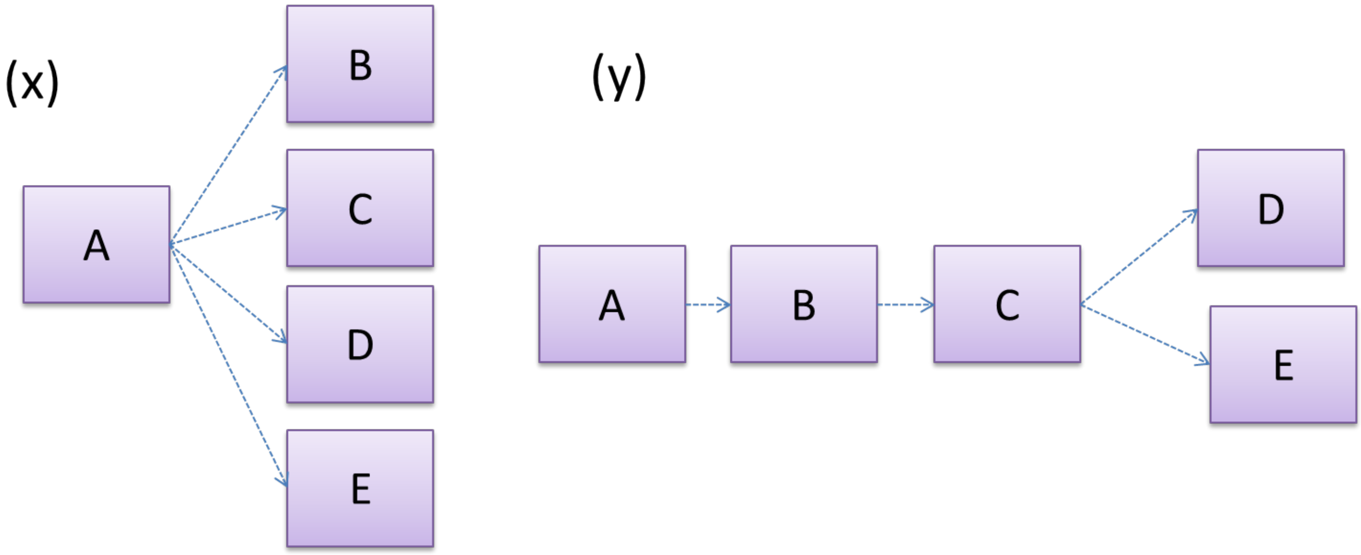
X is coupled to Y if a change to Y can potentially(not always) require a change in X.
Some examples of coupling: A is coupled to B if,

- 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

Types of coupling	
Content coupling	One module modifies or relies on the internal workings of another module
Common/Global coupling	Two modules share the same global data
Control coupling	One module controlling the flow of another, by passing it information on what to do
Data coupling	One module sharing data with another module
External coupling	Two modules share an externally imposed convention
Subclass coupling	A class inherits from another class. A child class is coupled to the parent class
Temporal coupling	Two actions are bundled together just because they happen to occur at the same time

E1

Discuss the coupling levels of alternative designs x and y.



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.

Cohesion: Cohesion is a measure of how strongly-related and focused the various responsibilities of a component are.

Higher cohesion is better.

Disadvantages of low 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 many modules may need to be modified to achieve a small change in behavior (reason: because the code related to that change is not localized to a single module).
Lowers reusability of modules because they do not represent logical units of functionality.

Represent in many forms
Code related to a single concept is kept together
Code that is invoked close together in time is kept together
Code that manipulates the same data structure is kept together

Single Responsibility Principle (SRP): A class should have one and only one reason to change

It needs to change only when there is a change to that responsibility.

Open-Close Principle (OCP): Aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself

This often requires separating the specification (i.e. interface) of a module from its implementation.
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.

Separation of Concerns Principle (SoC): To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate concern.

Concern: a set of information that affects the code of a computer program.
Reduce functional overlaps among code sections and also 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.

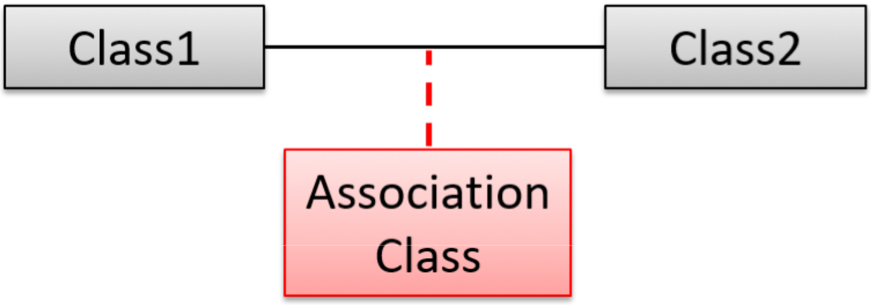
Integration Approaches

Timing and Frequency	Late and one-time: wait till all components are completed and integrate all finished components near the end of the project. • many component incompatibilities (due to previous miscommunications and misunderstandings) to surface which can lead to delivery delays
	Early and frequent: integrate early and evolve each part in parallel, in small steps, re-integrating frequently.
Amount Merged at A Time	Big-bang integration: integrate all components at the same time. • uncover too many problems at the same time which could make debugging and bug-fixing more complex
	Incremental integration: integrate few components at a time.
The Order in which Components are Integrated	Top-down integration: higher-level components are integrated before bringing in the lower-level components. • higher-level problems can be discovered early • requires the use of stubs in place of lower level components • higher-level components cannot function as they depend on lower level ones
	Bottom-up integration: the reverse of top-down integration. • drivers may be needed to test the integrated components

	Sandwich integration: a mix of the top-down and the bottom-up approaches.

Association Classes

An association class represents additional information about an association.



Testing

Unit testing: testing individual units (methods, classes, subsystems, ...) to ensure each piece works correctly.

A proper unit test requires the unit to be tested in isolation
Stubs can isolate the SUT(Software Under Test) from its dependencies.
Stub: A stub has the same interface as the component it replaces, but its implementation is so simple that it is unlikely to have any bugs. It mimics the responses of the component, but only for the a limited set of predetermined inputs. That is, it does not know how to respond to any other inputs. Typically, these mimicked responses are hard-coded in the stub rather than computed or retrieved from elsewhere, e.g. from a database. - inject stubs
Dependency injection: The process of 'injecting' objects to replace current dependencies with a different object.
Polymorphism can be used to implement dependency injection.

Integration testing: testing whether different parts of the software work together (i.e. integrates) as expected.

Run using the actual dependencies, focus on the interactions between the parts
Not necessary different classes, more like different components

System testing - use Gradle: take the whole system and test it against the system specification.

System test cases are based on the specified external behavior of the system

Includes testing against non-functional requirements too
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.

Includes testing against non-functional requirements too
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.

One approach to overcome the challenges of testing GUIs is to minimize logic aspects in the GUI.

Acceptance testing/User Acceptance Testing(UAT) - Dogfooding(be your own user):

test the delivered system to ensure it meets the user requirements.

Often defined at the beginning of the project, usually based on the use case specification

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.

Passing system tests does not necessarily mean passing acceptance testing:
Different environment, flaws in the system design

Alpha & Beta Testing

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.

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

Different coverage criteria	
Funcion/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 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.
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	possible paths through a given part of the code executed. -> commonly used notation for path analysis: Control Flow Graph(CFG)
Entry/exit coverage	possible calls to and exits from the operations in the SUT.

Highest intensity of testing: 100% path coverage

Measuring coverage is often done using coverage analysis tools.

Test-Driven Development(TDD): Writing the tests before writing the SUT, but not all, while evolving functionality and tests in small increments.

Scheduling and Tracking Tools

Milestone: The end of a stage which indicates a significant progress.

Buffer: A time set aside to absorb any unforeseen delays.

Do not inflate task estimates to create hidden buffers; have explicit buffers instead.

Reason: With explicit buffers it is easier to detect incorrect effort estimates which can serve as a feedback to improve future effort estimates.

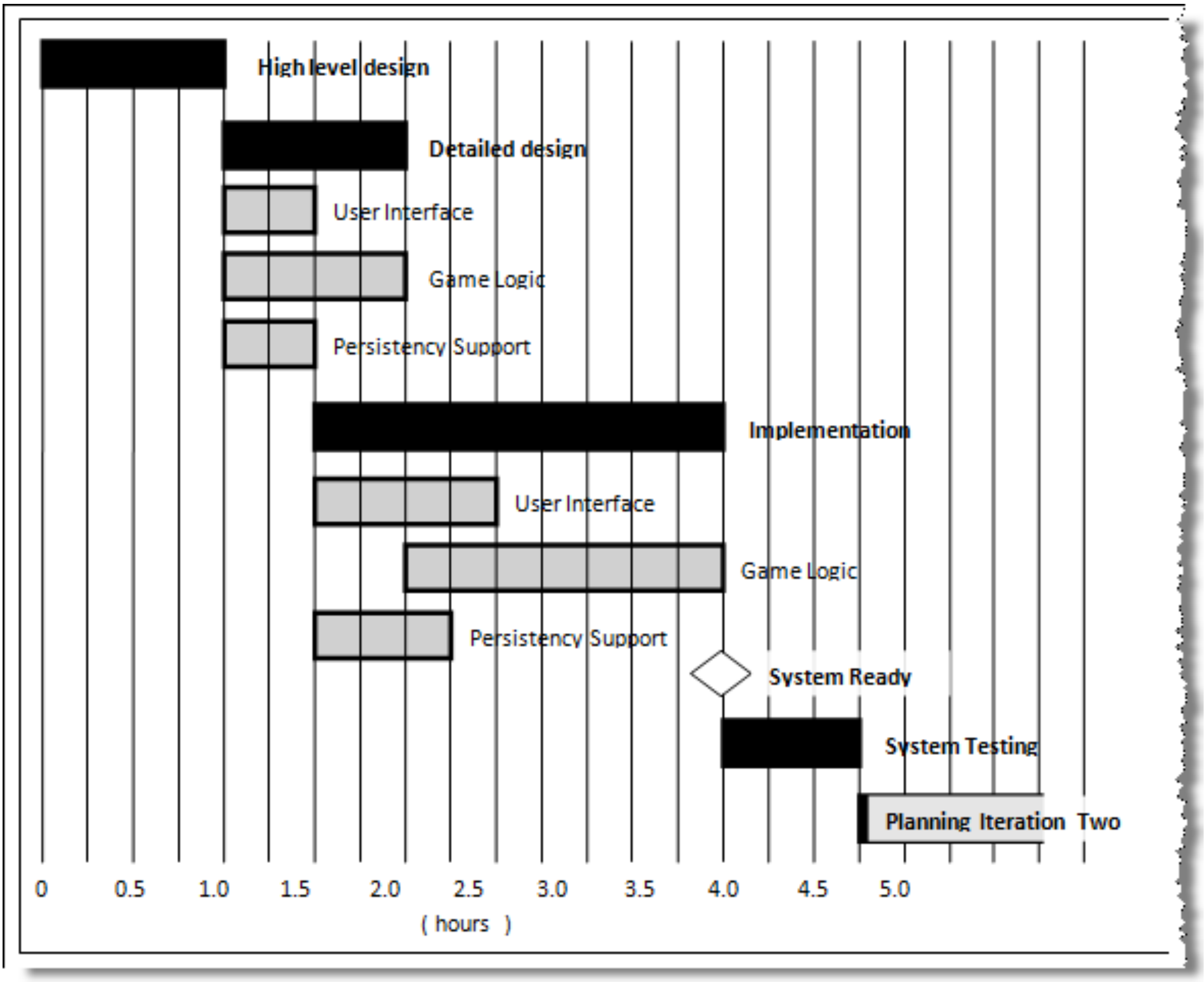
Issue Tracker (Bug Tracker): Track task assignment and progress.

Work Breakdown Structure (WBS): Depicts information about tasks and their details in terms of subtasks

The effort is traditionally measured in man hour/day/month

All tasks should be well-defined.

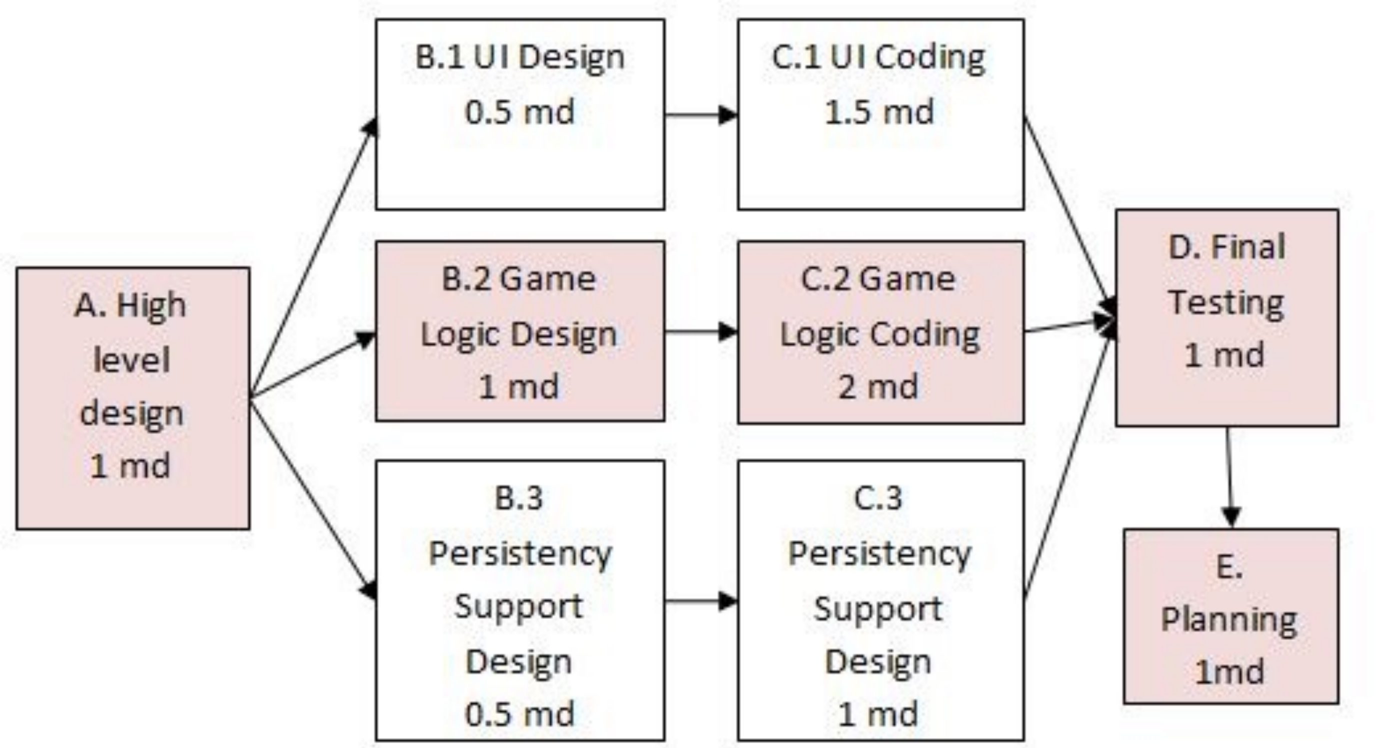
GANTT Charts: 2-D bar-chart, drawn as time vs tasks



Solid bar - main task
Grey bar - subtasks
Diamond - an important ddl/deliverable/milestone

Program Evaluation Review Technique (PERT) Charts: Uses a graphical technique to show the order/sequence of tasks

Node/vertex - the effort estimation of a task
Arrow - the precedence between tasks


A. High level design 1 md
B.1 UI Design 0.5 md
B.2 Game Logic Design 1 md
B.3 Persistency Support Design 0.5 md
C.1 UI Coding 1.5 md
C.2 Game Logic Coding 2 md
C.3 Persistency Support Design 1 md
D. Final Testing 1 md
E. Planning 1md

A PERT chart can help determine:
The order of tasks
Which tasks can be done concurrently
The shortest possible completion time
The critical path Critical path is the path in which any delay can directly affect the project duration. It is important to ensure tasks on the critical path are completed on time.

Team Structures

Egoless Team / democratic team structure	every team member is equal in terms of responsibility and accountability.	• finds a good solution to a relatively hard problem as all team members contribute ideas. • the democratic nature of the team structure bears a higher risk of falling apart due to the absence of an authority figure to manage the team and resolve conflicts.
Chief programmer team	there is a single authoritative figure, the chief programmer.	• The success of such a team structure relies heavily on the chief programmer.
Strict hierarchy team	a strictly defined organization among the team members	• In a large, resource-intensive, complex project, this could be a good team structure to reduce communication overhead.

Basic Design Approaches

Top-down and bottom-up design: 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
Big and novel systems
High-level design needs to be stable before lower levels can be designed

- Bootom-up

Design lower level components first and put them together to create the higher-level systems later
Design a variations of an existing system or re-purposing existing components to build a new system

- Mix

Agile design

Some initial architectural modeling at the very beginning, but it's just enough to get your team going.
Doesn't produce a fully documented set of models in place before you may begin coding.

Intermediate-level design principles

Polymorphism

Substitutability: Every instance of a subclass is an instance of the superclass, but not vice-versa.
the ability to substitute a child class object where a parent class object is expected.
Dynamic binding: a 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: When a method call is resolved at compile time.
Overloaded methods are resolved using static binding.

To achieve polymorphism	
Substitutability	Treat objects of different types as one type
Overriding	Allows objects of different subclasses to display different behaviors in response to the same method call
Dynamic binding	Call the method of the parent class and yet execute the implementation of the child class

Liskov Substitution Principle: 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.

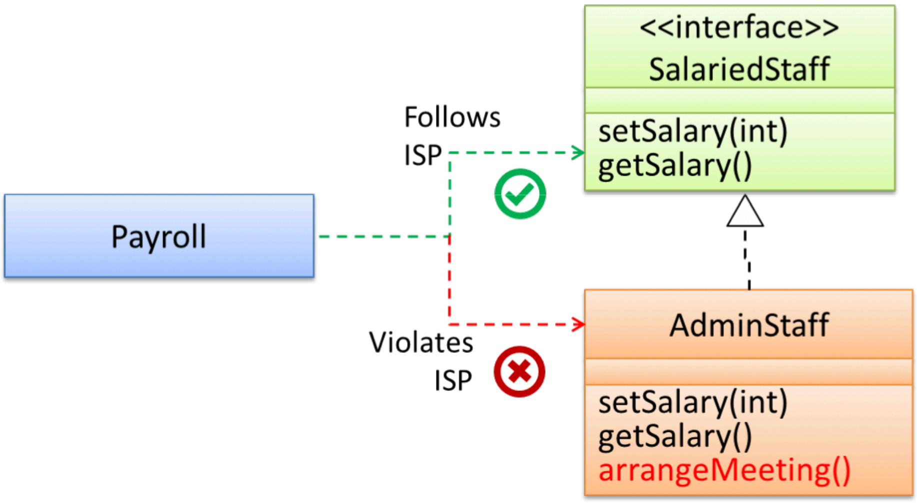
Law of Demeter (LoD) - principle of less knowledge, don’t talk to stranger

- An object should have limited knowledge of another object.
- An object should only interact with objects that are closely related to it.

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 (directly or indirectly)
• Objects from the direct association of O

Direct association of: objects that are held by instance variables of
LoD aims to prevent objects navigating internal structures of other objects.
Reduce coupling by limiting the interaction to a closely related group of classes.

Interface Segregation Principle (ISP): No client should be forced to depend on methods it does not use.



Dependency Inversion Principle (DIP)

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Inheritance is a dependency

√ a. It can complicate the design/implementation by introducing extra abstractions, but it has some benefits.
b. It is often used during testing, to replace dependencies with mocks.
c. It reduces dependencies in a design.
d. It advocates making higher level classes to depend on lower level classes.

Explanation: Replacing dependencies with mocks is Dependency Injection, not DIP. DIP does not reduce dependencies, rather, it changes the direction of dependencies. Yes, it can introduce extra abstractions but often the benefit can outweigh the extra complications.

SOLID Principles

Single Responsibility Principle (SRP)
Open-Closed Principle (OCP)
Liskov Substitution Principle (LSP)
Interface Segregation Principle (ISP)
Dependency Inversion Principle (DIP)

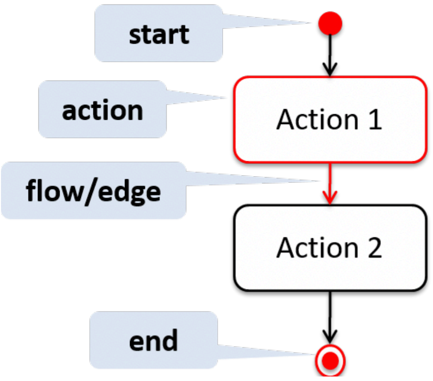
You Aren't Gonna Need It! (YAGNI) Principle: Do not add code simply because 'you might need it in the future'.

Don't Repeat Yourself (DRY) Principle: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

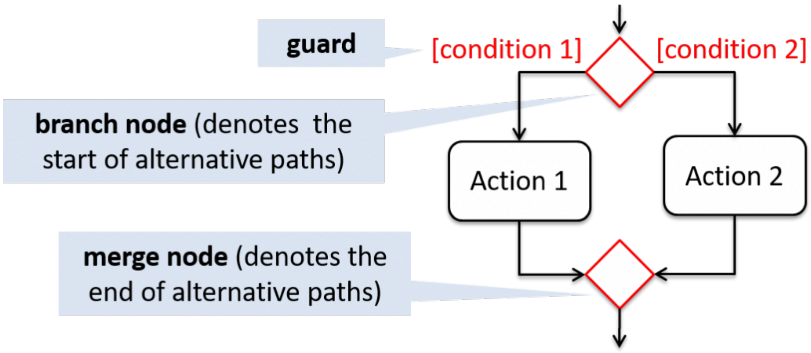
Brooks' Law: Adding people to a late project will make it later.

Activity diagrams

UML activity diagrams (AD): can model workflows.
Captures an activity of actions and control flows that makes up the activity
Action - single step - rectangle with rounded corners
Control flow - the flow of control from one action to the next - a line with an arrow-head



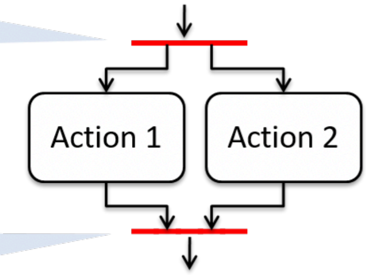
Branch node - the start of alternate paths - diamond shapes
Merge node - the end of alternate paths - diamond shapes
Guard conditions - in square brackets: a boolean condition that should be true for execution to take that path. Only one of the guard condition can be true at any time.


guard
[condition 1]
[condition 2]
branch node (denotes the start of alternative paths)
Action 1
Action 2
merge node (denotes the end of alternative paths)

Fork node - the start of concurrent flows of control - bar
Join node - the end of parallel paths - bar

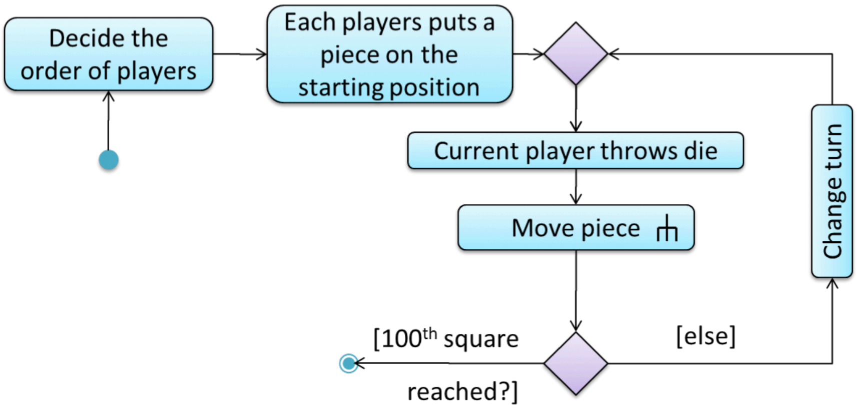
Fork (denotes the start of parallel paths - many outgoing edges)

Join (denotes the end of parallel paths – many incoming edges)

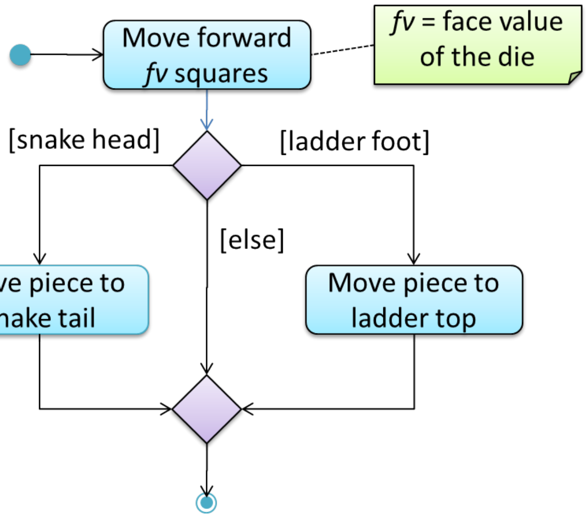


Rake notation - a part of the activity is given as a separate diagram

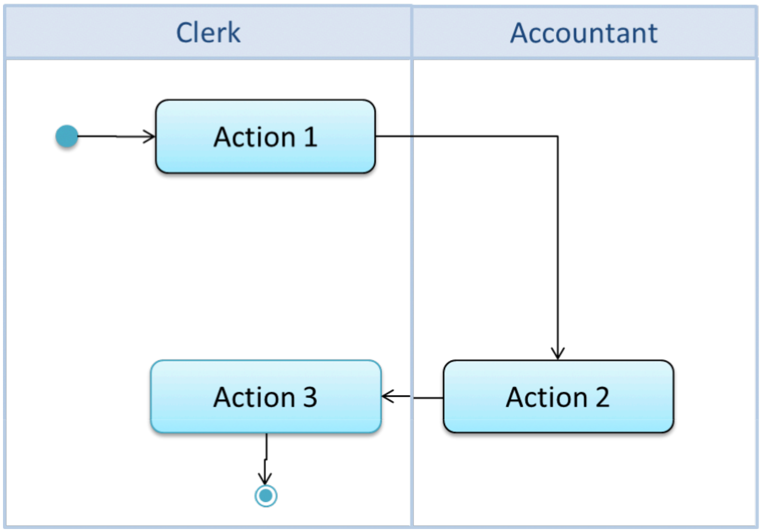
Activity: *snakes and ladders* game


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]

Activity: Move piece



Partitioned activity diagrams, swimlane diagrams



Defensive Programming

Not mean to be modified - return object.copy()
Not null - check if it's null
1-to-1 association - new() in constructor
Referential integrity - check bidirectional associations
Design by contract(DbC) - use assertions to confirm preconditions

Quality Assurance (QA)

Quality Assurance: the process of ensuring that the software being built has the required levels of quality.

Validation and Verification

Quality assurance = validation + verification
Validation - correct requirements - acceptance testing
Verification - correct implementation - system testing

Code Review: the systematic examination code with the intention of finding where the code can be improved.

- Pair programming
- PR reviews
- Formal inspections - a group of people systematically examining a project artifacts to discover defects

the author - the creator of the artifact
the moderator - the planner and executor of the inspection meeting
the secretary - the recorder of the findings of the inspection
the inspector/reviewer - the one who inspects/reviews the artifact.

Advantages of code reviews over testing:
• It 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.
Disadvantages:
• It is a manual process and therefore, error prone.

Static Analysis: the analysis of code without actually executing the code.

- Check style
- Unreachable function
- Unused variable
- Linters

Formal Verification: uses mathematical techniques to prove the correctness of a program.

Formal verification can be used to prove the absence of errors
Testing can be used to prove the presence of errors

Disadvantages
Can only prove the compliance with the specification, but not the actual utility of the software
Requires highly specialized notations and knowledge. Are more commonly used in safety-critical software such as flight control systems.

Developer docs

Documentation for developer-as-user:

API docs
Tutorial-style instructional docs

Documentation for developer-as-maintainer:

- Use plenty of diagrams
- Use plenty of examples
- Use simple and direct explanations
- Get rid of statements that do not add value
- It is not a good idea to have separate sections for each type of artifact

A top-down breadth-first explanation is easier to understand: 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.

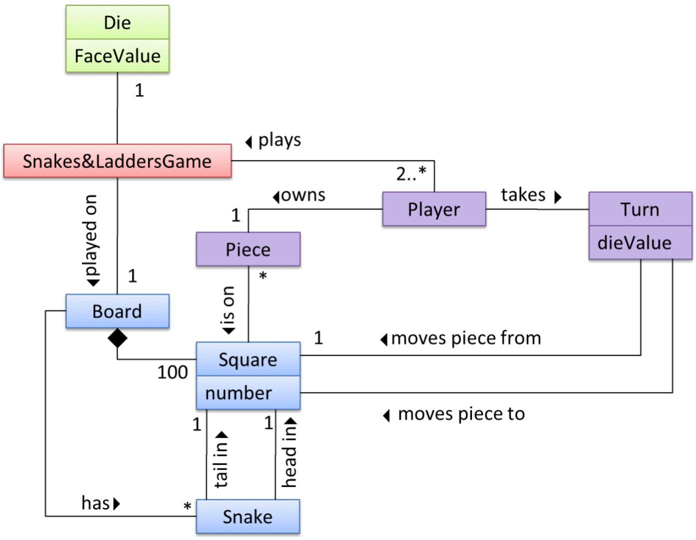
Object Oriented Domain Models / Conceptual Class Diagrams

OODMs: Class diagrams are used to model the problem domain

OODMs do not contain solution-specific classes (i.e. classes that are used in the solution domain but do not exist in the problem domain).

OODMs represents the class structure of the problem domain and not their behavior, just like class diagrams.

OODM notation is similar to class diagram notation but typically omit methods and navigability (arrow head).



Design Patterns

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

Format

- 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.

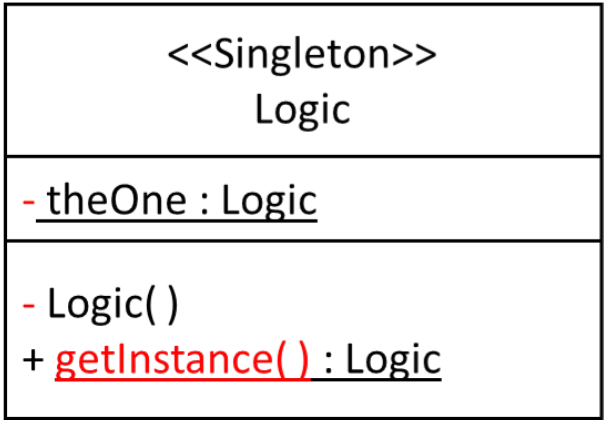
Singleton pattern

Context: A certain classes should have no more than one instance. These single instances are commonly known as singletons.

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

Solution: Constructor - private; Accessor - public

Single instance - private variable



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

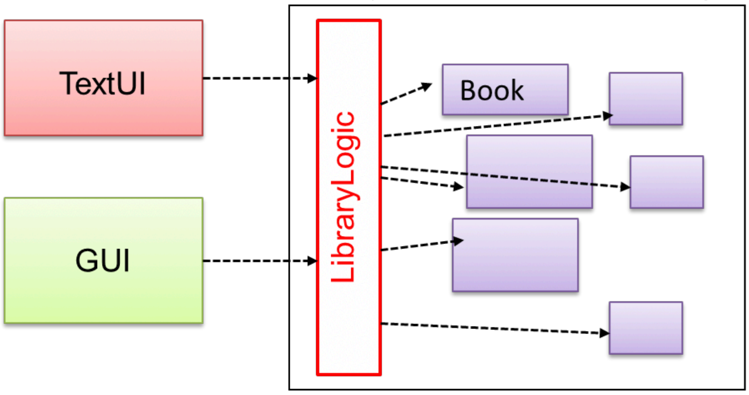
Cons:

- 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.

Facade Pattern

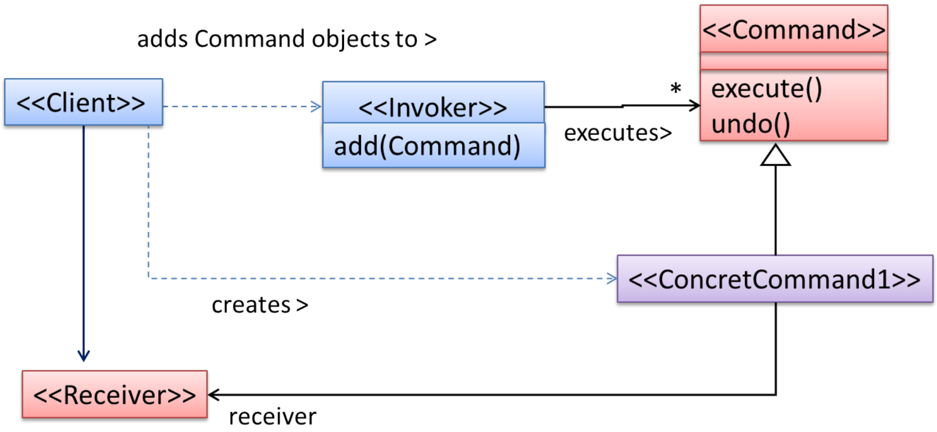
Context: Components need to access functionality deep inside other components

Problem: Access to the component should be allowed without exposing its internal details.
Solution: Include a Façade class - avoid exposing internal details



Command Pattern

Context: A system is required to execute a number of commands, each doing a different task.
Problem: It is preferable that some part of the code executes these commands without having to know each command type.
Solution: A general <Command> object that can be passed around, stored, executed, etc without knowing the type of command.

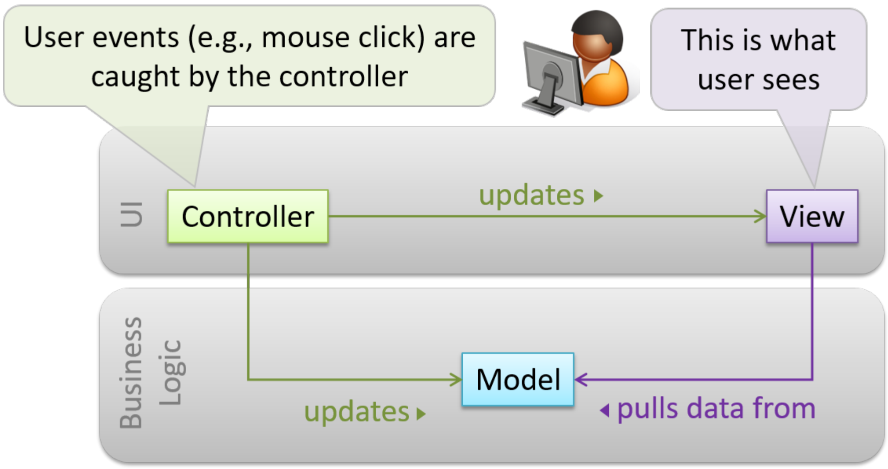


The <<Client>> creates a <<ConcreteCommand>> object, and passes it to the <<Invoker>>. The <<Invoker>> object treats all commands as a general <<Command>> type. <<Invoker>> issues a request by calling execute() on the command. If a command is undoable, <<ConcreteCommand>> will store the state for undoing the command prior to invoking execute(). In addition, the <<ConcreteCommand>> object may have to be linked to any <<Receiver>> of the command (?) before it is passed to the <<Invoker>>. Note that an application of the command pattern does not have to follow the structure given above.

Model View Controller (MVC)

Context: Most applications support storage/retrieval of information, displaying of information to the user, and changing stored information based on external inputs.
Problem: The high coupling that can result from the interlinked nature of the features described above.
Solution:

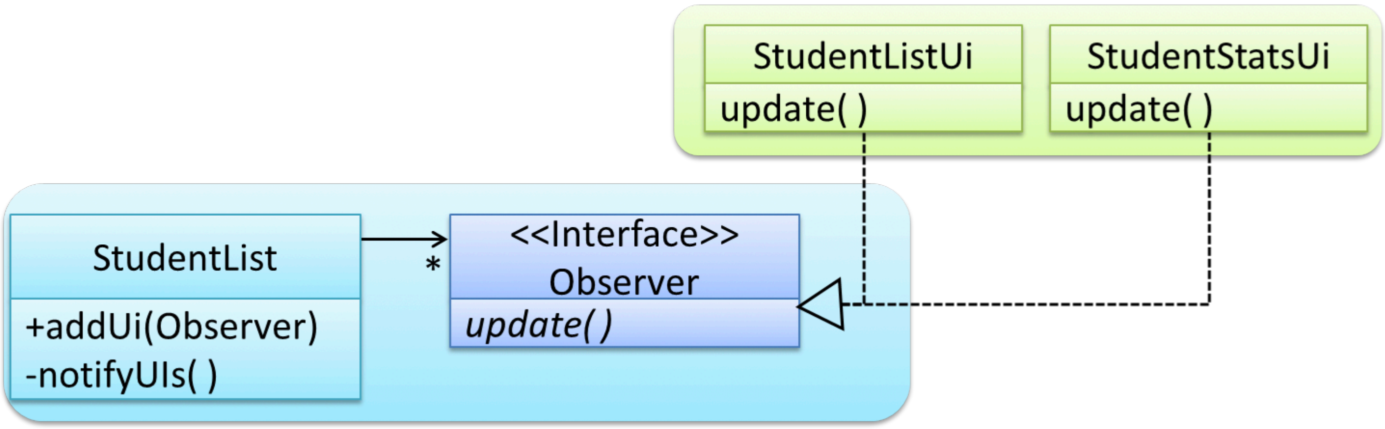
- View: Displays data, interacts with the user, and pulls data from the model if necessary.
- Controller: Detects UI events such as mouse clicks, button pushes and takes follow up action. Updates/changes the model/view when necessary.
- Model: Stores and maintains data. Updates views if necessary.

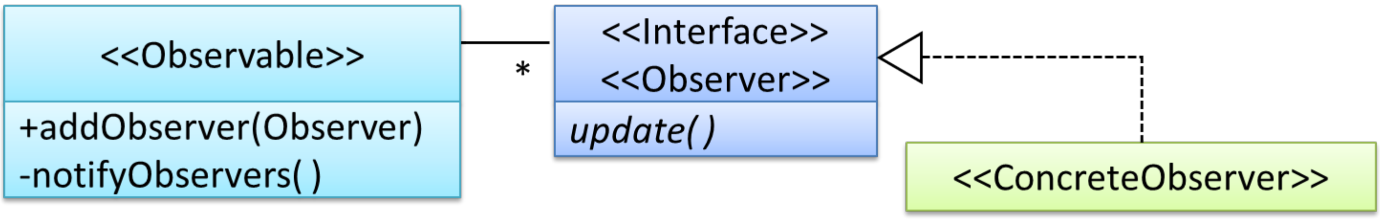


Note that in a simple UI where there's only one view, Controller and View can be combined as one class.

Observer Design 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.

Test Case Design

Effective - finds a high percentage of existing bugs
Efficient - has a high rate of success (bugs found/test cases)

Scripted testing: First write a set of test cases based on the expected behavior of the SUT, and then perform testing based on that set of test cases.

More systematic, likely to discover more bugs given sufficient time

Exploratory testing : Devise test cases on-the-fly, creating new test cases based on the results of the past test cases. - start with error-prone areas

(reactive testing, error guessing technique, attack-based testing, and bug hunting)
Tester's prior experience and intuition

Test case design can be of three types, based on how much of SUT internal details are considered when designing test cases:	
Black-box (aka specification-based or responsibility-based) approach	designed exclusively based on the SUT's specified external behavior
White-box (aka glass-box or structured or implementation-based) approach	designed based on what is known about the SUT's implementation, i.e. the code
Gray-box approach	uses some important information about the implementation

Equivalence partition (equivalence class)

Equivalence partition (aka equivalence class): A group of test inputs that are likely to be processed by the SUT in the same way.
Equivalence partitioning: A test case design technique that uses the below observation to improve the E&E of testing.
Most SUTs do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways. - a range of inputs is treated the same way inside the SUT.

1. Avoid testing too many inputs from one partition - efficiency
2. Ensure all partitions are tested - effectiveness

EPs are usually derived from the specifications of the SUT.
All data participants that can potentially 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, e.g. global variables

Boundary Value Analysis (BVA)

Based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions

One value from the boundary, one value just below the boundary, and one value just above the boundary.

Test Input Combination Strategies

All combinations strategy

Generates test cases for each unique combination of test inputs

At least once strategy

Includes each test input at least once

All pairs strategy

For any given pair of inputs, all combinations between them are tested

Variation: test all pairs of inputs but only for inputs that could influence each other

Random strategy

generates test cases using one of the other strategies and then pick a subset randomly

Each valid input at least once in positive test case

No more than one invalid input in a test case

Software Development Life Cycle (SDLC)

Software Development Life Cycle: software development goes through different stages -> requirements, analysis, design, implementation and testing.

Software Development Life Cycle Model (Software Process Models): different ways to go through SDLC.

Each process model prescribes a "roadmap" for the software developers to manage the development effort.

roadmap: describes the aims of the development stage(s), the artifacts or outcome of each stage as well as the workflow i.e. the relationship between stages.

Sequential process models (waterfall model): Linear process

When one stage of the process is completed, it should produce some artifacts to be used in the next stage. - require each stage to be completed before starting the next

Useful when the problem statement that is well-understood and stable - timely and systematic development effort

The 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

Only have a working product at the end

Iterative process models (iterative and incremental): Having several iterations of SDLC

each of the iterations produces a new version of the product

Have working product at every stage

Breadth-first: iteration evolves all major components in parallel

Depth-first: focuses on fleshing out only some components

Most project use a mixture of breadth-first and depth-first iterations.

Agile process models

Through this work we have come to value:
• Individuals and interactions over processes and tools
• Working software over comprehensive documentation
• Customer collaboration over contract negotiation
• Responding to change over following a plan

- Requirement are prioritized based on the needs of the user, are clarified regularly with the entire project team, and are factored into the development schedule as appropriate.
- Instead of doing a very elaborate and detailed design and a project plan for the whole project, the team works based on a rough project plan and a high level design that evolves as the project goes on.
- Strong emphasis on complete transparency and responsibility sharing among the team members. The team is responsible together for the delivery of the product. Team members are accountable, and regularly and openly share progress with each other and with the user.

Choose the correct statements about agile processes.

a. They value working software over comprehensive documentation.
b. They value responding to change over following a plan.
c. They may not be suitable for some type of projects.
d. XP and Scrum are agile processes.

Light way, flexible to change

SCRUM

Scrum: a process skeleton that contains sets of practices and predefined roles.
Scrum master: maintains the process
Product owner: represents the stakeholders and the business
Team: a cross-functional group, who do the actual analysis, design, implementation, testing
A Scrum project is divided into iterations called Sprints - basic unit of development in Scrum
Each sprint is preceded by a planning meeting
During each sprint, the team creates a potentially deliverable product increment
Scrum enables the creation of self-organizing teams by encouraging co-location of all team members
A key principle of Scrum: requirements churn - during a project the customers can change their minds about what they want and need
Key scrum practive: daily Scrum

EXTREME PROGRAMMING (XP)

XP: stresses customer satisfaction. emphasizes teamwork - all equal partners.
aims to empower developers to confidently respond to changing customer requirements
aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage
Pair programming, CRC cards, project velocity, and standup meetings are some interesting topics related to XP.

Reuse

By reusing tried-and-tested components, the robustness of a new software system can be enhanced while reducing the manpower and time requirement.

- The reused code **may be an overkill** increasing the size of, or/and degrading the performance of, your software.
- The reused software **may not be mature/stable enough** to be used in an important product.
- 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.

Frameworks: A reusable implementation of a software (or part thereof) providing generic functionality that can be selectively customized to produce a specific application

Some frameworks provide a complete implementation of a default behavior which makes them immediately usable.
A framework facilitates the adaptation and customization of some desired functionality.
Some frameworks cover only a specific components or an aspect.
Use inversion of control (Hollywood principle)

- Libraries are meant to be used ‘as is’ while frameworks are meant to be customized/extended.
- Your code calls the library code while the framework code calls your code. Frameworks use a technique called inversion of control, aka the “Hollywood principle”

Platform: Provides a runtime environment for applications

Bundled with various libraries, tools, frameworks, technologies, runtime environment.

Defensive coding
Coding standard
Reduce code duplications
Depth and completeness
Must write some tests - use test-driven development (TDD)
Documentation

Syncing branches

1. Merge master to your branch
2. Rebase your branch onto master
    a. Involves rewrite history, need force-push

Specifying
Text
Feature list / User stories
Prototypes
Use cases = User stories + find more details
Supplementary - mostly non-functional requirement
Glossary

Testing the system
SUT - system under test
Uint testing - test small parts seperately
Test automation - test driver pretend to be the user
Testing frameworks - compare expected and actual - IDE run tests

UML (Unified Modeling Language)

1. Modeling structure
    a. Class structure - name. properties, methods
    b. Object structure? - name, properties
2. Modeling behavior
    a. Features: Use case diagrams

Merge Conflict
Set upstream as a remote
Pull from upstream
Merge new master to your branch
Resolve confilct
Push new commit to branch