Software Engineering for Self-Directed Learners

CS2103/T edition - 2025 Aug-Nov

This is a **printer-friendly** version. It omits exercises, optional topics (i.e., four-star topics), and other extra content such as learning outcomes.

SECTION: SOFTWARE ENGINEERING

Software Engineering

Introduction

Pros and Cons

Software engineering: 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

The following description of the *Joys of the Programming Craft* was taken (and emphasis added) from Chapter 1 of the famous book *The Mythical Man-Month*, by Frederick P. Brooks.

> **Why is programming fun? What delights may its practitioner expect as his reward?**
>
> **First is the sheer joy of making things.** As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake.
>
> **Second is the pleasure of making things that are useful to other people.** Deep within, you want others to use your work and to find it helpful. In this respect the programming system is not essentially different from the child's first clay pencil holder "for Daddy's office."
>
> **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. The programmed computer has all the fascination of the pinball machine or the jukebox mechanism, carried to the ultimate.
>
> **Fourth is the joy of always learning**, which springs from the nonrepeating nature of the task. In one way or another the problem is ever new, and its solver learns something: sometimes practical, sometimes theoretical, and sometimes both.
>
> **Finally, there is the delight of working in such a tractable medium.** The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by the exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures....
>
> Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.
>
> Programming then is fun because it gratifies creative longings built deep within us and delights sensibilities you have in common with all men.

> **Not all is delight, however, and knowing the inherent woes makes it easier to bear them when they appear.**
>
> **First, one must perform perfectly.** The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.
>
> **Next, other people set one's objectives, provide one's resources, and furnish one's information.** One rarely controls the circumstances of his work, or even its goal. In management terms, one's authority is not sufficient for his responsibility. It seems that in all fields, however, the jobs where things get done never have formal authority commensurate with responsibility. In practice, actual (as opposed to formal) authority is acquired from the very momentum of accomplishment.

> The dependence upon others has a particular case that is especially painful for the system programmer. He depends upon other people's programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable.
>
> **The next woe is that designing grand concepts is fun; finding nitty little bugs is just work.** With any creative activity come dreary hours of tedious, painstaking labor, and programming is no exception.
>
> **Next, one finds that debugging has a linear convergence, or worse**, where one somehow expects a quadratic sort of approach to the end. So testing drags on and on, the last difficult bugs taking more time to find than the first.
>
> **The last woe, and sometimes the last straw, is that the product over which one has labored so long appears to be obsolete upon (or before) completion.** Already colleagues and competitors are in hot pursuit of new and better ideas. Already the displacement of one's thought-child is not only conceived, but scheduled.
>
> This always seems worse than it really is. The new and better product is generally not available when one completes his own; it is only talked about. It, too, will require months of development. The real tiger is never a match for the paper one, unless actual use is wanted. Then the virtues of reality have a satisfaction all their own.
>
> Of course the technological base on which one builds is always advancing. As soon as one freezes a design, it becomes obsolete in terms of its concepts. But implementation of real products demands phasing and quantizing. The obsolescence of an implementation must be measured against other existing implementations, not against unrealized concepts. The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.
>
> This then is programming, both a tar pit in which many efforts have floundered and a creative activity with joys and woes all its own. **For many, the joys far outweigh the woes....**

SECTION: PROGRAMMING PARADIGMS

Object-Oriented Programming

Introduction

What

Object-Oriented Programming (OOP) is a *programming paradigm*. A programming paradigm guides programmers to analyze programming problems, and structure programming solutions, in a specific way.

> Programming languages have traditionally divided the world into two parts—data and operations on data. Data is static and immutable, except as the operations may change it. The procedures and functions that operate on data have no lasting state of their own; they're useful only in their ability to affect data.
>
> This division is, of course, grounded in the way computers work, so it's not one that you can easily ignore or push aside. Like the equally pervasive distinctions between matter and energy and between nouns and verbs, it forms the background against which you work. At some point, all programmers—even object-oriented programmers—must lay out the data structures that their programs will use and define the functions that will act on the data.
>
> With a procedural programming language like C, that's about all there is to it. The language may offer various kinds of support for organizing data and functions, but it won't divide the world any differently. Functions and data structures are the basic elements of design.
>
> Object-oriented programming doesn't so much dispute this view of the world as restructure it at a higher level. It groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.
>
> -- Object-Oriented Programming with Objective-C, Apple

Some other examples of programming paradigms are:

Paradigm	Programming Languages
Procedural Programming paradigm	C
Functional Programming paradigm	F#, Haskell, Scala
Logic Programming paradigm	Prolog

Some programming languages support multiple paradigms.

> Java is primarily an OOP language but it supports limited forms of functional programming and it can be used to (although not recommended to) write procedural code. e.g., se-edu/addressbook-level1
>
> JavaScript and Python support functional, procedural, and OOP programming.

Objects

What

An *object* in Object-Oriented Programming (OOP) has *state* and *behavior*, similar to objects in the real world.

> Every object has both state (data) and behavior (operations on data). In that, they're not much different from ordinary physical objects. It's easy to see how a mechanical device, such as a pocket watch or a piano, embodies both state and behavior. But almost anything that's designed to do a job does, too. Even simple things with no moving parts such as an ordinary bottle combine state (how full the bottle is, whether or not it's open, how warm its contents are) with behavior (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand high or low temperatures).

> It's this resemblance to real things that gives objects much of their power and appeal. They can not only model components of real systems, but equally as well fulfill assigned roles as components in software systems.
>
> -- Object-Oriented Programming with Objective-C, Apple

OOP views the world as *a network of interacting objects*.

📦 A real world scenario viewed as a network of interacting objects:

You are asked to find out the average age of a group of people Adam, Beth, Charlie, and Daisy. You take a piece of paper and pen, go to each person, ask for their age, and note it down. After collecting the age of all four, you enter it into a calculator to find the total. And then, use the same calculator to divide the total by four, to get the average age. This can be viewed as the objects `You`, `Pen`, `Paper`, `Calculator`, `Adam`, `Beth`, `Charlie`, and `Daisy` interacting to accomplish the end result of calculating the average age of the four persons. These objects can be considered as *connected* in a certain *network* of certain structure that dictates how these objects can interact. For example, `You` object is connected to the `Pen` object, and hence `You` can use the `Pen` object to write.

OOP solutions try to create a similar object network inside the computer's memory – a sort of virtual simulation of the corresponding real world scenario – **so that a similar result can be achieved programmatically.**

OOP does not demand that the virtual world object network follow the real world *exactly*.

📦 Our previous example can be tweaked a bit as follows:

- Use an object called `Main` to represent your role in the scenario.
- As there is no physical writing involved, you can replace the `Pen` and `Paper` with an object called `AgeList` that is able to keep a list of ages.

Every object has both state (data) and behavior (operations on data).

📦 The state and behavior of our running example are as follows:

Object	Real World?	Virtual World?	Example of State (i.e., Data)	Examples of Behavior (i.e., Operations)
Adam	✓	✓	Name, Date of Birth	Calculate age based on birthday
Pen	✓	-	Ink color, Amount of ink remaining	Write
AgeList	-	✓	Recorded ages	Give the number of entries, Accept an entry to record
Calculator	✓	✓	Numbers already entered	Calculate the sum, divide
You/Main	✓	✓	Average age, Sum of ages	Use other objects to calculate

Every object has an *interface* and an *implementation*.

Every real world object has,

- an interface through which other objects can interact with it, and,
- an implementation that supports the interface but may not be accessible to the other object.

📦 The interface and implementation of some real-world objects in our example:

- Calculator: the buttons and the display are part of the interface; circuits are part of the implementation.
- Adam: In the context of our 'calculate average age' example,

- the interface of Adam consists of requests that Adam will respond to, e.g., "Give age to the nearest year, as at Jan 1st of this year" "State your name".
- the implementation includes the mental calculation Adam uses to calculate the age which is not visible to other objects.

Similarly, every object in the virtual world has an interface and an implementation.

📦 The interface and implementation of some virtual-world objects in our example:

- `Adam` : the interface might have a method `getAge(Date asAt)` ; the implementation of that method is not visible to other objects.

Objects interact by sending messages. Both real world and virtual world object interactions can be viewed as objects sending messages to each other. The message can result in the sender object receiving a response and/or the receiver object's state being changed. Furthermore, the result can vary based on which object received the message, even if the message is identical (see rows 1 and 2 in the example below).

📦 Same messages and responses from our running example:

World	Sender	Receiver	Message	Response	State Change
Real	You	Adam	"What is your name?"	"Adam"	-
Real	as above	Beth	as above	"Beth"	-
Real	You	Pen	Put nib on paper and apply pressure	Makes a mark on your paper	Ink level goes down
Virtual	Main	Calculator (current total is 50)	add(int i): int i = 23	73	total = total + 23

Objects as Abstractions

The concept of ***Objects*** **in OOP is an abstraction mechanism because it allows us to abstract away the lower level details and work with bigger granularity entities** i.e., ignore details of data formats and the method implementation details and work at the level of objects.

📦 You can deal with a `Person` object that represents the person Adam and query the object for Adam's age instead of dealing with details such as Adam's date of birth (DoB), in what format the DoB is stored, the algorithm used to calculate the age from the DoB, etc.

Encapsulation Of Objects

> Encapsulation protects an implementation from unintended actions and from inadvertent access.
>
> -- Object-Oriented Programming with Objective-C, Apple

An object is an *encapsulation* of some data and related behavior in terms of two aspects:

1. The *packaging* aspect: An object packages data and related behavior together into one self-contained unit.

2. The *information hiding* aspect: The data in an object is hidden from the outside world and are only accessible using the object's interface.

Classes

What

Writing an OOP program is essentially writing instructions that the computer will use to,

1. **create the virtual world of the object network, and**
2. **provide it the inputs to produce the outcome you want.**

A *class* contains instructions for creating a specific kind of objects. It turns out sometimes multiple objects keep the same type of data and have the same behavior because they are of the *same kind*. Instructions for creating a 'kind' (or 'class') of objects can be done once and those same instructions can be used to *instantiate* objects of that kind. We call such instructions a *Class*.

Classes and objects in an example scenario

Consider the example of writing an OOP program to calculate the average age of Adam, Beth, Charlie, and Daisy.

Instructions for creating objects `Adam`, `Beth`, `Charlie`, and `Daisy` will be very similar because they are all of the same kind: they all represent 'persons' with the same interface, the same kind of data (i.e., `name`, `dateOfBirth`, etc.), and the same kind of behavior (i.e., `getAge(Date)`, `getName()`, etc.). Therefore, you can have a class called `Person` containing instructions on how to create `Person` objects and use that class to instantiate objects `Adam`, `Beth`, `Charlie`, and `Daisy`.

Similarly, you need classes `AgeList`, `Calculator`, and `Main` classes to instantiate one each of `AgeList`, `Calculator`, and `Main` objects.

Class	Objects
`Person`	objects representing Adam, Beth, Charlie, Daisy
`AgeList`	an object to represent the age list
`Calculator`	an object to do the calculations
`Main`	an object to represent you (i.e., the one who manages the whole operation)

Class Level Members

While all objects of a class have the same attributes, each object has its own copy of the attribute value.

All `Person` objects have the `name` attribute but the value of that attribute varies between `Person` objects.

However, some attributes are not suitable to be maintained by individual objects. Instead, they should be maintained centrally, shared by all objects of the class. They are like 'global variables' but attached to a specific class. Such **variables whose value is shared by all instances of a class are called *class-level attributes***.

The attribute `totalPersons` should be maintained centrally and shared by all `Person` objects rather than copied at each `Person` object.

Similarly, when a normal method is being called, a message is being sent to the receiving object and the result may depend on the receiving object.

Sending the `getName()` message to the `Adam` object results in the response `"Adam"` while sending the same message to the `Beth` object results in the response `"Beth"`.

However, there can be methods related to a specific class but not suitable for sending messages to a specific object of that class. Such **methods that are called using the class instead of a specific instance are called *class-level methods***.

The method `getTotalPersons()` is not suitable to send to a specific `Person` object because a specific object of the `Person` class should not have to know about the total number of `Person` objects.

Class-level attributes and methods are collectively called *class-level members* (also called *static members* sometimes because some programming languages use the keyword `static` to identify class-level members). **They are to be accessed using the class name rather than an instance of the class**.

Enumerations

An *Enumeration* is a fixed set of values that can be considered as a data type. An enumeration is often useful when using a regular data type such as `int` or `String` would allow invalid values to be assigned to a variable.

> Suppose you want a variable called `priority` to store the priority of something. There are only three priority levels: high, medium, and low. You can declare the variable `priority` as of type `int` and use only values `2`, `1`, and `0` to indicate the three priority levels. However, this opens the possibility of an invalid value such as `9` being assigned to it. But if you define an enumeration type called `Priority` that has three values `HIGH`, `MEDIUM` and `LOW` only, a variable of type `Priority` will never be assigned an invalid value because the compiler is able to catch such an error.
>
> `Priority`: `HIGH`, `MEDIUM`, `LOW`

Associations

What

Objects in an OO solution need to be connected to each other to form a network so that they can interact with each other. Such **connections between objects are called *associations*.**

> Suppose an OOP program for managing a learning management system creates an object structure to represent the related objects. In that object structure you can expect to have associations between a `Course` object that represents a specific course and `Student` objects that represent students taking that course.

Associations in an object structure can change over time.

> To continue the previous example, the associations between a `Course` object and `Student` objects can change as students enroll in the course or drop the course over time.

Associations among objects can be generalized as associations between the corresponding classes too.

> In our example, as some `Course` objects can have associations with some `Student` objects, you can view it as an association between the `Course` class and the `Student` class.

Implementing associations

You use instance level variables to implement associations.

> In our example, the `Course` class can have a `students` variable to keeps track of students associated with a particular course.

Navigability

When two classes are linked by an association, it does not necessarily mean both objects taking part in an instance of the association *knows about* (i.e., has a reference to) each other. **The concept of *navigability* tells us if an object taking part in association knows about the other.** In other words, it tells us if we can 'navigate' from the one object to the other in a given direction -- because if the object 'knows' about the other, it has a reference to the other object, and we can use that reference to 'navigate to' (i.e., access) that other object.

Navigability can be *unidirectional* or *bidirectional*. Suppose there is an association between the classes `Box` and `Rope`, and the `Box` object `b` and the `Rope` object `r` is taking part in one instance of that association.

- **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`. That is, one can *navigate* from `b` to `r` using the `b`'s object reference of `r` (but not in the other direction).
  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 for the same single instance of the association.

Note that two unidirectional associations in opposite directions do not add up to a single bidirectional association.

In the code below, there is a bidirectional association between the `Person` class and the `Cat` class i.e., if `Person p` is the owner of the `Cat c`, `p` it will result in `p` and `c` having references to each other.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person owner;
    //...
}
```

The code below has two unidirectional associations between the `Person` class and the `Cat` class (in opposite directions). Because the breeder is not necessarily the same person keeping the cat as a pet, they are two separate associations, not a bidirectional association.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person breeder;
    //...
}
```

Multiplicity ⚡⚡

***Multiplicity* is the aspect of an OOP solution that dictates how many objects take part in each association.**

The multiplicity of the association between `Course` objects and `Student` objects tells you how many `Course` objects can be associated with one `Student` object and vice versa.

Implementing multiplicity

A normal instance-level variable gives us a `0..1` multiplicity (also called *optional associations*) because a variable can hold a reference to a single object or `null`.

In the code below, the `Logic` class has a variable that can hold `0..1` i.e., zero or one `Minefield` objects.

```
class Logic {
    Minefield minefield;
    // ...
}

class Minefield {
    //...
}
```

A variable can be used to implement a `1` multiplicity too (also called *compulsory associations*).

In the code below, the `Logic` class will always have a `ConfigGenerator` object, provided the variable is not set to `null` at some point.

```
class Logic {
    ConfigGenerator cg = new ConfigGenerator();
    ...
}
```

To implement other multiplicities, choose a suitable data structure such as Arrays, ArrayLists, HashMaps, Sets, etc.

This code uses a two-dimensional array to implement a 1-to-many association from the `Minefield` to `Cell`.

```
class Minefield {
    Cell[][] cell;
    //...
}
```

Dependencies ⚡⚡⚡

In the context of OOP associations, **a *dependency* is a need for one class to depend on another without having a direct association in the same direction.** Reason for the exclusion: If there is an association from class `Foo` to class `Bar` (i.e., navigable from `Foo` to `Bar`), that means `Foo` is *obviously* dependent on `Bar` and hence there is no point in mentioning *dependency* specifically. In other words, we are specifically focusing on *non-obvious* dependencies here. One cause of such dependencies is interactions between objects that do not have a long-term link between them.

A `Course` class can have a dependency on a `Registrar` class because the `Course` class needs to refer to the `Registrar` class to obtain the maximum number of students it can support (e.g., `Registrar.MAX_COURSE_CAPACITY`).

In the code below, `Foo` has a dependency on `Bar` but it is not an association because it is only a transient interaction and there is no long term relationship between a `Foo` object and a `Bar` object. i.e., the `Foo` object does not keep the `Bar` object it receives as a parameter.

```
class Foo {

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

class Bar {
    int value;

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

Composition ⚡⚡⚡

A composition is an association that represents a strong *whole-part* relationship.

A `Board` (used for playing board games) consists of `Square` objects.

Composition implies,

1. **when the *whole* is destroyed, *parts* are destroyed too** i.e., the *part* cannot exist without being attached to a *whole*.
2. **there cannot be cyclical links**.

The 'sub-folder' association between `Folder` objects is a composition type association. Consider the case of `Folder` object `subF` is a sub-folder of `Folder` object `F`. In this case,

1. if `F` is deleted, `subF` will be deleted with it.
2. `F` cannot be a sub-folder of `subF` (i.e., no cyclical 'sub-folder' association between the two objects).

Whether a relationship is a composition can depend on the context.

Is the relationship between `Email` and `EmailSubject` composition? That is, is the email subject *part* of an email to the extent that an email subject cannot exist without an email?

- When modeling an application that sends emails, the answer is 'yes'.
- When modeling an application that gather analytics about email traffic, the answer may be 'no' (e.g., the application might collect just the email subjects for text analysis).

A common use of composition is when parts of a big class are carved out as smaller classes for the ease of managing the internal design. In such cases, the classes extracted out still act as *parts* of the bigger class and the outside world has no business knowing about them.

***Cascading deletion* alone is not sufficient for composition.** Suppose there is a design in which `Person` objects are attached to `Task` objects and the former get deleted whenever the latter is deleted. This fact alone does not mean there is a composition relationship between the two classes. For it to be composition, a `Person` must be an integral *part* of a `Task` in the context of that association, at the concept level (not simply at implementation level).

Identifying and keeping track of composition relationships in the design has benefits such as helping to maintain the data integrity of the system. For example, when you know that a certain relationship is a composition, you can take extra care in your implementation to ensure that when the *whole* object is deleted, all its *parts* are deleted too.

Implementing composition

Composition is implemented using a normal variable. If correctly implemented, the 'part' object will be deleted when the 'whole' object is deleted. Ideally, the 'part' object may not even be visible to clients of the 'whole' object.

Here is one way to implement the composition between `Email` and `Subject`:

```
class Email {
    private Subject subject;
  ...
}
```

In this code, the `Email` has a composition type relationship with the `Subject` class, in the sense that the subject is part of the email.

Aggregation ⚡⚡⚡

***Aggregation* represents a *container-contained* relationship.** It is a weaker relationship than composition.

`SportsClub` can act as a *container* for `Person` objects who are members of the club. `Person` objects can survive without a `SportsClub` object.

Implementing aggregation

Implementation is similar to that of composition except the *containee* object can exist even after the *container* object is deleted.

In the code below, there is an aggregation association between the `Team` class and the `Person` class in that a `Team` contains a `Person` object who is the leader of the team.

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

Association Classes ϟϟϟ

An *association class* represents additional information about an association. It is a normal class but plays a special role from a design point of view.

📦 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. In such situations, an additional association class can be introduced, e.g., a `Marriage` class, to store such information.

Implementing association classes

There is no special way to implement an association class. It can be implemented as a normal class that has variables to represent the endpoint of the association it represents.

📦 In the code below, the `Transaction` class is an association class that represents a transaction between a `Person` who is the seller and another `Person` who is the buyer.

```
class Transaction {

    //all fields are compulsory
    Person seller;
    Person buyer;
    Date date;
    String receiptNumber;

    Transaction(Person seller, Person buyer, Date date, String receiptNumber) {
        //set fields
    }
}
```

Inheritance

What 🎓: ϟ

The OOP concept *Inheritance* allows you to define a new class based on an existing class.

📦 For example, you can use inheritance to define an `EvaluationReport` class based on an existing `Report` class so that the `EvaluationReport` class does not have to duplicate data/behaviors that are already implemented in the `Report` class. The `EvaluationReport` can inherit the `wordCount` attribute and the `print()` method from the *base class* `Report`.

- Other names for Base class: *Parent* class, *Superclass*
- Other names for Derived class: *Child* class, *Subclass*, *Extended* class

A superclass is said to be *more general* than the subclass. Conversely, a subclass is said to be more *specialized* than the superclass.

Applying inheritance on a group of similar classes can result in the common parts among classes being extracted into more general classes.

📦 `Man` and `Woman` behave the same way for certain things. However, the two classes cannot be simply replaced with a more general class `Person` because of the need to distinguish between `Man` and `Woman` for certain other things. A solution is to add the `Person` class as a superclass (to contain the code common to men and women) and let `Man` and `Woman` inherit from `Person` class.

Inheritance implies the derived class can be considered as a *subtype* of the base class (and the base class is a *super-type* of the derived class), resulting in an *is a* relationship.

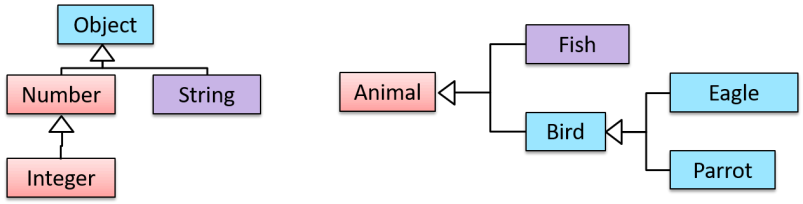
Inheritance does not necessarily mean a subtype relationship exists. However, the two often go hand-in-hand. For simplicity, at this point let us assume inheritance implies a subtype relationship.

To continue the previous example,

- `Woman` *is a* `Person`
- `Man` *is a* `Person`

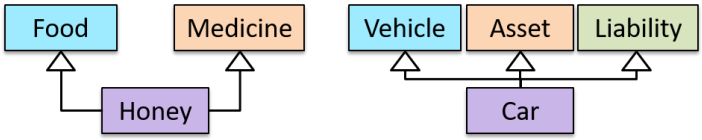
Inheritance relationships through a chain of classes can result in inheritance *hierarchies* (aka inheritance *trees*).

Two inheritance hierarchies/trees are given below. Note that the triangle points to the parent class. Observe how the `Parrot` *is a* `Bird` as well as it *is an* `Animal`.



***Multiple Inheritance* is when a class inherits *directly* from multiple classes.** Multiple inheritance among classes is allowed in some languages (e.g., Python, C++) but not in other languages (e.g., Java, C#).

The `Honey` class inherits from the `Food` class *and* the `Medicine` class because honey can be consumed as a food as well as a medicine (in some oriental medicine practices). Similarly, a `Car` is a `Vehicle`, an `Asset` and a `Liability`.



Overriding

Method *overriding* is when a subclass changes the behavior inherited from the parent class by re-implementing the method. Overridden methods have the same name, the same type signature, and the same (or a subtype of the) return type.

Consider the following case of `EvaluationReport` class inheriting the `Report` class:

`Report` methods	`EvaluationReport` methods	Overrides?
`print()`	`print()`	Yes
`write(String)`	`write(String)`	Yes
`read():String`	`read(int):String`	No. Reason: the two methods have different signatures; This is a case of *overloading* (rather than overriding).

Overloading

Method overloading is when there are multiple methods with the same name but different type signatures. Overloading is used to indicate that multiple operations do similar things but take different parameters.

Type signature: The *type signature* of an operation is the type sequence of the parameters. The return type and parameter names are not part of the type signature. However, the parameter order is significant.

Example:

Method	Type Signature
`int add(int X, int Y)`	`(int, int)`
`void add(int A, int B)`	`(int, int)`
`void m(int X, double Y)`	`(int, double)`
`void m(double X, int Y)`	`(double, int)`

In the case below, the `calculate` method is overloaded because the two methods have the same name but different type signatures `(String)` and `(int)`.

- `calculate(String): void`
- `calculate(int): void`

Interfaces

An *interface* is a behavior specification i.e., a collection of method specifications. If a class implements the interface, it means the class is able to support the behaviors specified by the said interface.

> There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts. --Oracle Docs on Java

Suppose `SalariedStaff` is an interface that contains two methods `setSalary(int)` and `getSalary()`. `AcademicStaff` can declare itself as *implementing* the `SalariedStaff` interface, which means the `AcademicStaff` class must implement all the methods specified by the `SalariedStaff` interface i.e., `setSalary(int)` and `getSalary()`.

A class implementing an interface results in an *is-a* relationship, just like in class inheritance.

In the example above, `AcademicStaff` *is a* `SalariedStaff`. An `AcademicStaff` object can be used anywhere a `SalariedStaff` object is expected e.g., `SalariedStaff ss = new AcademicStaff()`.

Abstract Classes

Abstract class: A class declared as an *abstract class* cannot be instantiated, but it can be subclassed.

You can declare a class as *abstract* when a class is merely a representation of commonalities among its subclasses in which case it does not make sense to instantiate objects of that class.

The `Animal` class that exists as a generalization of its subclasses `Cat`, `Dog`, `Horse`, `Tiger` etc. can be declared as abstract because it does not make sense to instantiate an `Animal` object.

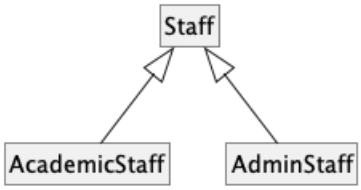
Abstract method: An *abstract method* is a method signature without a method implementation.

The `move` method of the `Animal` class is likely to be an abstract method as it is not possible to implement a `move` method at the `Animal` class level to fit all subclasses because each animal type can move in a different way.

A class that has an abstract method becomes an abstract class because the class definition is incomplete (due to the missing method body) and it is not possible to create objects using an incomplete class definition.

Substitutability

Every instance of a subclass is an instance of the superclass, but not vice-versa. As a result, inheritance allows *substitutability*: the ability to substitute a child class object where a parent class object is expected.



An `AcademicStaff` is an instance of a `Staff`, but a `Staff` is not necessarily an instance of an `AcademicStaff`. i.e., wherever an object of the superclass is expected, it can be substituted by an object of any of its subclasses.

The following code is valid because an `AcademicStaff` object is substitutable as a `Staff` object.

```
Staff staff = new AcademicStaff(); // OK
```

But the following code is not valid because `staff` is declared as a `Staff` type and therefore its value may or may not be of type `AcademicStaff`, which is the type expected by variable `academicStaff`.

```
Staff staff;
...
AcademicStaff academicStaff = staff; // Not OK
```

Dynamic and Static Binding

Dynamic binding (aka late 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.

Consider the code below. The declared type of `s` is `Staff` and it appears as if the `adjustSalary(int)` operation of the `Staff` class is invoked.

```
void adjustSalary(int byPercent) {
    for (Staff s: staff) {
        s.adjustSalary(byPercent);
    }
}
```

However, at runtime `s` can receive an object of any subclass of `Staff`. That means the `adjustSalary(int)` operation of the actual subclass object will be called. If the subclass does not override that operation, the operation defined in the superclass (in this case, `Staff` class) will be called.

Static binding (aka early binding): When a method call is resolved at compile time.

In contrast, overloaded methods are resolved using static binding.

Note how the constructor is overloaded in the class below. The method call `new Account()` is bound to the first constructor at compile time.

```
class Account {

    Account() {
        // Signature: ()
        ...
    }

    Account(String name, String number, double balance) {
        // Signature: (String, String, double)
        ...
    }
}
```

Similarly, the `calculateGrade` method is overloaded in the code below and a method call `calculateGrade("A1213232")` is bound to the second implementation, at compile time.

```
void calculateGrade(int[] averages) { ... }
void calculateGrade(String matric) { ... }
```

Polymorphism

What

Polymorphism:

> The ability of different objects to respond, each in its own way, to identical messages is called polymorphism. -- Object-Oriented Programming with Objective-C, Apple

***Polymorphism* allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object.**

Assume classes `Cat` and `Dog` are both subclasses of the `Animal` class. You can write code targeting `Animal` objects and use that code on `Cat` and `Dog` objects, achieving possibly different results based on whether it is a `Cat` object or a `Dog` object. Some examples:

- Declare an array of type `Animal` and still be able to store `Dog` and `Cat` objects in it.
- Define a method that takes an `Animal` object as a parameter and yet be able to pass `Dog` and `Cat` objects to it.
- Call a method on a `Dog` or a `Cat` object as if it is an `Animal` object (i.e., without knowing whether it is a `Dog` object or a `Cat` object) and get a different response from it based on its actual class e.g., call the `Animal` class's method `speak()` on object `a` and get a `"Meow"` as the return value if `a` is a `Cat` object and `"Woof"` if it is a `Dog` object.

Polymorphism literally means "ability to take many forms".

How

Three concepts combine to achieve polymorphism: *substitutability, operation overriding, and dynamic binding*.

- **Substitutability:** Because of substitutability, you can write code that expects objects of a parent class and yet use that code with objects of child classes. That is how polymorphism is able to *treat objects of different types as one type*.
- **Overriding:** To get polymorphic behavior from an operation, the operation in the superclass needs to be overridden in each of the subclasses. That is how overriding allows objects of different subclasses to *display different behaviors in response to the same method call*.
- **Dynamic binding**: Calls to overridden methods are bound to the implementation of the actual object's class dynamically during the runtime. That is how the polymorphic code can call the method of the parent class and yet execute the implementation of the child class.

More

Miscellaneous ⚡⚡⚡

What is the difference between a Class, an Abstract Class, and an Interface?

- An interface is a behavior specification with no implementation.
- A class is a behavior specification + implementation.
- An abstract class is a behavior specification + a possibly incomplete implementation.

How does *overriding* differ from *overloading*?

Overloading is used to indicate that multiple operations do similar things but take different parameters. Overloaded methods have the same method name but different method signatures and possibly different return types.

Overriding is when a sub-class redefines an operation using the same method name and the same type signature. Overridden methods have the same name, same method signature, and same return type.

SECTION: REQUIREMENTS

Requirements

Introduction

A *software requirement* specifies a need to be fulfilled by the software product.

A software project may be,

- **a *brownfield* project** i.e., develop a product to replace/update an existing software product
- **a *greenfield* project** i.e., develop a totally new system from scratch

In either case, requirements need to be gathered, analyzed, specified, and managed.

Requirements come from *stakeholders*.

Stakeholder: An individual or an organization that is involved or potentially affected by the software project. e.g., users, sponsors, developers, interest groups, government agencies, etc.

Identifying requirements is often not easy. For example, stakeholders may not be aware of their precise needs, may not know how to communicate their requirements correctly, may not be willing to spend effort in identifying requirements, etc.

Non-Functional Requirements

Requirements can be divided into two in the following way:

1. ***Functional requirements* specify what the system should do.**
2. ***Non-functional requirements* specify the constraints under which the system is developed and operated.**

Some examples of non-functional requirement categories:

- Data requirements e.g. size, volatility, persistency etc.,
- Environment requirements e.g. technical environment in which the system would operate in or needs to be compatible with.
- Accessibility, Capacity, Compliance with regulations, Documentation, Disaster recovery, Efficiency, Extensibility, Fault tolerance, Interoperability, Maintainability, Privacy, Portability, Quality, Reliability, Response time, Robustness, Scalability, Security, Stability, Testability, and more ...

Some concrete examples of NFRs

- Business/domain rules: e.g. the size of the minefield cannot be smaller than five.
- Constraints: e.g. the system should be backward compatible with data produced by earlier versions of the system; system testers are available only during the last month of the project; the total project cost should not exceed $1.5 million.
- Technical requirements: e.g. the system should work on both 32-bit and 64-bit environments.
- Performance requirements: e.g. the system should respond within two seconds.
- Quality requirements: e.g. the system should be usable by a novice who has never carried out an online purchase.
- Process requirements: e.g. the project is expected to adhere to a schedule that delivers a feature set every one month.
- Notes about project scope: e.g. the product is not required to handle the printing of reports.
- Any other noteworthy points: e.g. the game should not use images deemed offensive to those injured in real mine clearing activities.

You may have to spend an extra effort in digging NFRs out as early as possible because,

1. **NFRs are easier to miss** e.g., stakeholders tend to think of functional requirements first
2. sometimes **NFRs are critical to the success of the software.** E.g. A web application that is too slow or that has low security is unlikely to succeed even if it has all the right functionality.

Quality of Requirements ⚡⚡⚡

Here are some characteristics of well-defined requirements [📖 zielczynski]:

- Unambiguous
- Testable (verifiable)
- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic
- Necessary
- Implementation-free (i.e., abstract)

Besides these criteria for individual requirements, the set of requirements as a whole should be

- Consistent
- Non-redundant
- Complete

Prioritizing Requirements ⚡⚡⚡

Requirements can be prioritized based on the importance and urgency, while keeping in mind the constraints of schedule, budget, staff resources, quality goals, and other constraints.

A common approach is to group requirements into priority categories. Note that all such scales are subjective, and stakeholders define the meaning of each level in the scale for the project at hand.

📦 An example scheme for categorizing requirements:

- `Essential` : The product must have this requirement fulfilled or else it does not get user acceptance.
- `Typical` : Most similar systems have this feature although the product can survive without it.
- `Novel` : New features that could differentiate this product from the rest.

📦 Other schemes:

- `High` , `Medium` , `Low`
- `Must-have` , `Nice-to-have` , `Unlikely-to-have`
- `Level 0` , `Level 1` , `Level 2` , ...

Some requirements can be discarded if they are considered 'out of *scope*'.

📦 The requirement given below is for a Calendar application. Stakeholders of the software (e.g. product designers) might decide the following requirement is not in the scope of the software.

> The software records the actual time taken by each task and show the difference between the *actual* and *scheduled* time for the task.

Gathering requirements

Brainstorming ⚡⚡⚡

Brainstorming: A group activity designed to *generate a large number of diverse and creative ideas* for the *solution* of a problem.

In a brainstorming session there are no "bad" ideas. The aim is to *generate ideas; not to validate* them. Brainstorming encourages you to "think outside the box" and put "crazy" ideas on the table without fear of rejection.

User Surveys ⚡⚡⚡

Surveys can be used to *solicit responses and opinions* from a large number of stakeholders regarding a current product or 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 about how an existing system is being used, which can help in building a better replacement e.g. to find the situations where the user makes mistakes when using the current system.

Interviews ⚡⚡⚡

Interviewing stakeholders and domain experts can produce useful information about project requirements.

Focus Groups ⚡⚡⚡

Focus groups are a kind of informal interview within an interactive group setting. A group of people (e.g. potential users, beta testers) are asked about their understanding of a specific issue, process, product, advertisement, etc.

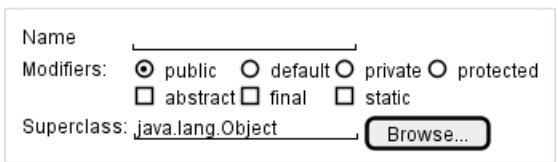
Prototyping ⚡⚡⚡

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.

Prototyping can uncover requirements, in particular, those related to *how users interact with the system*. UI prototypes or *mock ups* are often used in brainstorming sessions, or in meetings with the users to get quick feedback from them.

A mock up (also called a *wireframe* diagram) of a dialog box:



[source: plantuml.com]

Prototyping can be used for *discovering* as well as *specifying* requirements e.g. a UI prototype can serve as a specification of what to build.

Product Surveys ⚡⚡⚡

Studying existing products can unearth shortcomings of existing solutions that can be addressed by a new product. Product manuals and other forms of documentation of an existing system can tell us how the existing solutions work.

> 📦 When developing a game for a mobile device, a look at a similar PC game can give insight into the kind of features and interactions the mobile game can offer.

Specifying requirements

Prose

What ⚡⚡⚡

A textual description (i.e., *prose*) can be used to describe requirements. Prose is especially useful when describing abstract ideas such as the vision of a product.

The product vision of the TEAMMATES Project given below is described using prose.

> TEAMMATES aims to become **the biggest student project in the world** (*biggest* here refers to 'many contributors, many users, large codebase, evolving over a long period'). Furthermore, it aims to serve as a training tool for Software Engineering students who want to learn SE skills in the context of **a non-trivial real software product**.

Avoid using lengthy prose to describe requirements; they can be hard to follow.

Feature lists

What ⚡⚡

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

A sample feature list from a simple Minesweeper game (only a brief description has been provided to save space):

1. Basic play – Single player play.
2. Difficulty levels
   - Medium levels
   - Advanced levels
3. Versus play – Two players can play against each other.
4. Timer – Additional fixed time restriction on the player.
5. ...

User stories

Introduction ⚡

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. [Mike Cohn]

A common format for writing user stories is:

User story format: `As a {user type/role} I can {function} so that {benefit}`

Examples (from a *Learning Management System*):

1. As a student, I can download files uploaded by lecturers, so that I can get my own copy of the files
2. As a lecturer, I can create discussion forums, so that students can discuss things online
3. As a tutor, I can print attendance sheets, so that I can take attendance during the class

You can write user stories using a physical medium or a digital tool. For example, you can use index cards or sticky notes, and arrange them on walls or tables. Alternatively, you can use a software (e.g., GitHub Project Boards, Trello, Google Docs, ...) to manage user stories digitally.

Details

The `{benefit}` can be omitted if it is obvious.

As a user, I can login to the system ~~so that I can access my data~~

It is recommended to confirm there is a concrete benefit even if you omit it from the user story. If not, you could end up adding features that have no real benefit.

You can add more characteristics to the `{user role}` to provide more context to the user story.

- As a forgetful user, I can view a password hint, so that I can recall my password.
- As an expert user, I can tweak the underlying formatting tags of the document, so that I can format the document exactly as I need.

You can write user stories at various levels. High-level user stories, called *epics* (or *themes*) cover bigger functionality. You can then break down these epics to multiple user stories of normal size.

[Epic] As a lecturer, I can monitor student participation levels

- As a lecturer, I can view the forum post count of each student
  so that I can identify the activity level of students in the forum
- As a lecturer, I can view webcast view records of each student
  so that I can identify the students who did not view webcasts
- As a lecturer, I can view file download statistics of each student
  so that I can identify the students who did not download lecture materials

You can add *conditions of satisfaction* to a user story to specify things that need to be true for the user story implementation to be accepted as 'done'.

As a lecturer, *I can* view the forum post count of each student *so that* I can identify the activity level of students in the forum.

Conditions:

☑ Separate post count for each forum should be shown
☐ Total post count of a student should be shown
☑ The list should be sortable by student name and post count

Other useful info that can be added to a user story includes (but not limited to)

- Priority: how important the user story is
- Size: the estimated effort to implement the user story
- Urgency: how soon the feature is needed

Usage ⚡⚡

User stories capture user requirements in a way that is convenient for scoping, estimation, and scheduling.

> [User stories] strongly shift the focus from writing about features to discussing them. In fact, these discussions are more important than whatever text is written. [Mike Cohn, MountainGoat Software 🔗]

User stories differ from traditional requirements specifications mainly in the level of detail. User stories should only provide enough details to make a reasonably low risk estimate of how long the user story will take to implement. When the time comes to implement the user story, the developers will meet with the customer face-to-face to work out a more detailed description of the requirements. [more...]

User stories can capture non-functional requirements too because even NFRs must benefit some stakeholder.

📦 An example of an NFR captured as a user story:

As a/an ___,	I want to ___,	so that ___.
impatient user	to be able to experience reasonable response time from the website while up to 1000 concurrent users are using it	I can use the app even when the traffic is at the maximum expected level

Given their lightweight nature, **user stories are quite handy for recording requirements during early stages of requirements gathering**.

A recipe for brainstorming user stories

Given below is a possible *recipe* you can take when using user stories for early stages of requirement gathering.

Step 0: Clear your mind of preconceived product ideas

Even if you already have some idea of what your product will look/behave like in the end, clear your mind of those ideas. The product is the *solution*. At this point, we are still at the stage of figuring out the *problem* (i.e., user requirements). Let's try to get from the problem to the solution in a systematic way, one step at a time.

Step 1: Define the *target user* as a *persona*:

Decide your target user's profile (e.g. a student, office worker, programmer, salesperson) and work patterns (e.g. Does he work in groups or alone? Does he share his computer with others?). A clear understanding of the target user will help when deciding the importance of a user story. You can even narrow it down to a *persona*. Here is an example:

> Jean is a university student studying in a non-IT field. She interacts with a lot of people due to her involvement in university clubs/societies. ...

Step 2: Define the *problem scope*:

Decide the exact problem you are going to solve for the target user. It is also useful to specify what related problems it will *not* solve so that the exact scope is clear.

> ProductX helps Jean keep track of all her school contacts. It does not cover communicating with contacts.

Step 3: List *scenarios* to form a *narrative*:

Think of the various scenarios your target user is likely to go through as she uses your app. Following a chronological sequence as if you are telling a story might be helpful.

> **A. First use:**
>
> 1. Jean gets to know about ProductX. She downloads it and launches it to check out what it can do.
> 2. After playing around with the product for a bit, Jean wants to start using it for real.
> 3. ...

B. Second use: (Jean is still a beginner)

1. Jean launches ProductX. She wants to find ...
2. ...

C. 10th use: (Jean is a little bit familiar with the app)

1. ...

D. 100th use: (Jean is an expert user)

1. Jean launches the app and does ... and ... followed by ... as per her usual habit.
2. Jean feels some of the data in the app are no longer needed. She wants to get rid of them to reduce clutter.
3.

More examples that might apply to some products:

- Jean uses the app at the start of the day to ...
- Jean uses the app before going to sleep to ...
- Jean hasn't used the app for a while because she was on a three-month training programme. She is now back at work and wants to resume her daily use of the app.
- Jean moves to another company. Some of her clients come with her but some don't.
- Jean starts freelancing in her spare time. She wants to keep her freelancing clients separate from her other clients.

Step 4: List the user stories to support the scenarios:

Based on the scenarios, decide on the user stories you need to support. For example, based on the scenario 'A. First use', you might have user stories such as these:

- As a potential user exploring the app, I can see the app populated with sample data, so that I can easily see how the app will look like when it is in use.
- As a user ready to start using the app, I can purge all current data, so that I can get rid of sample/experimental data I used for exploring the app.

To give another example, based on the scenario 'D. 100th use', you might have user stories such as these:

- As an expert user, I can create shortcuts for tasks, so that I can save time on frequently performed tasks.
- As a long-time user, I can archive/hide unused data, so that I am not distracted by irrelevant data.

Do not 'evaluate' the value of user stories while brainstorming. Reason: an important aspect of brainstorming is not judging the ideas generated.

Other tips:

- **Don't be too hasty to discard 'unusual' user stories:** Those might make your product unique and stand out from the rest, at least for the target users.
- **Don't go into too much detail:** For example, consider this user story: `As a user, I want to see a list of tasks that need my attention most at the present time, so that I pay attention to them first.`
  When discussing this user story, don't worry about what tasks should be considered 'needs my attention most at the present time'. Those details can be worked out later.
- **Don't be biased by preconceived product ideas:** When you are at the stage of identifying user needs, clear your mind of ideas you have about what your end product will look like. That is, don't try to reverse-engineer a preconceived product idea into user stories.
- **Don't discuss implementation details or whether you are actually going to implement it:** When gathering requirements, your decision is whether the user's need is important enough for you to want to fulfil it. Implementation details can be discussed later. If a user story turns out to be too difficult to implement later, you can always omit it from the implementation plan.

While use cases can be recorded on physical paper in the initial stages, an online tool is more suitable for longer-term management of user stories, especially if the team is not co-located.

Use cases

Introduction

Use 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 [📖 : uml-user-guide].

A use case describes an *interaction between the user and the system* for a *specific functionality* of the system.

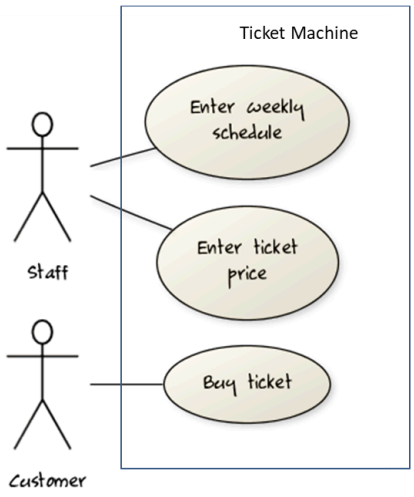
Example 1: 'transfer money' use case for an online banking system

```
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.
```

Example 2: 'upload file' use case of an LMS

UML includes a diagram type called use case diagrams that can illustrate use cases of a system visually, providing a visual 'table of contents' of the use cases of a system.

In the example on the right, note how use cases are shown as ovals and user roles relevant to each use case are shown as stick figures connected to the corresponding ovals.



Use cases capture the *functional requirements* of a system.

Identifying ⚡⚡

A use case is an interaction between a system and its *actors*.

Actors in Use Cases

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.

Some example actors for a Learning Management System:

- Actors: Guest, Student, Staff, Admin, ExamSys, LibSys.

A use case can involve multiple actors.

- Software System: LearnSys
- Use case: UC01 Conduct Survey
- Actors: Staff, Student

An actor can be involved in many use cases.

- Software System: LearnSys
- Actor: Staff
- Use cases: UC01 Conduct Survey, UC02 Set Up Course Schedule, UC03 Email Class, ...

A single person/system can play many roles.

- Software System: LearnSys
- Person: a student
- Actors (or Roles): Student, Guest, Tutor

Many persons/systems can play a single role.

- Software System: LearnSys
- Actor (or role): Student
- Persons that can play this role: undergraduate student, graduate student, a staff member doing a part-time course, exchange student

Use cases can be specified at *various levels of detail*.

Consider the three use cases given below. Clearly, (a) is at a higher level than (b) and (b) is at a higher level than (c).

- System: LearnSys
- Use cases:
  a. Conduct a survey
  b. Take the survey
  c. Answer survey question

While modeling user-system interactions,

- start with high level use cases and progressively work toward lower level use cases.
- be mindful of which level of detail you are working at and not to mix use cases of different levels.

Details

Writing use case steps

The main body of the use case is a sequence of steps that describes the interaction between the system and the actors. Each step is given as a simple statement describing *who does what*.

An example of the main body of a use case.

1. Student requests to upload file
2. LMS requests for the file location
3. Student specifies the file location
4. LMS uploads the file

A use case describes only the externally visible behavior, not internal details, of a system i.e., should minimize details that are not part of the interaction between the user and the system.

This example use case step refers to a behavior not externally visible (i.e., user is not meant to be aware of).

1. LMS saves the file into the cache and indicates success.

A step gives the intention of the actor (not the mechanics). That means UI details are usually omitted. The idea is to leave as much flexibility to the UI designer as possible. That is, the use case specification should be as general as possible (less specific) about the UI.

The first example below is not a good use case step because it contains UI-specific details. The second one is better because it omits UI-specific details.

Bad : User right-clicks the text box and chooses 'clear'

Good : User clears the input

A use case description can show loops too.

An example of how you can show a loop:

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

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.

The *Main Success Scenario (MSS)* describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong. This is also called the *Basic Course of Action* or the *Main Flow of Events* of a use case.

📦 Note how the MSS in the example below assumes that all entered details are correct and ignores problems such as timeouts, network outages etc. For example, the MSS does not tell us what happens if the user enters incorrect data.

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. OBS transfers the money and displays the new account balance.

Use case ends.

***Extensions* are "add-on"s to the MSS that describe *exceptional/alternative flow of events*.** They describe variations of the scenario that can happen if certain things are not as expected by the MSS. Extensions appear below the MSS.

📦 This example adds some extensions to the use case in the previous example.

```
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.
```

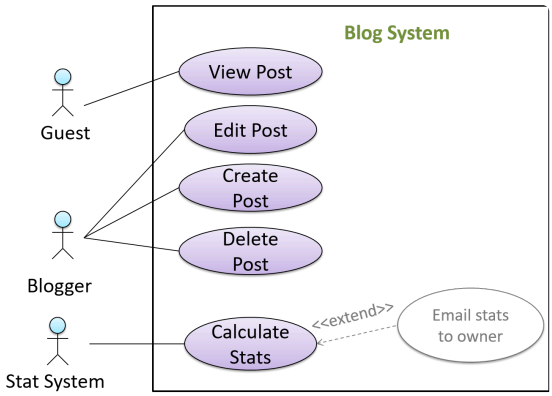
```
*b1. OBS cancels the transfer.
*b2. OBS informs the User of the cancellation.
Use case ends.
```

Note that the numbering style is not a universal rule but a widely used convention. Based on that convention,

- 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 `*`).

When separating extensions from the MSS, keep in mind that the **MSS should be self-contained**. That is, the MSS should give us a complete usage scenario.

Also note that it is not useful to mention events such as power failures or system crashes as extensions because the system cannot function beyond such catastrophic failures.



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

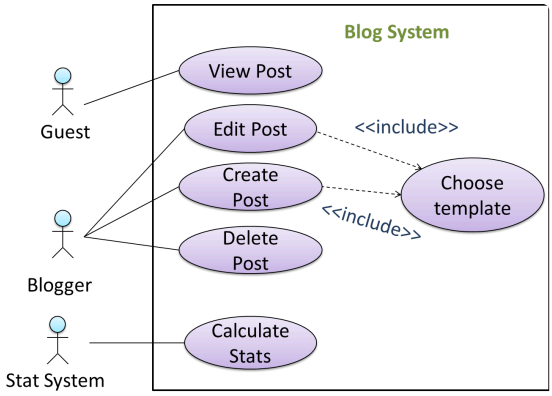
This use case includes two other use cases, one in step 1 and one in step 2.

- 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.

You use a dotted arrow and an `<<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 you expect the system to be in before the use case starts.**

Software System: Online Banking System
Use case: UC23 - Transfer Money
Actor: User
Preconditions: User is logged in
MSS:

1. User chooses to transfer money.
2. OBS requests for details for the transfer.
   ...

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.

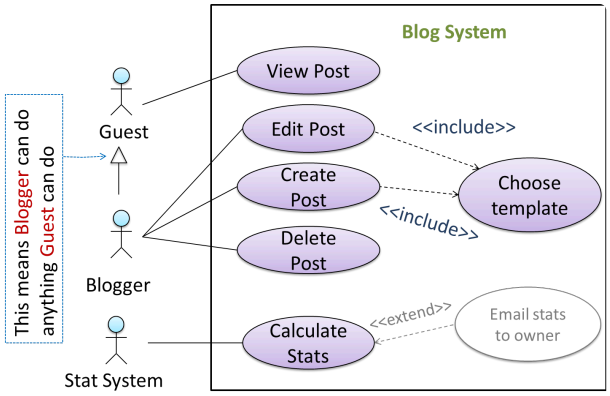
MSS:

1. User chooses to transfer money.
2. OBS requests for details for the transfer.
   ...

Usage ⚡⚡⚡

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

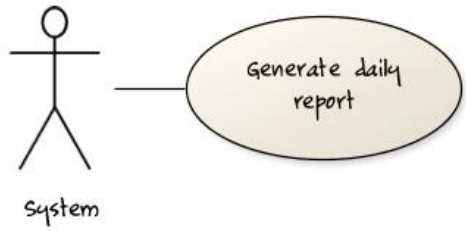
📦 In this example, actor `Blogger` can do all the use cases the actor `Guest` can do, as a result of the actor generalization relationship given in the diagram.



Do not over-complicate use case diagrams by trying to include everything possible. A use case diagram is a brief summary of the use cases that is used as a starting point. Details of the use cases can be given in the use case descriptions.

Some include 'System' as an actor to indicate that something is done by the system itself without being initiated by a user or an external system.

The diagram below can be used to indicate that the system generates daily reports at midnight.



However, others argue that only use cases providing value to an external user/system should be shown in the use case diagram. For example, they argue that `view daily report` should be the use case and `generate daily report` is not to be shown in the use case diagram because it is simply something the system has to do to support the `view daily report` use case.

You are recommended to follow the latter view (i.e., not to use System as a user). Limit use cases for modeling behaviors that involve an external actor.

UML is not very specific about the text contents of a use case. Hence, there are many styles for writing use cases. For example, the steps can be written as a continuous paragraph.

Use cases should be easy to read. Note that there is no strict rule about writing all details of all steps or a need to use all the elements of a use case.

There are some advantages of documenting system requirements as use cases:

- 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.

One of the main disadvantages of use cases is that they are not good for capturing requirements that do not involve a user interacting with the system. Hence, they should not be used as the sole means to specify requirements.

Glossary

What ⚡⚡⚡

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

Here is a partial glossary from a variant of the *Snakes and Ladders* game:

- Conditional square: A square that specifies a specific face value which a player has to throw before his/her piece can leave the square.
- Normal square: a normal square does not have any conditions, snakes, or ladders in it.

Supplementary requirements

What ⚡⚡⚡

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.

SECTION: DESIGN

Software design

Introduction

What ⚡⚡⚡

> Design is the creative process of transforming the problem into a solution; the solution is also called design. -- 📖 *Software Engineering Theory and Practice*, Shari Lawrence; Atlee, Joanne M. Pfleeger

Software design has two main aspects:

- **Product/external design: designing the external behavior of the product to meet the users' requirements.** This is usually done by product designers with input from business analysts, user experience experts, user representatives, etc.
- **Implementation/internal design: designing how the product will be implemented to meet the required external behavior.** This is usually done by software architects and software engineers.

Design fundamentals

Abstraction

What ⚡⚡

Abstraction is 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.

The guiding principle of abstraction is that only details that are relevant to the current perspective or the task at hand need to be considered. As most programs are written to solve complex problems involving large amounts of intricate details, it is impossible to deal with all these details at the same time. That is where abstraction can help.

Data abstraction **: abstracting away the lower level data items and thinking in terms of bigger entities**

📦 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**

📦 `print("Hello")` is an abstraction of the actual output mechanism within the computer.

Abstraction can be applied repeatedly to obtain progressively *higher levels of abstraction*.

📦 An example of different levels of data abstraction: a `File` is a data item that is at a higher level than an array and an array is at a higher level than a bit.

📦 An example of different levels of control abstraction: `execute(Game)` is at a higher level than `print(Char)` which is at a higher level than an Assembly language instruction `MOV` .

Abstraction is a general concept that is not limited to just data or control abstractions.

📦 Some more general examples of abstraction:

- 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.

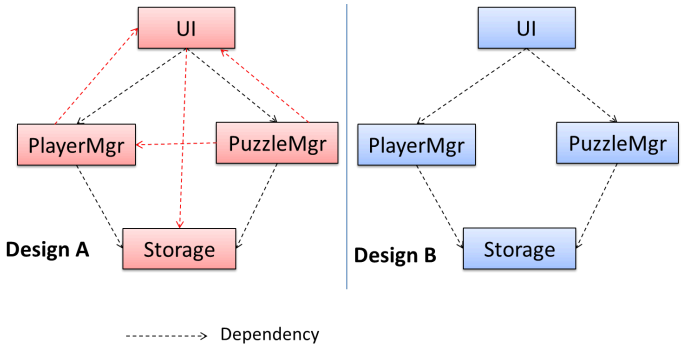
Coupling

What ⚡⚡

Coupling **is 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** due to the following disadvantages:

- **Maintenance is harder** because a change in one module could cause changes in other modules coupled to it (i.e., a ripple effect).
- **Integration is harder** because 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.

📦 In the example below, design `A` appears to have more coupling between the components than design `B` .



How ⚡⚡⚡

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()`.

code for the above example

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

Cohesion

What ⚡⚡

***Cohesion* is 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 cohesion (aka *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 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.

How ⚡⚡⚡

Cohesion can be present in many forms. Some examples:

- 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 initializing 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.

Suppose a Payroll application contains a class that deals with writing data to the database. If the class includes some code to show an error dialog to the user if the database is unreachable, that class is not cohesive because it seems to be interacting with the user as well as the database.

Modeling

Introduction

What ⚡⚡

A *model* is a representation of something else.

> 📦 A class diagram is a model that represents a software design.

A *model* provides a simpler view of a complex entity because a model captures only a selected aspect. This omission of some aspects implies models are abstractions.

> 📦 A class diagram captures the structure of the software design but not the behavior.

Multiple models of the same entity may be needed to capture it fully.

> 📦 In addition to a class diagram (or even multiple class diagrams), a number of other diagrams may be needed to capture various interesting aspects of the software.

How ⚡⚡⚡

In software development, models are useful in several ways:

a) To analyze a complex entity related to software development.

> 📦 Some examples of using models for analysis:
>
> 1. Models of the problem domain can be built to aid the understanding of the problem to be solved.
> 2. When planning a software solution, models can be created to figure out how the solution is to be built. An architecture diagram is such a model.

b) To communicate information among stakeholders. Models can be used as a visual aid in discussions and documentation.

> 📦 Some examples of using models to communicate:
>
> 1. You can use an *architecture diagram* to explain the high-level design of the software to developers.
> 2. A business analyst can use a *use case diagram* to explain to the customer the functionality of the system.
> 3. A *class diagram* can be reverse-engineered from code so as to help explain the design of a component to a new developer.

c) As a blueprint for creating software. Models can be used as instructions for building software.

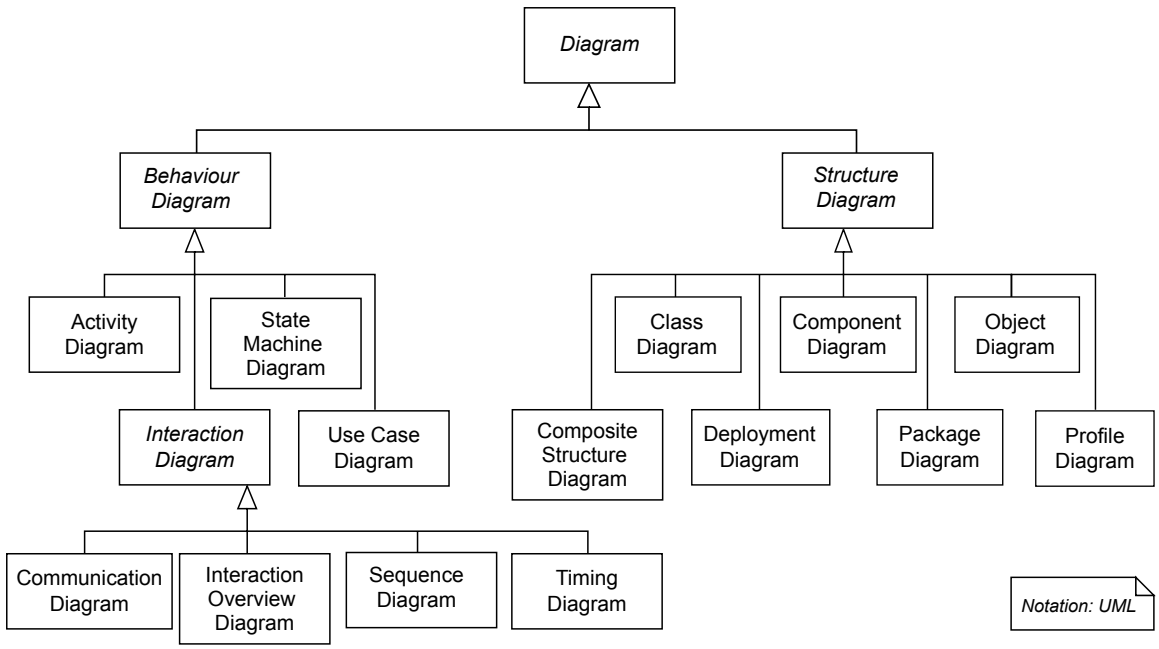
> 📦 Some examples of using models as blueprints:
>
> 1. A senior developer draws a class diagram to propose a design for an OOP software and passes it to a junior programmer to implement.
> 2. A software tool allows users to draw UML models using its interface and the tool automatically generates the code based on the model.

> ▶ Model Driven Development + extra

UML Models ⚡⚡⚡

Unified Modeling Language (UML) is a graphical notation to describe various aspects of a software system. UML is the brainchild of three software modeling specialists James Rumbaugh, Grady Booch and Ivar Jacobson (also known as the Three Amigos). Each of them had developed their own notation for modeling software systems before joining forces to create a unified modeling language (hence, the term 'Unified' in UML). UML is currently the most commonly used modeling notation used in the software industry.

The following diagram uses the class diagram notation to show the different types of UML diagrams.



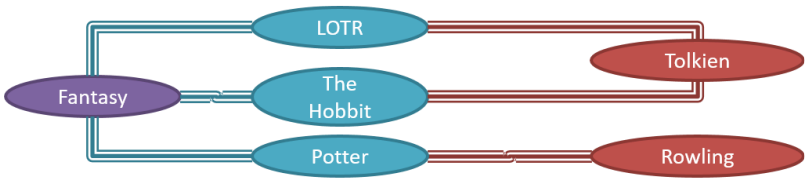
source:https://en.wikipedia.org/

Modeling structures

OO Structures ⚡

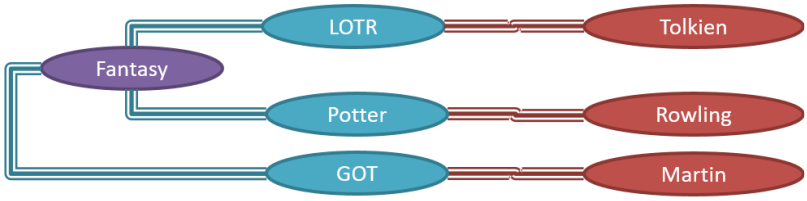
An OO solution is basically a network of objects interacting with each other. Therefore, **it is useful to be able to model how the relevant objects are 'networked' together** inside a software i.e., how the objects are connected together.

Given below is an illustration of some objects and how they are connected together. Note: the diagram uses an ad-hoc notation.



Note that these **object structures within the same software can change over time**.

Given below is how the object structure in the previous example could have looked like at a different time.

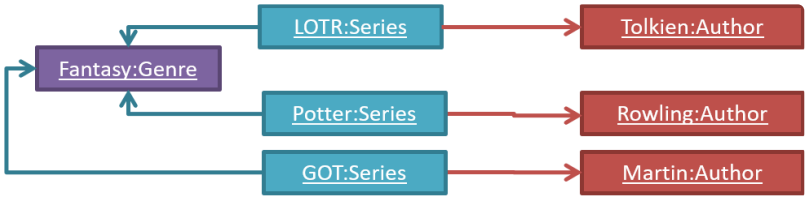


However, object structures do not change at random; they change based on a set of rules set by the designer of that software. Those **rules that object structures need to follow can be illustrated as a *class structure*** i.e., a structure that exists among the relevant classes.

Here is a class structure (drawn using an ad-hoc notation) that matches the object structures given in the previous two examples. For example, note how this class structure does not allow any connection between `Genre` objects and `Author` objects, a rule followed by the two object structures above.

UML *Object Diagrams* model object structures. UML *Class Diagrams* model class structures.

Here is an object diagram for the above example:



And here is the class diagram for it:

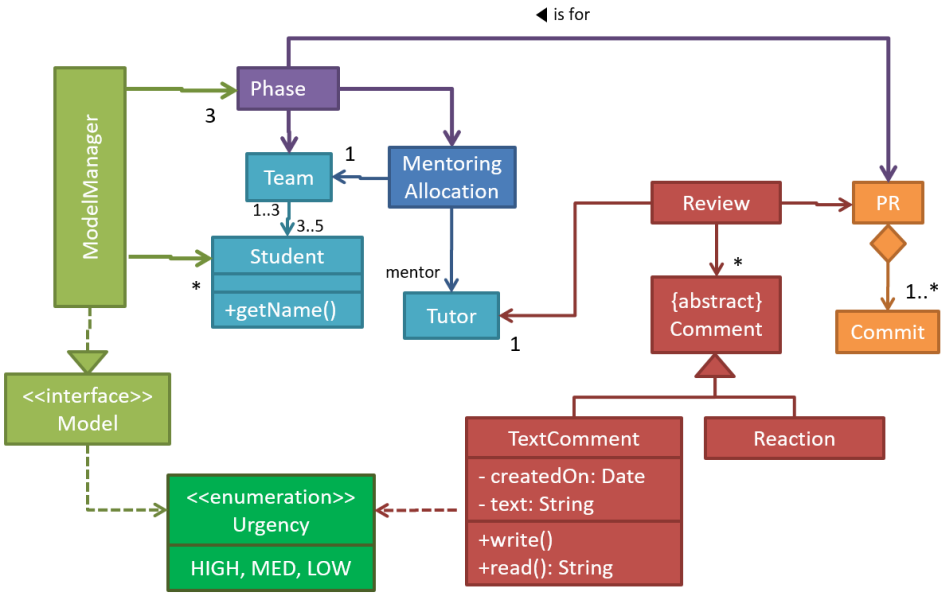
Class Diagrams (Basics)

Classes form the basis of class diagrams.

UML Class Diagrams → Introduction → What

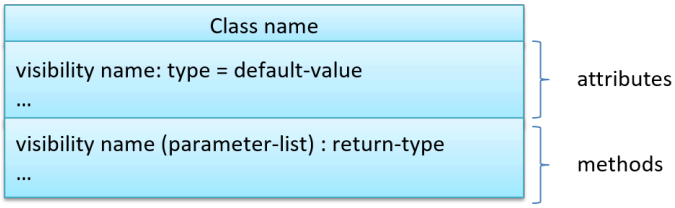
UML *class diagrams* describe the structure (but not the behavior) of an OOP solution. These are possibly the most often used diagrams in the industry and are an indispensable tool for an OO programmer.

An example class diagram:

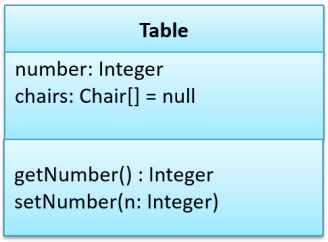


UML Class Diagrams → Classes → What

The basic UML notations used to represent a *class*:

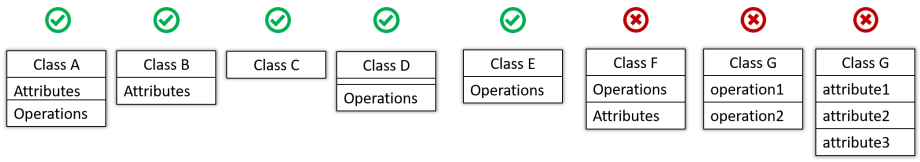


A `Table` class shown in UML notation:



The equivalent code

The 'Operations' compartment and/or the 'Attributes' compartment may be omitted if such details are not important for the task at hand. Similarly, *some* attributes/operations can be omitted if not relevant to the purpose of the diagram. '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.

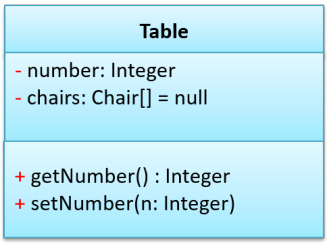


The *visibility* of attributes and operations is used to indicate the level of access allowed for each attribute or operation. The types of visibility and their exact meanings depend on the programming language used. Here are some common visibilities and how they are indicated in a class diagram:

- `+` : public
- `–` : private
- `#` : protected
- `~` : package private

▶ How visibilities map to programming language features

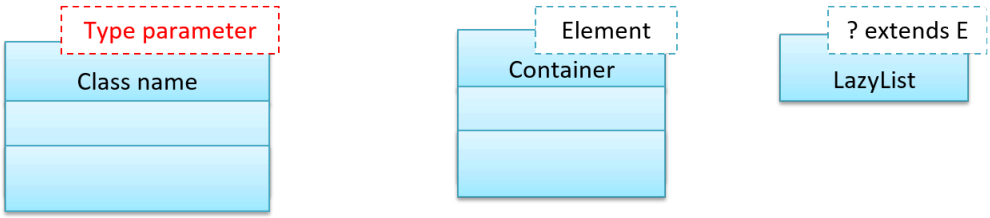
`Table` class with visibilities shown:



▶ The equivalent code

! **There is no *default* visibility** in UML. If a class diagram does not show the visibility of a member, it simply means the *visibility is unspecified* (for reasons such as the visibility not being decided yet or it being not important to the purpose of the diagram).

Generic classes can be shown as given below. The notation format is shown on the left, followed by two examples.



❓ Exercises:

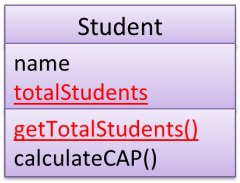
❓ Which classes are correct?

❓ Draw Car class

UML Class Diagrams → Class-Level Members → What

In UML class diagrams, **underlines denote class-level attributes and methods.**

In the class diagram below, the `totalStudents` attribute and the `getTotalStudents` method are class-level.



***Associations* are the main connections among the classes in a class diagram.**

OOP Associations → What

Objects in an OO solution need to be connected to each other to form a network so that they can interact with each other. Such **connections between objects are called *associations*.**

Suppose an OOP program for managing a learning management system creates an object structure to represent the related objects. In that object structure you can expect to have associations between a `Course` object that represents a specific course and `Student` objects that represent students taking that course.

Associations in an object structure can change over time.

To continue the previous example, the associations between a `Course` object and `Student` objects can change as students enroll in the course or drop the course over time.

Associations among objects can be generalized as associations between the corresponding classes too.

In our example, as some `Course` objects can have associations with some `Student` objects, you can view it as an association between the `Course` class and the `Student` class.

Implementing associations

You use instance level variables to implement associations.

In our example, the `Course` class can have a `students` variable to keeps track of students associated with a particular course.

UML Class Diagrams → Associations → What

You should use a solid line to show an association between two classes.

This example shows an association between the `Admin` class and the `Student` class:

UML Class Diagrams → Associations as Attributes

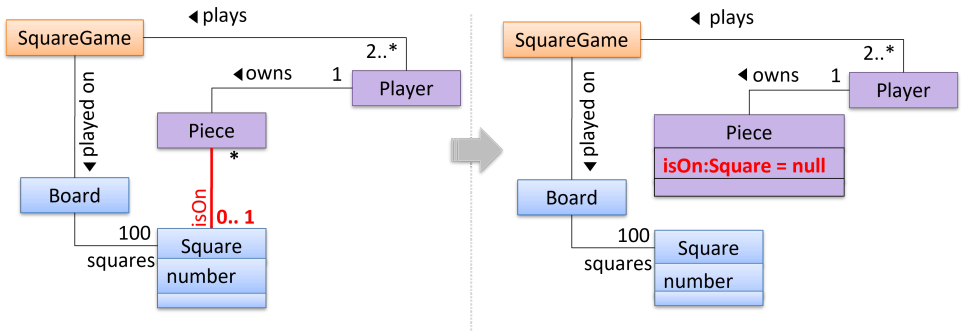
An association can be shown as an attribute instead of a line.

Association multiplicities and the default value can be shown as part of the attribute using the following notation. Both are optional.

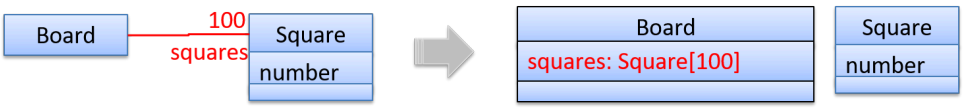
```
name: type [multiplicity] = default value
```

The diagram below depicts a multi-player *Square Game* being played on a board comprising of 100 squares. Each of the squares may be occupied with any number of pieces, each belonging to a certain player.

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`.



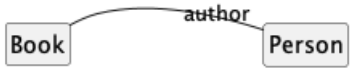
The association that a `Board` has 100 `Square`s can be shown in either of these two ways:



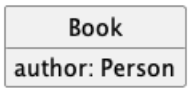
Show each association as **either an attribute or a line but not both**. A line is preferred as it is easier to spot.

Diagram (a) given below shows the 'author' association between the `Book` class and the `Person` class as a line while (b) shows the same association as an attribute in the `Book` class. Both are correct and the two are equivalent. But (c) is not correct as it uses both a line and an attribute to show the same association.

✓ (a)



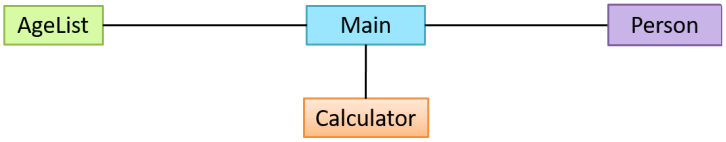
✓ (b)



✗ (c)

The most basic class diagram is a bunch of classes with some solid lines among them to represent associations, such as this one.

An example class diagram showing associations between classes.

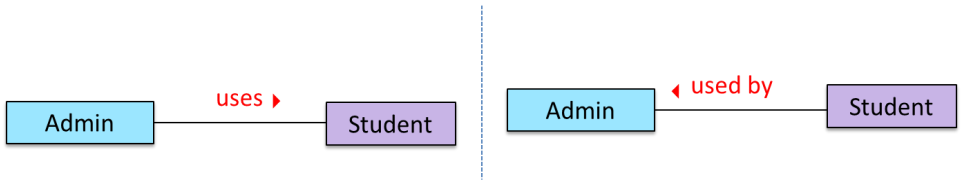


In addition, **associations can show additional decorations such as *association labels*, *association roles*, *multiplicity* and *navigability*** to add more information to a class diagram.

UML Class Diagrams → Associations → Labels

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

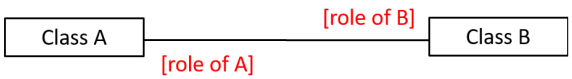
In this example, the same association is described using two different labels.



- 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.

UML Class Diagrams → Associations → Roles

***Association Role* are used to indicate the role played by the classes in the association.**

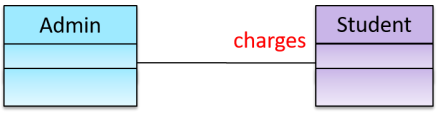


This association represents a marriage between a `Man` object and a `Woman` object. The respective roles played by objects of these two classes are `husband` and `wife`.

Note how the variable names match closely with the association roles.

```
class Man {
    Woman wife;
}

class Woman {
    Man husband;
}
```

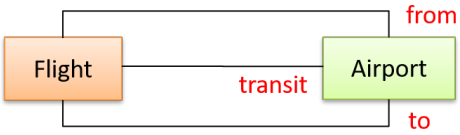
The role of `Student` objects in this association is `charges` (i.e., Admin is in charge of students)




```
class Admin {
    List<Student> charges;
}
```

Association roles are optional to show. They are particularly useful for differentiating among multiple associations between the same two classes.

In each the three associations between the `Flight` class and the `Airport` class given below, the `Airport` class plays a different role.



OOP Associations → Navigability

When two classes are linked by an association, it does not necessarily mean both objects taking part in an instance of the association *knows about* (i.e., has a reference to) each other. **The concept of *navigability* tells us if an object taking part in association knows about the other.** In other words, it tells us if we can 'navigate' from the one object to the other in a given direction -- because if the object 'knows' about the other, it has a reference to the other object, and we can use that reference to 'navigate to' (i.e., access) that other object.

Navigability can be *unidirectional* or *bidirectional*. Suppose there is an association between the classes `Box` and `Rope`, and the `Box` object `b` and the `Rope` object `r` is taking part in one instance of that association.

- **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`. That is, one can *navigate* from `b` to `r` using the `b`'s object reference of `r` (but not in the other direction). 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 for the same single instance of the association.

Note that two unidirectional associations in opposite directions do not add up to a single bidirectional association.

In the code below, there is a bidirectional association between the `Person` class and the `Cat` class i.e., if `Person p` is the owner of the `Cat c`, `p` it will result in `p` and `c` having references to each other.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person owner;
    //...
}
```

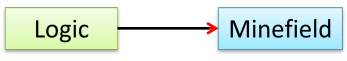
The code below has two unidirectional associations between the `Person` class and the `Cat` class (in opposite directions). Because the breeder is not necessarily the same person keeping the cat as a pet, they are two separate associations, not a bidirectional association.

```
class Person {
    Cat pet;
    //...
}

class Cat{
    Person breeder;
    //...
}
```

UML Class Diagrams → Associations → Navigability

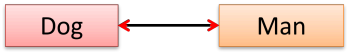
Use arrowheads to indicate the navigability of an association.

In this example, the navigability is unidirectional, and is from the `Logic` class to the `Minefield` class. That means if a `Logic` object `L` is associated with a `Minefield` object `M`, `L` has a reference to `M` but `M` doesn't have a reference to `L`.




```
class Logic {
    Minefield minefield;
    // ...
}

class Minefield {
    //...
}
```

Here is an example of a bidirectional navigability; i.e., if a `Dog` object `d` is associated with a `Man` object `m`, `d` has a reference to `m` and `m` has a reference to `d`.



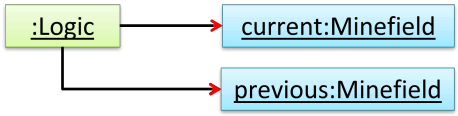

```
class Dog {
    Man man;
    // ...
}

class Man {
    Dog dog;
    // ...
}
```

The arrowhead (not the entire arrow) denotes the navigability. The line denotes the association, as before. So, the navigability (i.e., the arrowhead) is an extra annotation added to an association line.

For example, both diagrams below show an association between `Foo` and `Bar`. But the one on the right shows the navigability as well.

Navigability can be shown in class diagrams as well as object diagrams.

According to this object diagram, the given `Logic` object is associated with and aware of two `MineField` objects.



Exercises:

What does the navigability given by this diagram mean?

OOP Associations → Multiplicity

***Multiplicity* is the aspect of an OOP solution that dictates how many objects take part in each association.**

The multiplicity of the association between `Course` objects and `Student` objects tells you how many `Course` objects can be associated with one `Student` object and vice versa.

Implementing multiplicity

A normal instance-level variable gives us a `0..1` multiplicity (also called *optional associations*) because a variable can hold a reference to a single object or `null`.

In the code below, the `Logic` class has a variable that can hold `0..1` i.e., zero or one `Minefield` objects.

```
class Logic {
    Minefield minefield;
    // ...
}

class Minefield {
    //...
}
```

A variable can be used to implement a `1` multiplicity too (also called *compulsory associations*).

In the code below, the `Logic` class will always have a `ConfigGenerator` object, provided the variable is not set to `null` at some point.

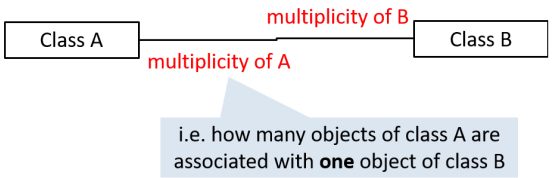
```
class Logic {
    ConfigGenerator cg = new ConfigGenerator();
    ...
}
```

To implement other multiplicities, choose a suitable data structure such as Arrays, ArrayLists, HashMaps, Sets, etc.

This code uses a two-dimensional array to implement a 1-to-many association from the `Minefield` to `Cell`.

```
class Minefield {
    Cell[][] cell;
    //...
}
```

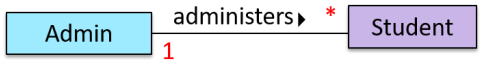
UML Class Diagrams → Associations → Multiplicity



Commonly used multiplicities:

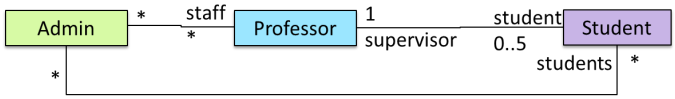
- `0..1` : *optional*, can be linked to 0 or 1 objects.
- `1` : *compulsory*, must be linked to one object at all times.
- `*` : can be linked to 0 or more objects.
- `n..m` : the number of linked objects must be within `n` to `m` inclusive e.g., `2..5`, `1..*` (one or more), `*..5` (up to five)

In the diagram below, 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.



In the diagram below,

- Each student must be supervised by exactly one professor. i.e., There cannot be a student who doesn't have a supervisor or has multiple supervisors.
- A professor cannot supervise more than 5 students but can have no students to supervise.
- An admin can handle any number of professors and any number of students, including none.
- A professor/student can be handled by any number of admins, including none.

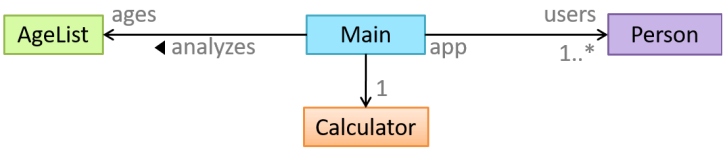


! **There is no *default* multiplicity** in UML. If a class diagram does not show the multiplicity of an association, it simply means the *multiplicity is unspecified*.

Exercises:

Which statement agrees with the multiplicity shown in this diagram?

Here is the same class diagram shown earlier but with some additional information included:



Adding More Info to UML Models

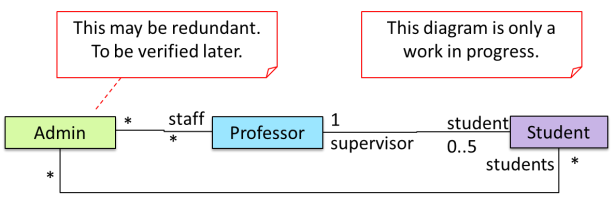
UML notes can be used to add more info to any UML model.

UML → Notes

Notes

UML notes can augment UML diagrams with additional information. These notes can be shown connected to a particular element in the diagram or can be shown without a connection. The diagram below shows examples of both.

Example:



Class Diagrams - Intermediate ⚡⚡

A class diagram can also show different types of relationships between classes: inheritance, compositions, aggregations, dependencies.

Modeling inheritance

▼ 🎓 OOP → Inheritance → What

The OOP concept *Inheritance* allows you to define a new class based on an existing class.

📦 For example, you can use inheritance to define an `EvaluationReport` class based on an existing `Report` class so that the `EvaluationReport` class does not have to duplicate data/behaviors that are already implemented in the `Report` class. The `EvaluationReport` can inherit the `wordCount` attribute and the `print()` method from the *base class* `Report` .

- Other names for Base class: *Parent* class, *Superclass*
- Other names for Derived class: *Child* class, *Subclass*, *Extended* class

A superclass is said to be *more general* than the subclass. Conversely, a subclass is said to be more *specialized* than the superclass.

Applying inheritance on a group of similar classes can result in the common parts among classes being extracted into more general classes.

📦 `Man` and `Woman` behave the same way for certain things. However, the two classes cannot be simply replaced with a more general class `Person` because of the need to distinguish between `Man` and `Woman` for certain other things. A solution is to add the `Person` class as a superclass (to contain the code common to men and women) and let `Man` and `Woman` inherit from `Person` class.

Inheritance implies the derived class can be considered as a *subtype* of the base class (and the base class is a *super-type* of the derived class), resulting in an *is a* relationship.

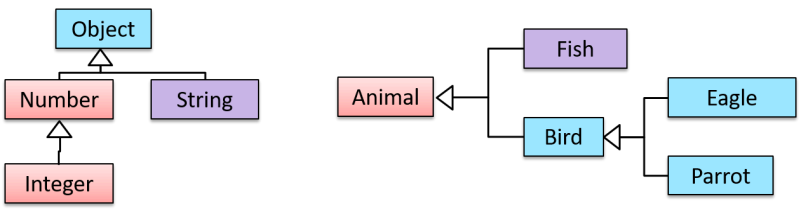
ℹ️ Inheritance does not necessarily mean a subtype relationship exists. However, the two often go hand-in-hand. For simplicity, at this point let us assume inheritance implies a subtype relationship.

📦 To continue the previous example,

- `Woman` *is a* `Person`
- `Man` *is a* `Person`

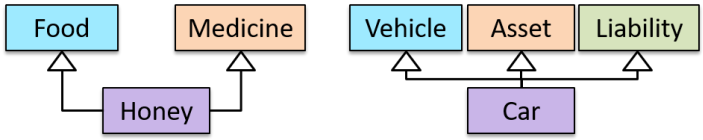
Inheritance relationships through a chain of classes can result in inheritance *hierarchies* (aka inheritance *trees*).

📦 Two inheritance hierarchies/trees are given below. Note that the triangle points to the parent class. Observe how the `Parrot` *is a* `Bird` as well as it *is an* `Animal` .



***Multiple Inheritance* is when a class inherits *directly* from multiple classes.** Multiple inheritance among classes is allowed in some languages (e.g., Python, C++) but not in other languages (e.g., Java, C#).

The `Honey` class inherits from the `Food` class *and* the `Medicine` class because honey can be consumed as a food as well as a medicine (in some oriental medicine practices). Similarly, a `Car` is a `Vehicle`, an `Asset` and a `Liability`.



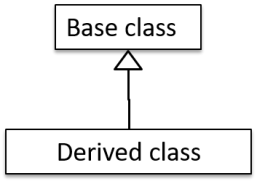
Exercises:

Which statements are correct?

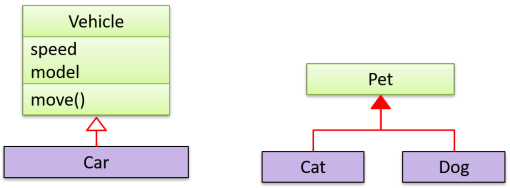
UML → Class Diagrams → Inheritance → What

You can use a triangle and a solid line (not to be confused with an arrow) to indicate class inheritance.

Notation:



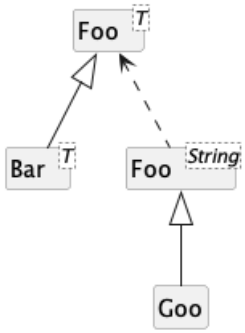
Examples: The `Car` class *inherits* from the `Vehicle` class. The `Cat` and `Dog` classes *inherit* from the `Pet` class.



It does not matter whether the triangle is filled or empty.

Here's an example that combines inheritance with generics:

```
class Foo<T> {
}

class Bar<T> extends Foo<T> {
}

class Goo extends Foo<String> {
}
```



Modeling composition

OOP → Associations → Composition

A composition is an association that represents a strong *whole-part* relationship.

A `Board` (used for playing board games) consists of `Square` objects.

Composition implies,

1. **when the *whole* is destroyed, *parts* are destroyed too** i.e., the *part* cannot exist without being attached to a *whole*.
2. **there cannot be cyclical links**.

The 'sub-folder' association between `Folder` objects is a composition type association. Consider the case of `Folder` object `subF` is a sub-folder of `Folder` object `F`. In this case,

1. if `F` is deleted, `subF` will be deleted with it.
2. `F` cannot be a sub-folder of `subF` (i.e., no cyclical 'sub-folder' association between the two objects).

Whether a relationship is a composition can depend on the context.

Is the relationship between `Email` and `EmailSubject` composition? That is, is the email subject *part* of an email to the extent that an email subject cannot exist without an email?

- When modeling an application that sends emails, the answer is 'yes'.
- When modeling an application that gather analytics about email traffic, the answer may be 'no' (e.g., the application might collect just the email subjects for text analysis).

A common use of composition is when parts of a big class are carved out as smaller classes for the ease of managing the internal design. In such cases, the classes extracted out still act as *parts* of the bigger class and the outside world has no business knowing about them.

***Cascading deletion* alone is not sufficient for composition.** Suppose there is a design in which `Person` objects are attached to `Task` objects and the former get deleted whenever the latter is deleted. This fact alone does not mean there is a composition relationship between the two classes. For it to be composition, a `Person` must be an integral *part* of a `Task` in the context of that association, at the concept level (not simply at implementation level).

Identifying and keeping track of composition relationships in the design has benefits such as helping to maintain the data integrity of the system. For example, when you know that a certain relationship is a composition, you can take extra care in your implementation to ensure that when the *whole* object is deleted, all its *parts* are deleted too.

Implementing composition

Composition is implemented using a normal variable. If correctly implemented, the 'part' object will be deleted when the 'whole' object is deleted. Ideally, the 'part' object may not even be visible to clients of the 'whole' object.

Here is one way to implement the composition between `Email` and `Subject`:

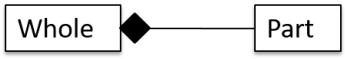
```
class Email {
    private Subject subject;
  ...
}
```

In this code, the `Email` has a composition type relationship with the `Subject` class, in the sense that the subject is part of the email.

UML → Class Diagrams → Composition → What

UML uses a solid diamond symbol to denote composition.

Notation:



A `Book` is *composed* of `Chapter` objects. As a result, when the `Book` object is destroyed, its `Chapter` objects are destroyed too.

Modeling aggregation

OOP → Associations → Aggregation

***Aggregation* represents a *container-contained* relationship.** It is a weaker relationship than composition.

`SportsClub` can act as a *container* for `Person` objects who are members of the club. `Person` objects can survive without a `SportsClub` object.

Implementing aggregation

Implementation is similar to that of composition except the *containee* object can exist even after the *container* object is deleted.

In the code below, there is an aggregation association between the `Team` class and the `Person` class in that a `Team` contains a `Person` object who is the leader of the team.

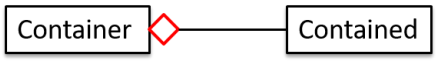
```
class Team {
    Person leader;
    ...
    void setLeader(Person p) {
```

```
        leader = p;
    }
}
```

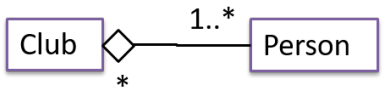
UML → Class Diagrams → Aggregation → What

UML uses a hollow diamond to indicate an aggregation.

Notation:



Example:



Aggregation vs Composition

The distinction between composition (◆) and aggregation (◇) is rather blurred. Martin Fowler's famous book *UML Distilled* advocates omitting the aggregation symbol altogether because using it adds more confusion than clarity.

Exercises:

Which one is not recommended to use?

Modeling dependencies

OOP → Associations → Dependencies

In the context of OOP associations, **a *dependency* is a need for one class to depend on another without having a direct association in the same direction.** Reason for the exclusion: If there is an association from class `Foo` to class `Bar` (i.e., navigable from `Foo` to `Bar`), that means `Foo` is *obviously* dependent on `Bar` and hence there is no point in mentioning *dependency* specifically. In other words, we are specifically focusing on *non-obvious* dependencies here. One cause of such dependencies is interactions between objects that do not have a long-term link between them.

A `Course` class can have a dependency on a `Registrar` class because the `Course` class needs to refer to the `Registrar` class to obtain the maximum number of students it can support (e.g., `Registrar.MAX_COURSE_CAPACITY`).

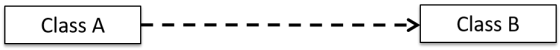
In the code below, `Foo` has a dependency on `Bar` but it is not an association because it is only a transient interaction and there is no long term relationship between a `Foo` object and a `Bar` object. i.e., the `Foo` object

does not keep the `Bar` object it receives as a parameter.

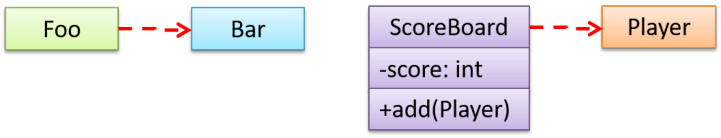
```
class Foo {

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

class Bar {
    int value;

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

▼ 🎓 UML → Class Diagrams → Dependencies → What

UML uses a dashed arrow to show dependencies.



📦 Two examples of dependencies:



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** (for instance, as an association or an inheritance) e.g., class `Foo` accessing a constant in `Bar` but there is no association/inheritance from `Foo` to `Bar`.

A class diagram can also show different types of class-like entities:

Modeling enumerations

▼ 🎓 OOP → Classes → Enumerations

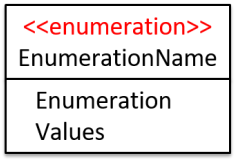
An *Enumeration* is a fixed set of values that can be considered as a data type. An enumeration is often useful when using a regular data type such as `int` or `String` would allow invalid values to be assigned to a variable.

Suppose you want a variable called `priority` to store the priority of something. There are only three priority levels: high, medium, and low. You can declare the variable `priority` as of type `int` and use only values `2`, `1`, and `0` to indicate the three priority levels. However, this opens the possibility of an invalid value such as `9` being assigned to it. But if you define an enumeration type called `Priority` that has three values `HIGH`, `MEDIUM` and `LOW` only, a variable of type `Priority` will never be assigned an invalid value because the compiler is able to catch such an error.

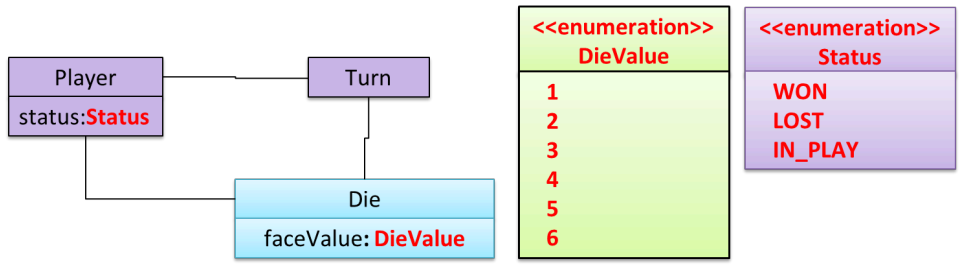
```
Priority : HIGH , MEDIUM , LOW
```

UML → Class Diagrams → Enumerations → What

Notation:



In the class diagram below, there are two enumerations in use:



Exercises:

Define WeekDay Enum

Modeling abstract classes

OOP → Inheritance → Abstract Classes

Abstract class: A class declared as an *abstract class* cannot be instantiated, but it can be subclassed.

You can declare a class as *abstract* when a class is merely a representation of commonalities among its subclasses in which case it does not make sense to instantiate objects of that class.

The `Animal` class that exists as a generalization of its subclasses `Cat`, `Dog`, `Horse`, `Tiger` etc. can be declared as abstract because it does not make sense to instantiate an `Animal` object.

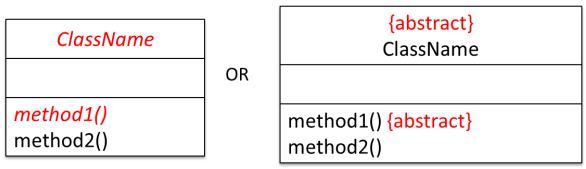
Abstract method: An *abstract method* is a method signature without a method implementation.

The `move` method of the `Animal` class is likely to be an abstract method as it is not possible to implement a `move` method at the `Animal` class level to fit all subclasses because each animal type can move in a different way.

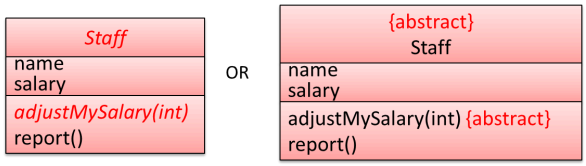
A class that has an abstract method becomes an abstract class because the class definition is incomplete (due to the missing method body) and it is not possible to create objects using an incomplete class definition.

UML → Class Diagrams → Abstract Classes → What

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



Example:



Modeling interfaces

OOP → Inheritance → Interfaces

An *interface* is a behavior specification i.e., a collection of method specifications. If a class implements the interface, it means the class is able to support the behaviors specified by the said interface.

> There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts. ..
>
> Oracle Docs on Java

Suppose `SalariedStaff` is an interface that contains two methods `setSalary(int)` and `getSalary()`. `AcademicStaff` can declare itself as *implementing* the `SalariedStaff` interface, which means the `AcademicStaff` class must implement all the methods specified by the `SalariedStaff` interface i.e., `setSalary(int)` and `getSalary()`.

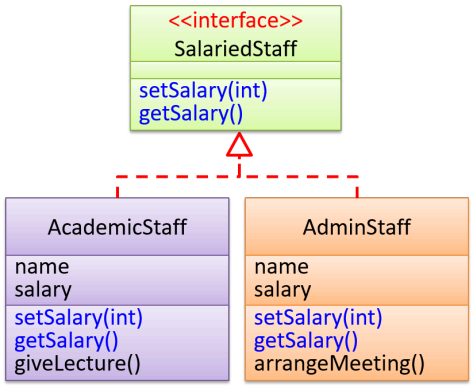
A class implementing an interface results in an *is-a* relationship, just like in class inheritance.

In the example above, `AcademicStaff` *is a* `SalariedStaff`. An `AcademicStaff` object can be used anywhere a `SalariedStaff` object is expected e.g., `SalariedStaff ss = new AcademicStaff()`.

UML → Class Diagrams → Interfaces → What

An interface is shown similar to a class with an additional keyword `<<interface>>`. When a class implements an interface, it is shown similar to class inheritance except a dashed line is used instead of a solid line.

The `AcademicStaff` and the `AdminStaff` classes *implement* the `SalariedStaff` interface.



Class Diagrams - Advanced ϟϟϟ

A class diagram can show association classes too.

OOP → Associations → Association Classes

An *association class* represents additional information about an association. It is a normal class but plays a special role from a design point of view.

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. In such situations, an additional association class can be introduced, e.g., a `Marriage` class, to store such information.

Implementing association classes

There is no special way to implement an association class. It can be implemented as a normal class that has variables to represent the endpoint of the association it represents.

In the code below, the `Transaction` class is an association class that represents a transaction between a `Person` who is the seller and another `Person` who is the buyer.

```
class Transaction {

    //all fields are compulsory
```

```
    Person seller;
    Person buyer;
    Date date;
    String receiptNumber;

    Transaction(Person seller, Person buyer, Date date, String receiptNumber) {
        //set fields
    }
}
```

❓✅ Exercises:

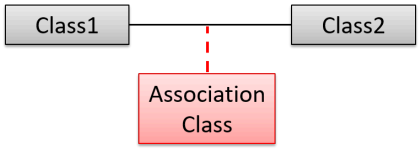
❓✅ Which are suitable as an Association Class?

✖ ⌄

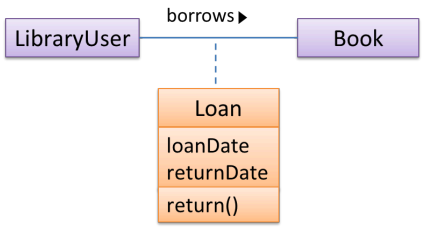
⌃

⌄ 🎓 UML → Class Diagrams → Association Classes → What

Association classes are denoted as a connection to an association link using a dashed line as shown below.



📦 In this example `Loan` is an association class because it stores information about the `borrows` association between the `User` and the `Book`.



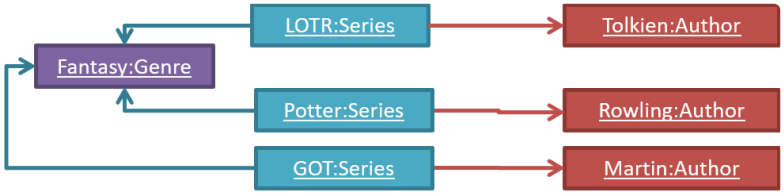
⌃

Object Diagrams

⌄ 🎓 UML → Object Diagrams → Introduction

An object diagram shows an object structure at a given point of time.

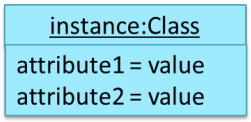
📦 An example object diagram:



Object diagrams can be used to complement class diagrams. For example, you can use object diagrams to model different object structures that can result from a design represented by a given class diagram.

UML → Object Diagrams → Objects

Notation:



Notes:

- The class name and object name are underlined e.g., `car1:Car` .
- `objectName:ClassName` is meant to say 'an instance of `ClassName` identified as `objectName` '.
- Unlike classes, there is no compartment for methods.
- *Attributes* compartment can be omitted if it is not relevant to the purpose of the diagram.
- Object name can be omitted too e.g., `:Car` which is meant to say 'an *unnamed* instance of a `Car` object'.

Some example objects:

Exercises:

Draw Book object

UML → Object Diagrams → Associations

A solid line indicates an association between two objects.

An example object diagram showing two associations:

Conceptual Class Diagrams (aka OODMs)

> The analysis process for identifying objects and object classes is recognized as one of the most difficult areas of object-oriented development. --Ian Sommerville, in the book *Software Engineering*

Sidebar: Domain Modeling

***Domain modeling* is modeling the problem domain** i.e., to model how things actually work in the real world. Domain modeling is useful in understanding the problem domain, which is essential to the success of a project.

Domain modeling can be done using,

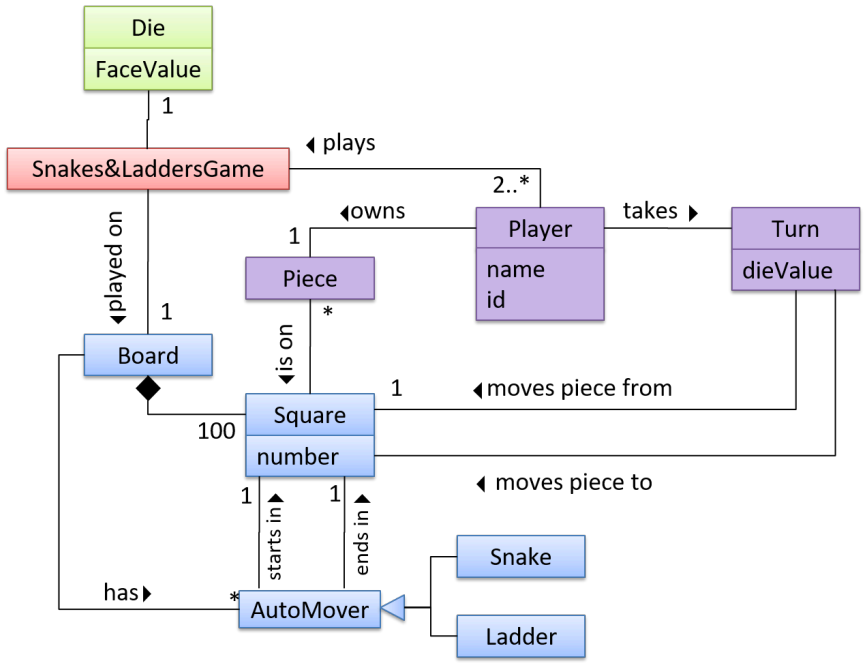
- **a *domain-specific* modeling notation** if such a notation exists (e.g., a modeling notation specific to the banking domain might have elements to represent loans, accounts, transactions etc.),
- **or a general purpose modeling notation**, such as UML (e.g., you can use an activity diagram to model the workflow of processing a loan application),
- **or even other general purpose notations** (e.g., you can use an organization chart to model the employee hierarchy of a company).

When building an OOP system, it makes sense to build OOP models of the problem domain, given OOP aspires to emulate the objects in the real world.

The UML model that captures class structures in the problem domain are called *conceptual class diagrams*. They are in fact a lighter version of class diagrams, and sometimes also called *OO domain models (OODMs)*. The latter name is somewhat misleading as conceptual class diagrams (CCDs) are actually only one type of domain models that can model an OOP problem domain.

The CCD of a snakes and ladders game is given below.

Description: 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.



CCDs do not contain solution-specific classes (i.e., classes that are used in the solution domain but do not exist in the problem domain). For example, a class called DatabaseConnection could appear in a class diagram but not usually in a CCD because DatabaseConnection is something related to a software solution but not an entity in the problem domain.

CCDs represents the class *structure* of the problem domain and not their behavior, just like class diagrams. To show behavior, use other diagrams such as sequence diagrams.

CCD notation is a subset of the class diagram notation (omits methods and navigability).

Modeling behaviors

Activity Diagrams - Basic ⚡⚡⚡

Software projects often involve *workflows*. Workflows define the flow in which a process or a set of tasks is executed. Understanding such workflows is important for the success of the software project.

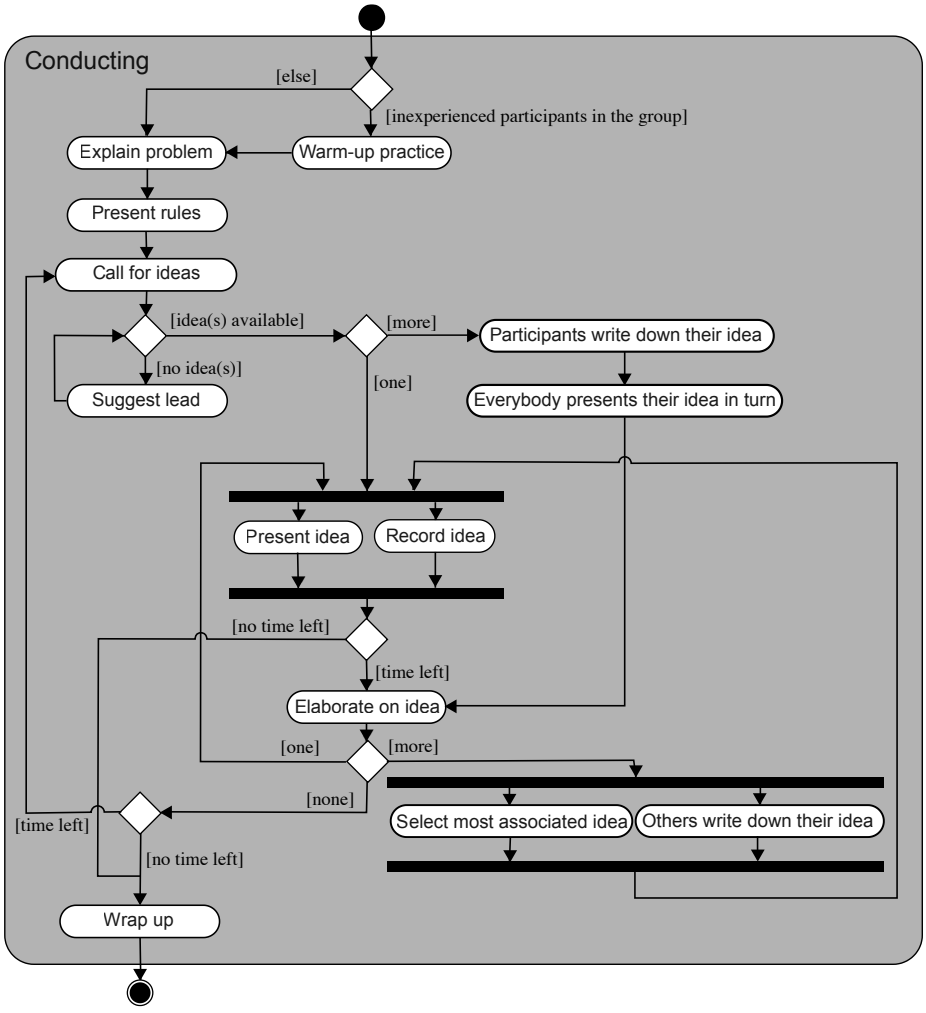
Some examples in which a certain workflow is relevant to software project:

- A software that automates the work of an insurance company needs to take into account the workflow of processing an insurance claim.
- The algorithm of a piece of code represents the workflow (i.e., the execution flow) of the code.

UML Activity Diagrams → Introduction → What

UML *activity diagrams* (AD) can model workflows. *Flow charts* are another type of diagram that can model workflows. Activity diagrams are the UML equivalent of flow charts.

An example activity diagram:

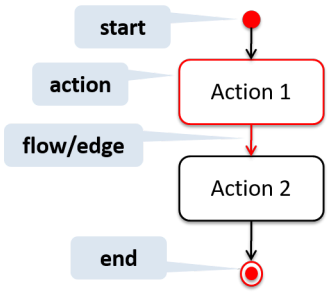


[source:wikipedia]

UML Activity Diagrams → Basic Notation → Linear Paths

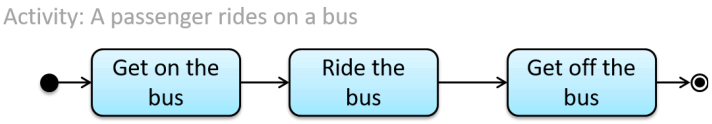
An activity diagram (AD) captures an *activity* through the *actions* and *control flows* that make up the activity.

- An *action* is a single step in an activity. It is shown as a rectangle with rounded corners.
- A *control flow* shows the flow of control from one action to the next. It is shown by drawing a line with an arrow-head to show the direction of the flow.



Note the slight difference between the *start node* and the *end node* which represent the start and the end of the activity, respectively.

This activity diagram shows the action sequence of the activity *a passenger rides the bus*:



Exercises:

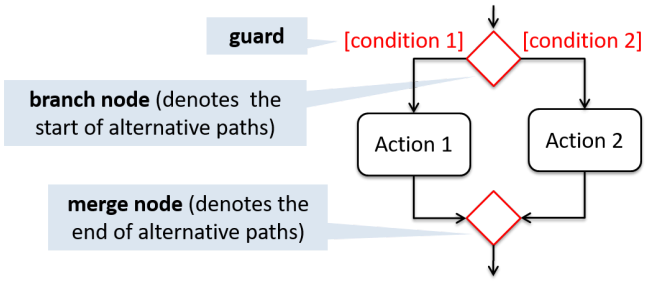
Which activity diagrams are correct?

UML Activity Diagrams → Basic Notation → Alternate Paths

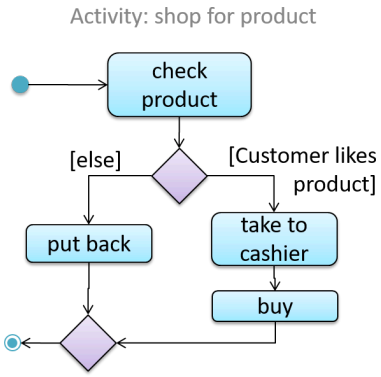
A *branch node* shows the start of alternate paths. Each control flow exiting a branch node has a *guard condition*: a boolean condition that should be true for execution to take that path. ***Exactly one* of the guard conditions should be true** at any given branch node.

A *merge node* shows the end of alternate paths.

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



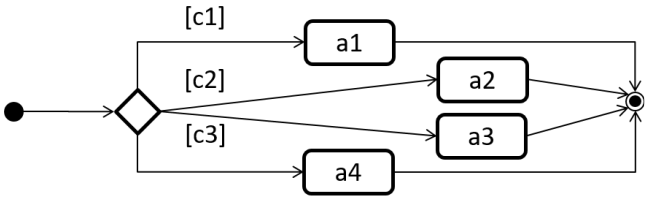
The AD below shows alternate paths involved in the workflow of the activity *shop for product*:



Some acceptable simplifications (by convention):

- Omitting the merge node if it doesn't cause any ambiguities.
- Multiple arrows can start from the same corner of a branch node.
- Omitting the `[Else]` condition.

The AD below illustrates the simplifications mentioned above:



Exercises:

Which activity diagrams are correct?

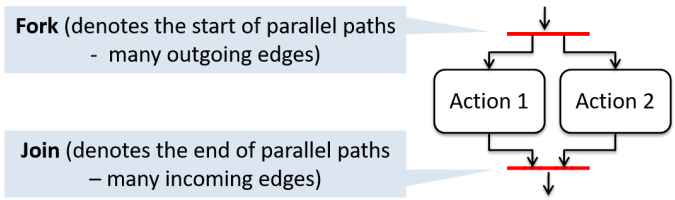
UML Activity Diagrams → Basic Notation → Parallel Paths

***Fork* nodes indicate the start of concurrent flows of control.**

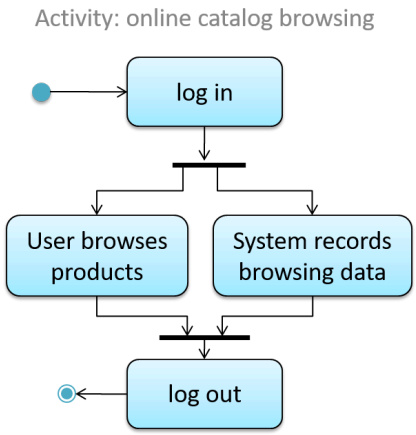
***Join* nodes indicate the end of parallel paths.**

Both have the same notation: a bar.

In a set of parallel paths, execution along **all parallel paths should be complete before the execution can start on the outgoing control flow of the *join*.**



In this activity diagram (from an online shop website) the actions *User browses products* and *System records browsing data* happen in parallel. Both of them need to finish before the *log out* action can take place.



Exercises:

Which activity diagrams are correct?

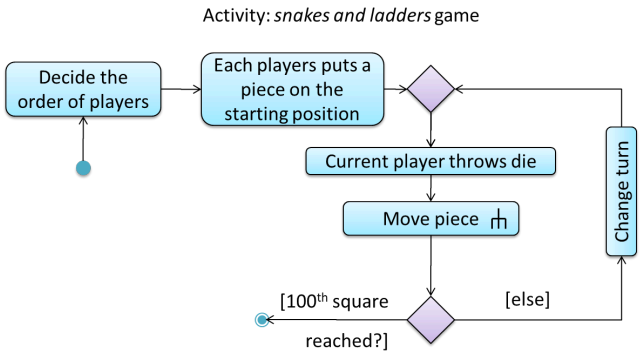
Which sequence of actions are supported?

Activity Diagrams - Intermediate

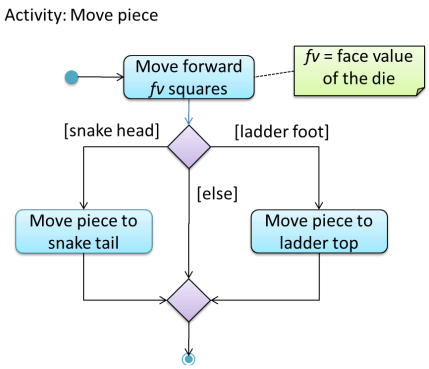
UML Activity Diagrams → Intermediate Notation → Rakes

The rake notation is used to indicate that a part of the activity is given as a separate diagram.

Here is the AD for a game of 'Snakes and Ladders'.



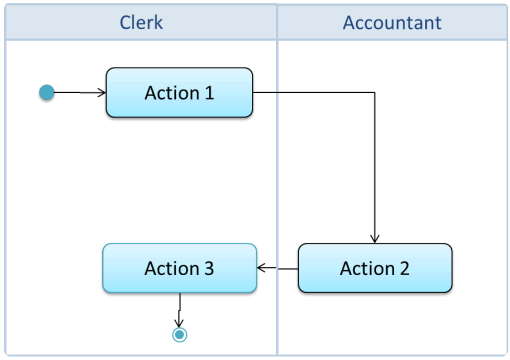
The *rake* symbol (in the `Move piece` action above) is used to show that the action is described in another subsidiary activity diagram elsewhere. That diagram is given below.



UML Activity Diagrams → Intermediate Notation → Swim Lanes

It is possible to *partition* an activity diagram to show who is doing which action. Such partitioned activity diagrams are sometime called *swimlane diagrams*.

A simple example of a swimlane diagram:



Sequence Diagrams - Basic

Sequence diagrams model the interactions between various entities in a system, in a specific scenario. Modelling such scenarios is useful, for example, to verify the design of the internal interactions is able to provide the expected outcomes.

Some examples where a sequence diagram can be used:

To model how components of a system interact with each other to respond to a user action.

To model how objects inside a component interact with each other to respond to a method call it received from another component.

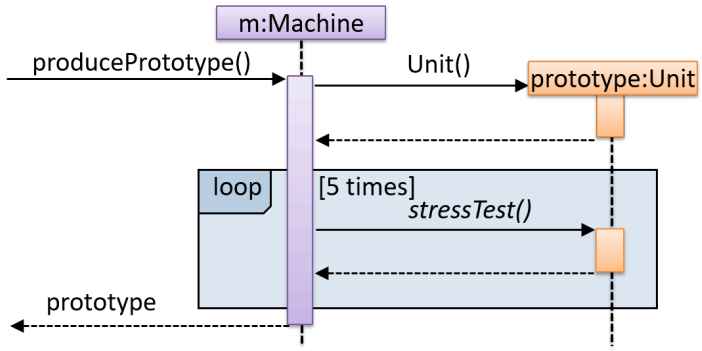
UML Sequence Diagrams → Introduction

A UML sequence diagram *captures the interactions between multiple entities for a given scenario*.

Consider the code below.

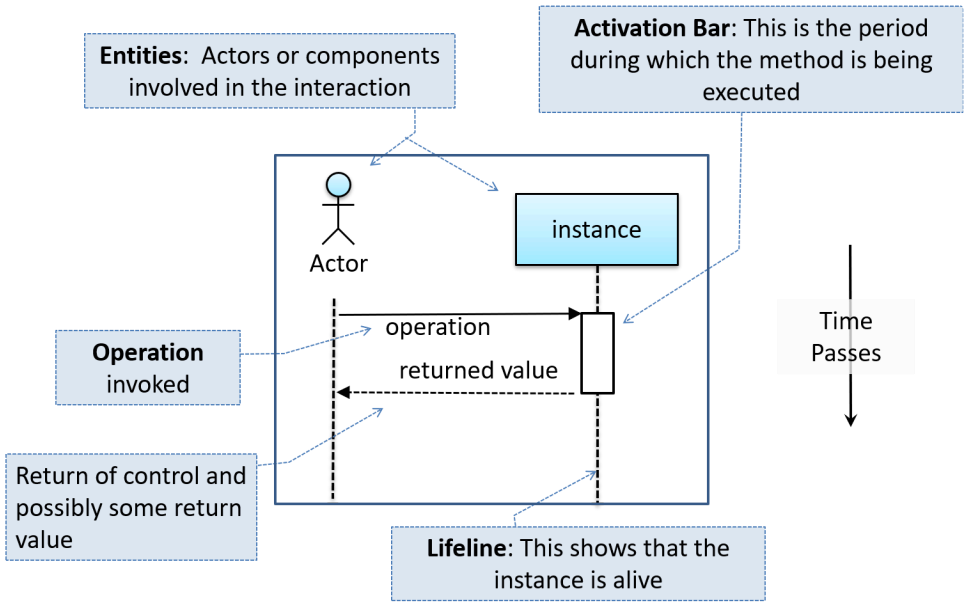
```
class Machine {

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

class Unit {

    public void stressTest() {

    }
}
```

Here is the sequence diagram to model the interactions for the method call `produceProtoype()` on a `Machine` object.

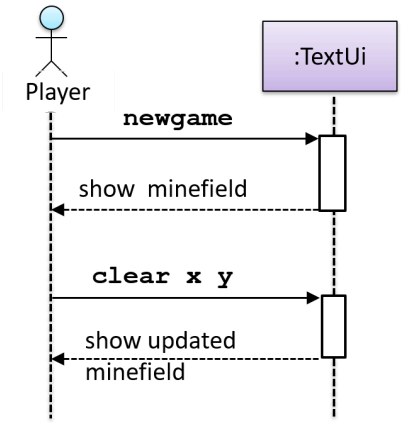


▼ 🎓 UML 🔜 Sequence Diagrams → Basic Notation

Notation:



This sequence diagram shows some interactions between a human user and the Text UI of a CLI Minesweeper game.



The player runs the `newgame` action on the `TextUi` object which results in the `TextUi` showing the minefield to the player. Then, the player runs the `clear x y` command; in response, the `TextUi` object shows the updated minefield.

The `:TextUi` in the above example denotes *an unnamed instance of the class TextUi*. If there were two instances of `TextUi` in the diagram, they can be distinguished by naming them e.g., `TextUi1:TextUi` and `TextUi2:TextUi`.

Arrows representing method calls should be solid arrows while those representing method returns should be dashed arrows.

Note that unlike in object diagrams, the **class/object name is not underlined in sequence diagrams**.

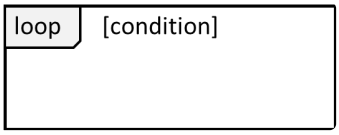
The arrowhead style depends on the type of method call. Arrows showing *synchronous* (i.e., the caller method is blocked from doing anything else until the called method returns) should use *filled* arrowheads (e.g., ➡). Most method calls (e.g., normal Java method calls) are synchronous. *Asynchronous* method calls (i.e., the caller method does not have to wait till the called method returns) are shown using *lined* arrowheads (e.g., →). As the latter is out of scope for this textbook, all sequence diagram arrow heads you encounter in this textbook will be of the first type.

✕ **[Common notation error] Activation bar too long:** The activation bar of a method cannot start before the method call arrives and a method cannot remain active after the method has returned. In the two sequence diagrams below, the one on the left commits this error because the activation bar starts *before* the method `Foo#xyz()` is called and remains active *after* the method returns.

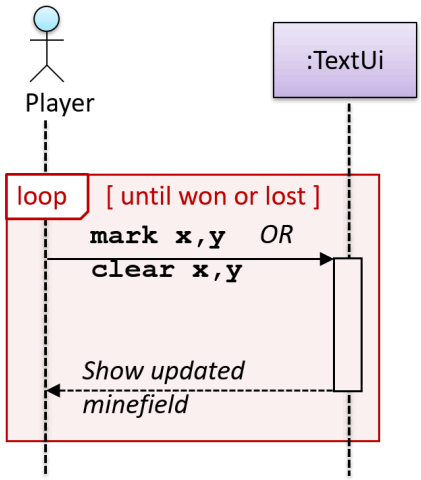
✕ **[Common notation error] Broken activation bar:** The activation bar should remain unbroken while the method is being executed, from the point the method is called until the method returns. In the two sequence diagrams below, the one on the left commits this error because the activation bar for the method `Foo#abc()` is broken in the middle.

UML Sequence Diagrams → Loops

Notation:

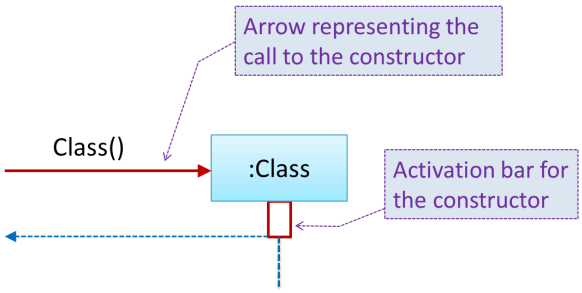


The `Player` calls the `mark x,y` command or `clear x y` command repeatedly until the game is won or lost.



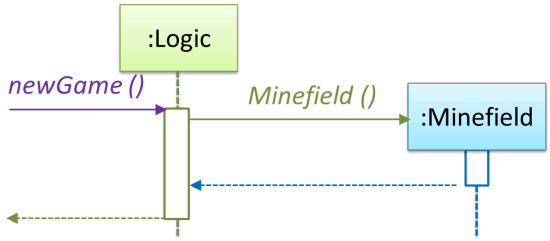
UML Sequence Diagrams → Object Creation

Notation:



- 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.

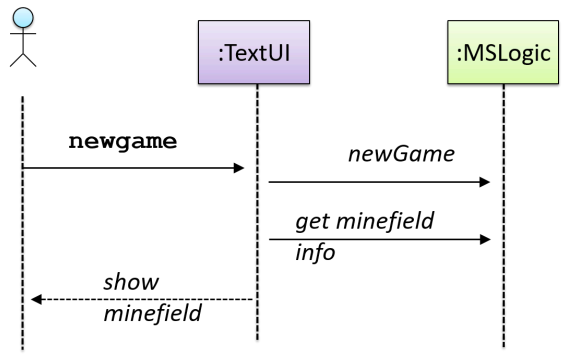
The `Logic` object creates a `Minefield` object.



UML Sequence Diagrams → Minimal Notation

To reduce clutter, **optional elements (e.g, activation bars, return arrows) may be omitted** if the omission does not result in ambiguities or loss of *relevant* information. Informal operation descriptions such as those given in the example below can be used, if more precise details are not required for the task at hand.

A minimal sequence diagram



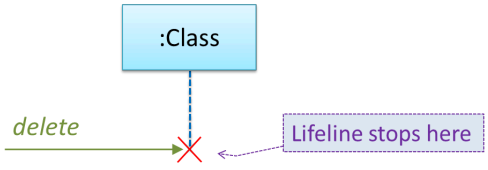
If method parameters don't matter to the purpose of the sequence diagram, you can omit them using `...` e.g., use `foo(...)` instead of `foo(int size, double weight)`.

Sequence Diagrams - Intermediate ⚡⚡⚡

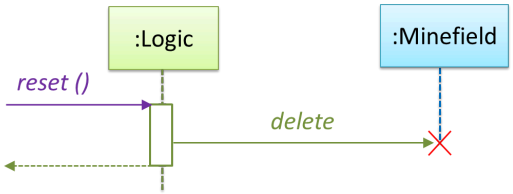
UML Sequence Diagrams → Object Deletion

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

Notation:



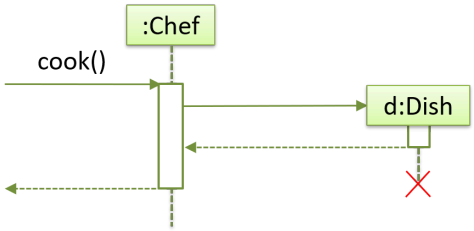
Note how the below diagram shows the deletion of the `Minefield` object.



Although languages such as Java do not support a `delete` operation (because they use automatic memory management), **you can use the object deletion notation to indicate the point at which the object becomes *ready to be garbage-collected*** (i.e., the point at which it ceases to be referenced).

Note how `d` lifeline ends with an `X` to show that it is 'deleted' (i.e., ready to be garbage collected) after the `cook()` method returns.

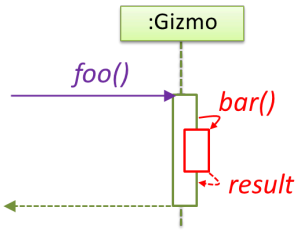
```
class Chef {
    void cook() {
        Dish d = new Dish();
    }
}
```



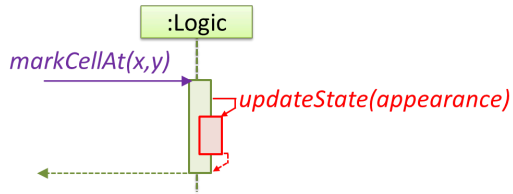
UML Sequence Diagrams → Self-Invocation

UML can show a method of an object calling another of its own methods.

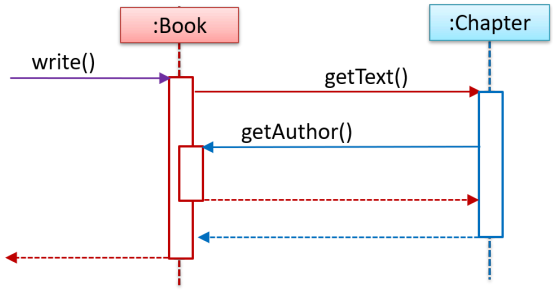
Notation:



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.



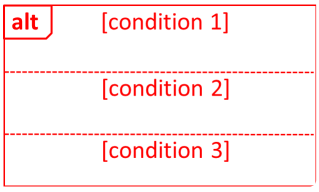
'Unroll' chained/compound method calls before drawing sequence diagram. Consider the Java statement `new Book().add(new Chapter());` . How do we show it as a sequence diagram? First, 'unroll' it into a simpler series of statements, which can then be drawn as a sequence diagram easily. For example, that statement is equivalent to the following:

```
Book b = new Book();
Chapter c = new Chapter();
b.add(c);
```

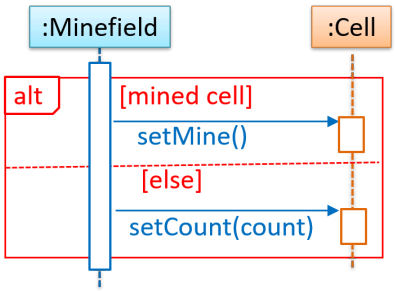
UML Sequence Diagrams → Alternative Paths

UML uses `alt` frames to indicate alternative paths.

Notation:



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

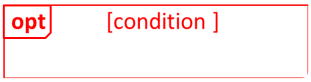


No more than one alternative partitions be executed in an `alt` frame. That is, it is acceptable for none of the alternative partitions to be executed but it is not acceptable for multiple partitions to be executed.

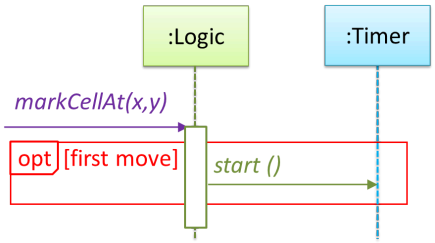
UML Sequence Diagrams → Optional Paths

UML uses `opt` frames to indicate optional paths.

Notation:



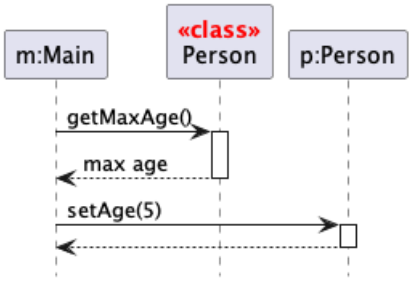
`Logic#markCellAt(...)` calls `Timer#start()` only if it is the first move of the player.



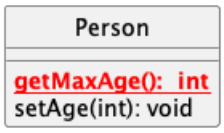
UML Sequence Diagrams → Calls to Static Methods

Method calls to `static` (i.e., class-level) methods are received by the class itself, not an instance of that class. You can use `<<class>>` to show that a participant is the class itself.

In this example, `m` calls the static method `Person.getMaxAge()` and also the `setAge()` method of a `Person` object `p`.



Here is the `Person` class, for reference:

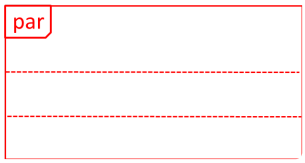


Sequence Diagrams - Advanced ⚡⚡⚡

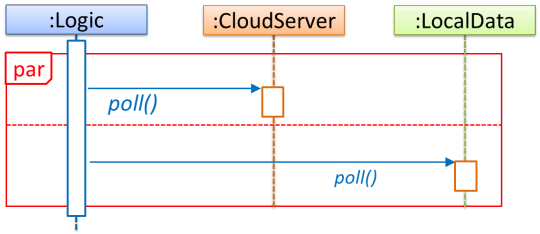
UML: Sequence Diagrams: Parallel Paths

UML uses `par` frames to indicate parallel paths.

Notation:



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

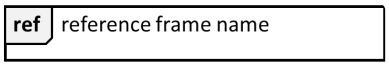


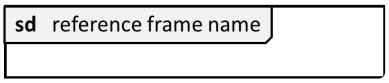
If you show parallel paths in a sequence diagram, the corresponding Java implementation is likely to be *multi-threaded* because a normal Java program cannot do multiple things at the same time.

UML: Sequence Diagrams: Reference Frames

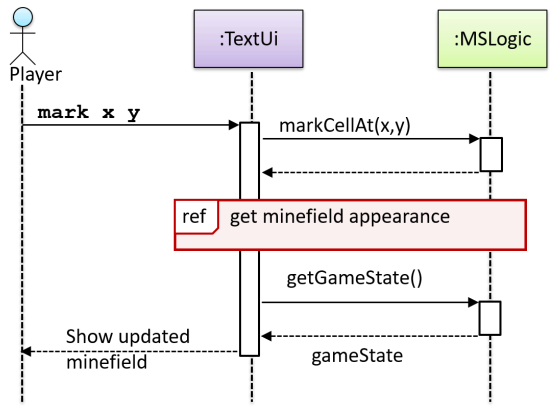
UML uses *ref frame* to allow a segment of the interaction to be omitted and shown as a separate sequence diagram. Reference frames help you to break complicated sequence diagrams into multiple parts or simply to omit details you are not interested in showing.

Notation:

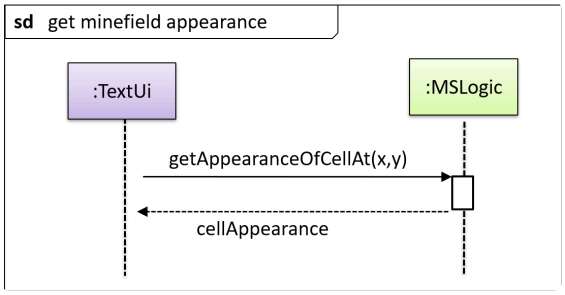




The details of the `get minefield appearance` interactions have been omitted from the diagram.



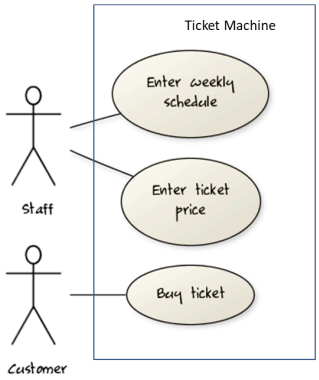
Those details are shown in a separate sequence diagram given below.



Use Case Diagrams

Use case diagrams model the mapping between *features* of a system and its user *roles* i.e., which user roles can perform which tasks using the software.

A simple use case diagram:

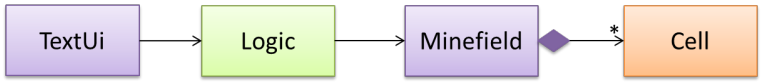


Modeling a solution

Introduction

You can use models to analyze and design software before you start coding.

Suppose you are planning to implement a simple minesweeper game that has a text based UI and a GUI. Given below is a possible OOP design for the game.



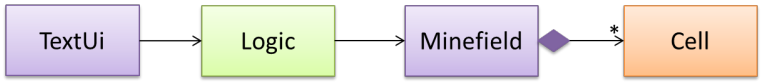
Before jumping into coding, you may want to find out things such as,

- Is this class structure able to produce the behavior you want?
- What API should each class have?
- Do you need more classes?

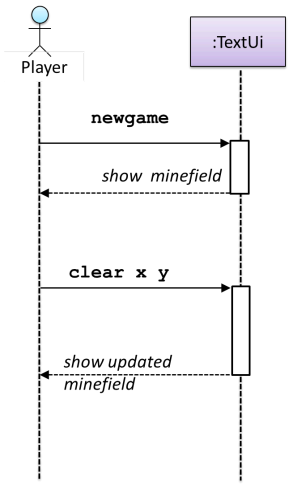
To answer these questions, you can analyze how the objects of these classes will interact with each other to produce the behavior you want.

Basic

As mentioned in [Design → Modeling → Modeling a Solution → Introduction], this is the Minesweeper design you have come up with so far. Our objective is to analyze, evaluate, and refine that design.

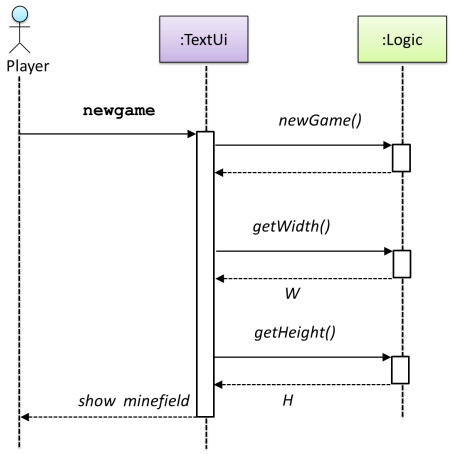


Let us start by modeling a sample interaction between the person playing the game and the `TextUi` object.



`newgame` and `clear x y` represent commands typed by the `Player` on the `TextUi` .

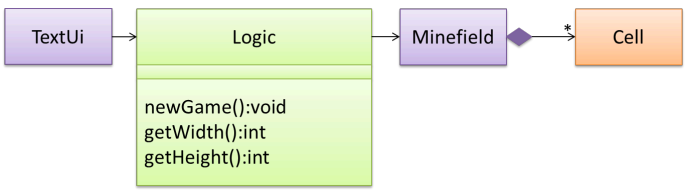
How does the `TextUi` object carry out the requests it has received from the player? It would need to interact with other objects of the system. Because the `Logic` class is the one that controls the game logic, the `TextUi` needs to collaborate with `Logic` to fulfill the `newgame` request. Let us extend the model to capture that interaction.



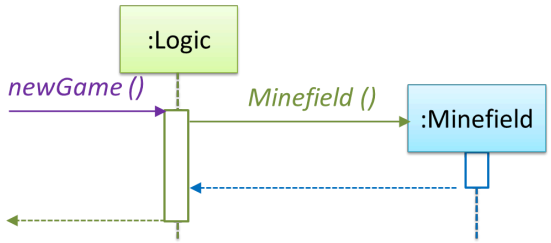
`W` = Width of the minefield; `H` = Height of the minefield

The above diagram assumes that `W` and `H` are the only information `TextUi` requires to display the minefield to the `Player` . Note that there could be other ways of doing this.

The `Logic` methods you conceptualized in our modeling so far are:

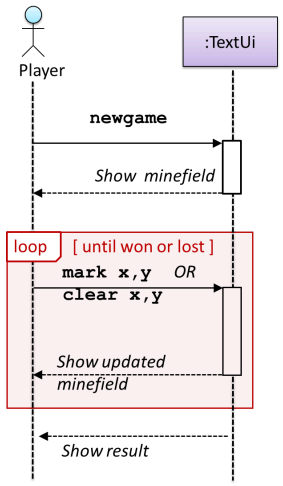


Now, let us look at what other objects and interactions are needed to support the `newGame()` operation. It is likely that a new `Minefield` object is created when the `newGame()` method is called.



Note that the behavior of the `Minefield` constructor has been abstracted away. It can be designed at a later stage.

Given below are the interactions between the player and the `TextUi` for the whole game.



Note that a similar technique can be used when discovering/defining the architecture-level APIs.

Software architecture

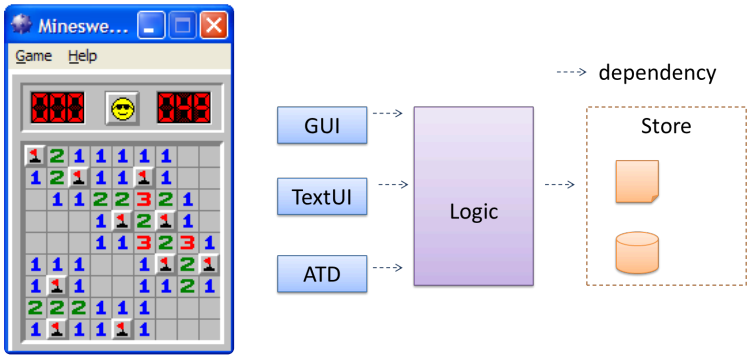
Introduction

What

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural. -- *Software Architecture in Practice (2nd edition)*, Bass, Clements, and Kazman

The software architecture shows the overall organization of the system and can be viewed as a very high-level design. It usually consists of a set of interacting components that fit together to achieve the required functionality. It should be a simple and technically viable structure that is well-understood and agreed-upon by everyone in the development team, and it forms the basis for the implementation.

A possible architecture for a *Minesweeper* game:



Main components:

- `GUI` : Graphical user interface
- `TextUi` : Textual user interface
- `ATD` : An automated test driver used for testing the game logic
- `Logic` : Computation and logic of the game
- `Store` : Storage and retrieval of game data (high scores etc.)

The architecture is typically designed by the *software architect*, who provides the technical vision of the system and makes high-level (i.e., architecture-level) technical decisions about the project.

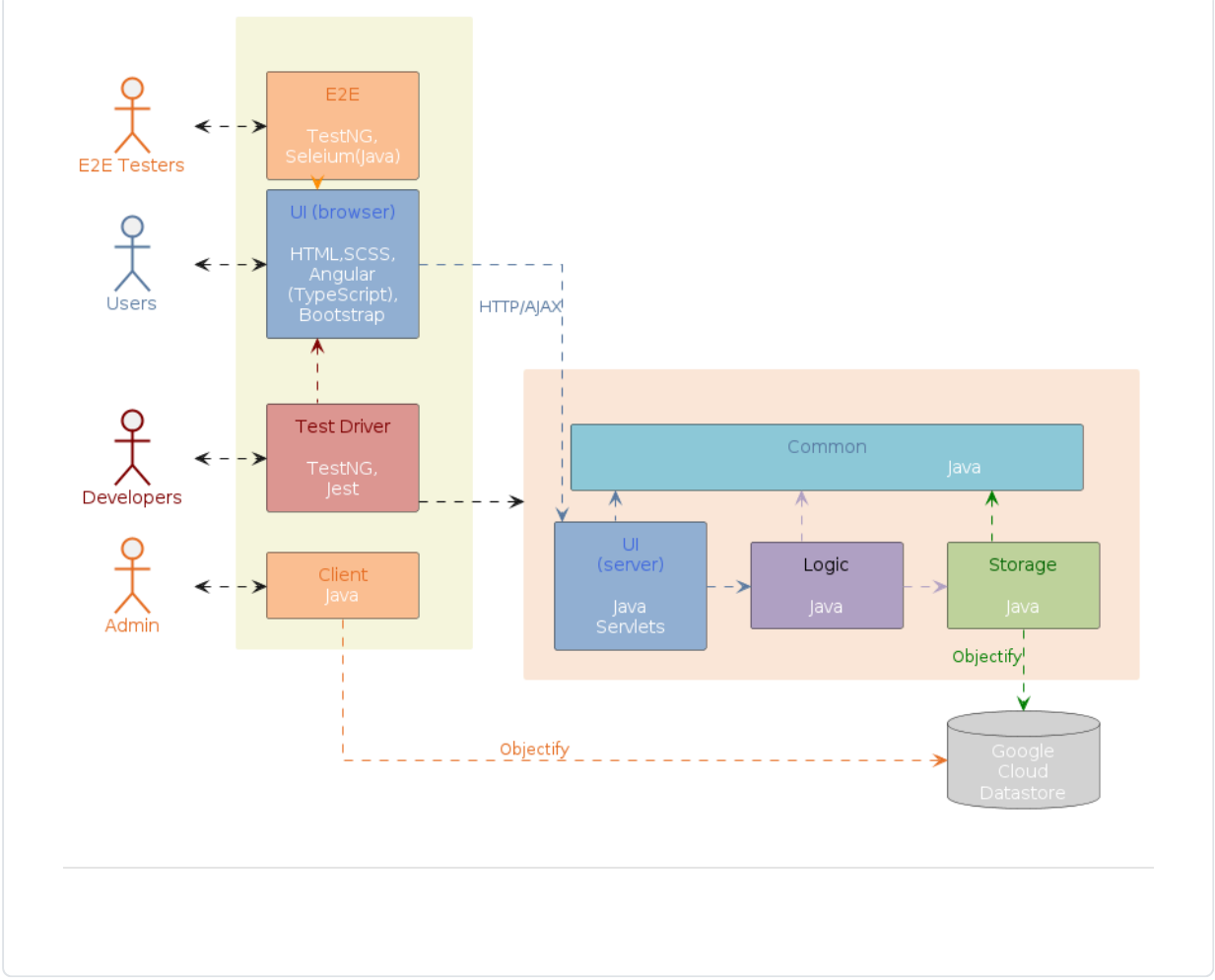
Architecture diagrams

Reading

Architecture diagrams are free-form diagrams. There is no universally adopted standard notation for architecture diagrams. Any symbols that reasonably describe the architecture may be used.

Some example architecture diagrams:

TEAMMATES

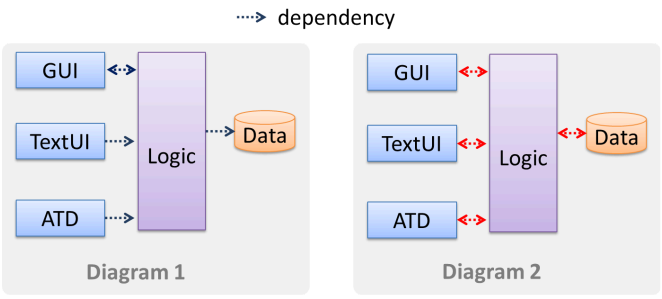


Drawing

While architecture diagrams have no standard notation, try to follow these basic guidelines when drawing them.

- Minimize the variety of symbols. If the symbols you choose do not have widely-understood meanings e.g., A drum symbol is widely-understood as representing a database, explain their 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

Introduction

What

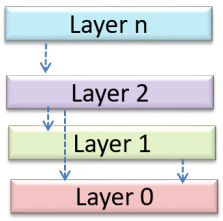
Software architectures follow various high-level styles (aka *architectural patterns*), just like how building architectures follow various architecture styles.

n-tier style, client-server style, event-driven style, transaction processing style, service-oriented style, pipes-and-filters style, message-driven style, broker style, ...

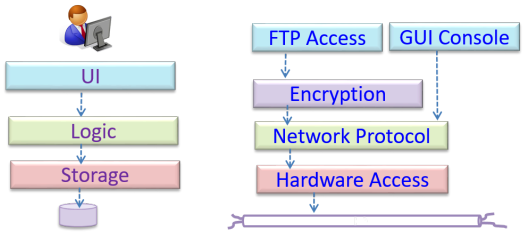
N-tier architectural style

What ⚡⚡

In the *n-tier* style, higher layers make use of services provided by lower layers. Lower layers are independent of higher layers. Other names: *multi-layered*, *layered*.



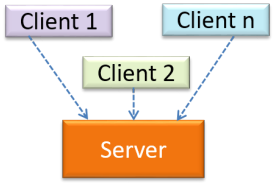
Operating systems and network communication software often use n-tier style.



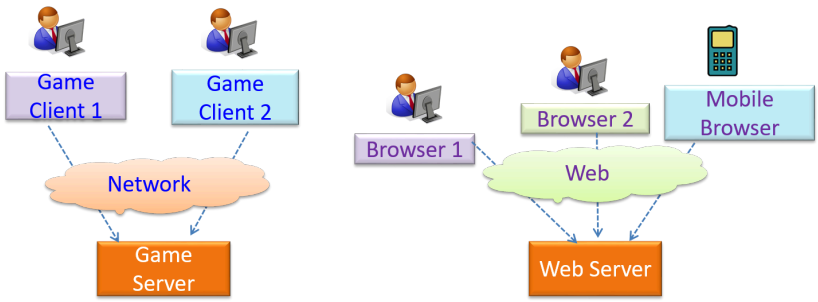
Client-server architectural style

What ⚡⚡⚡

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. This is an architectural style used often in distributed applications.



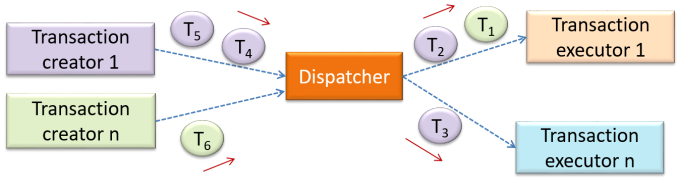
The online game and the web application below use the client-server style.



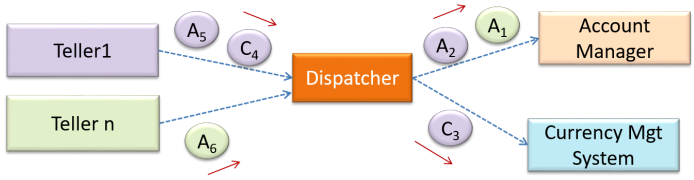
Transaction processing architectural style

What ⚡⚡⚡

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.



In this example from a banking system, transactions are generated by the terminals used by tellers, which are then sent to a central dispatching unit, which in turn dispatches the transactions to various other units to execute.

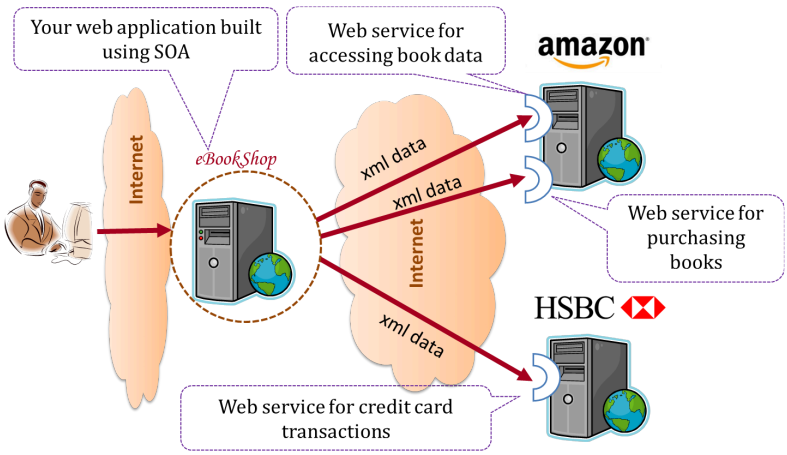


Service-oriented architectural style

What ϟϟϟ

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.

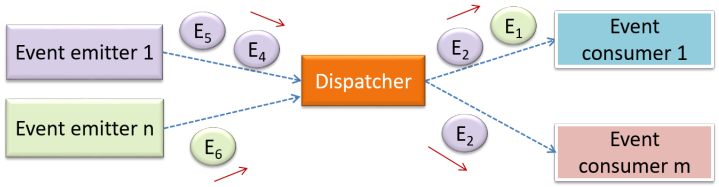
Suppose that Amazon.com provides a web service for customers to browse and buy merchandise, while HSBC provides a web service for merchants to charge HSBC credit cards. Using these web services, an 'eBookShop' web application can be developed that allows HSBC customers to buy merchandise from Amazon and pay for them using HSBC credit cards. Because both Amazon and HSBC services follow the SOA architecture, their web services can be reused by the web application, even if all three systems use different programming platforms.



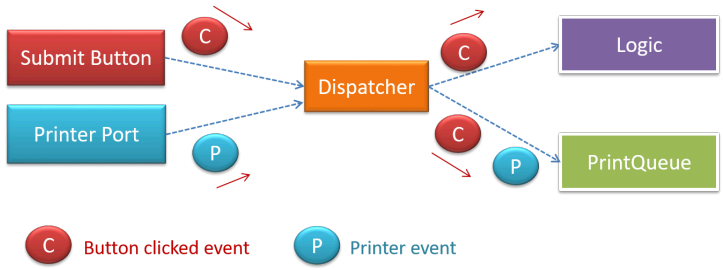
Event-driven architectural style

What ϟϟ

***Event-driven* style controls the flow of the application by detecting events from event *emitters* and communicating those events to interested event *consumers*.** This architectural style is often used in GUIs.



When the 'button clicked' event occurs in a GUI, that event can be transmitted to components that are interested in reacting to that event. Similarly, events detected at a printer port can be transmitted to components related to operating the printer. The same event can be sent to multiple consumers too.



More

Using Styles ϟϟϟ

Most applications use a mix of these architectural styles.

An application can use a client-server architecture where the server component comprises several layers, i.e., it uses the n-tier architecture.

Software design patterns

Introduction

What ⚡⚡

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

In software development, there are certain problems that recur in a certain context.

📦 Some examples of recurring design problems:

Design Context	Recurring Problem
Assembling a system that makes use of other existing systems implemented using different technologies	What is the best architecture?
UI needs to be updated when the data in the application backend changes	How to initiate an update to the UI when data changes without coupling the backend to the UI?

After repeated attempts at solving such problems, better solutions are discovered and refined over time. These solutions are known as design patterns, **a term popularized by the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by the so-called "Gang of Four" (GoF)** written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Format ⚡⚡

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 pattern

What ⚡⚡

Context

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

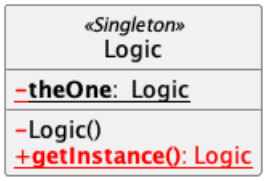
Problem

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*.

📦 Example:



The `<<Singleton>>` in the class above uses the UML *stereotype* notation, which is used to (optionally) indicate the purpose or the role played by a UML element. In this example, the class `Logic` is playing the role of a Singleton class. The general format is `<<role/purpose>>` .

Implementation ⚡⚡

Here is the typical implementation of how the Singleton pattern is applied to a class:

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

    private Logic() {
        ...
    }

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

Notes:

- The constructor is `private` , which prevents instantiation from outside the class.
- The single instance of the singleton class is maintained by a `private` class-level variable.
- Access to this object is provided by a `public` class-level operation `getInstance()` which instantiates a single copy of the singleton class when it is executed for the first time. Subsequent calls to this operation return the single instance of the class.

If `Logic` was not a Singleton class, a `Logic` object can be created as follows:

```
Logic m = new Logic();
```

But when it is a Singleton class, the single `Logic` object needs to be accessed as follows:

```
Logic m = Logic.getInstance();
```

Evaluation ⚡⚡⚡

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 codebase

Cons:

- The singleton object acts like a global variable that increases coupling across the codebase.
- 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 you want each test to be independent of the others.

Given that there are some significant cons, it is recommended that you apply the Singleton pattern when, in addition to requiring only one instance of a class, there is a risk of creating multiple objects by mistake, and creating such multiple objects has real negative consequences.

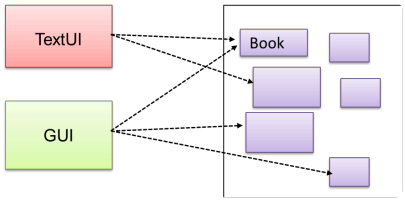
Facade pattern

What ⚡⚡⚡

Context

Components need to access functionality deep inside other components.

The `UI` component of a `Library` system might want to access functionality of the `Book` class contained inside the `Logic` component.



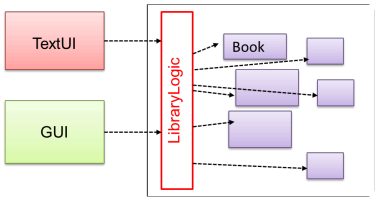
Problem

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 Facade class that sits between the component internals and users of the component such that all access to the component happens through the Facade class.

The following class diagram applies the Facade pattern to the `Library System` example. The `LibraryLogic` class is the Facade class.



Command pattern

What ⚡⚡⚡

Context

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

Problem

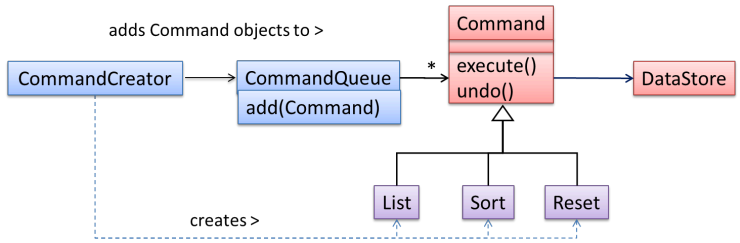
It is preferable that some part of the code executes these commands without having to know each command type. e.g., there can be a `CommandQueue` object that is responsible for queuing commands and executing them without knowledge of what each command does.

Solution

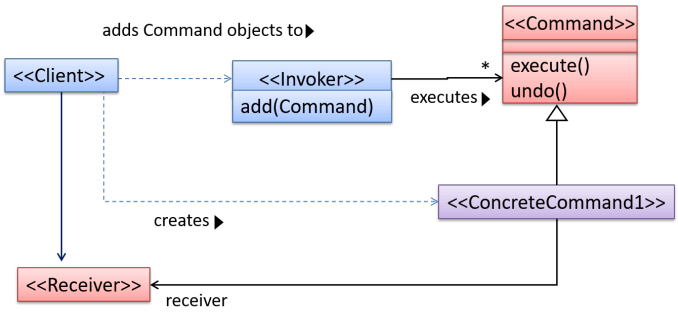
The essential element of this pattern is to have a general `<<Command>>` object that can be passed around, stored, executed, etc without knowing the type of command (i.e., via polymorphism).

Let us examine an example application of the pattern first:

In the example solution below, the `CommandCreator` creates `List`, `Sort`, and `Reset Command` objects and adds them to the `CommandQueue` object. The `CommandQueue` object treats them all as `Command` objects and performs the execute/undo operation on each of them without knowledge of the specific `Command` type. When executed, each `Command` object will access the `DataStore` object to carry out its task. The `Command` class can also be an abstract class or an interface.



The general form of the solution is as follows.



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) pattern

What ⚡⚡

Context

Most applications support storage/retrieval of information, displaying of information to the user (often via multiple UIs having different formats), and changing stored information based on external inputs.

Problem

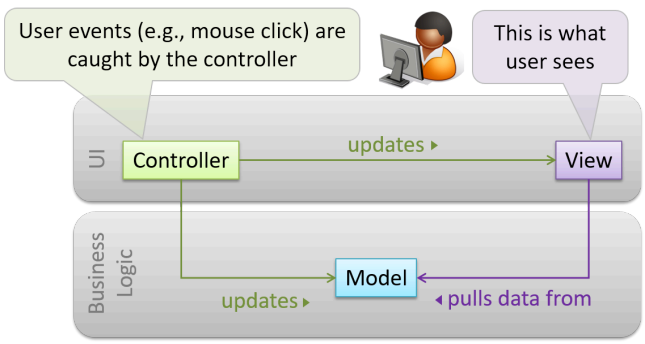
The high coupling that can result from the interlinked nature of the features described above.

Solution

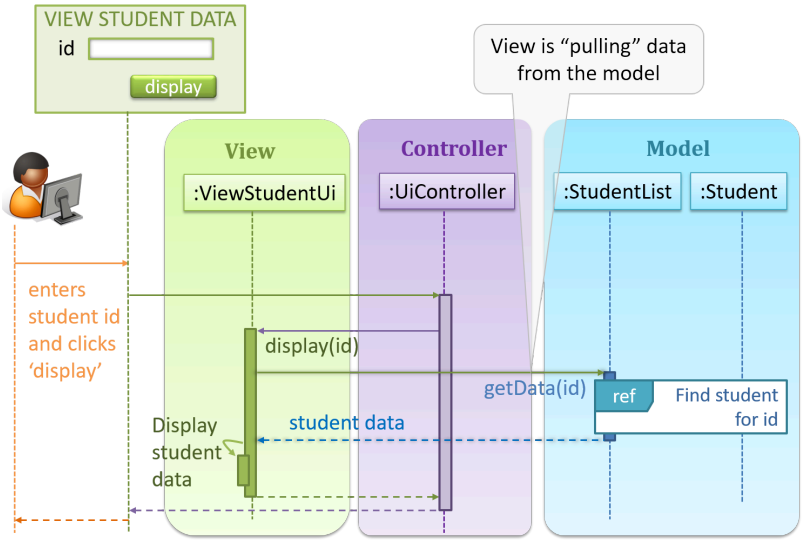
Decouple data, presentation, and control logic of an application by separating them into three different components: *Model*, *View* and *Controller*.

- *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.

The relationship between the components can be observed in the diagram below. Typically, the UI is the combination of *View* and *Controller*.



Given below is a concrete example of MVC applied to a student management system. In this scenario, the user is retrieving the data of a student.



In the diagram above, when the user clicks on a button using the UI, the ‘click’ event is caught and handled by the `UiController`. The `ref` frame indicates that the interactions within that frame have been extracted out to another separate sequence diagram.

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

There are many variations of the MVC model used in different domains. For example, the one used in a desktop GUI could be different from the one used in a web application.

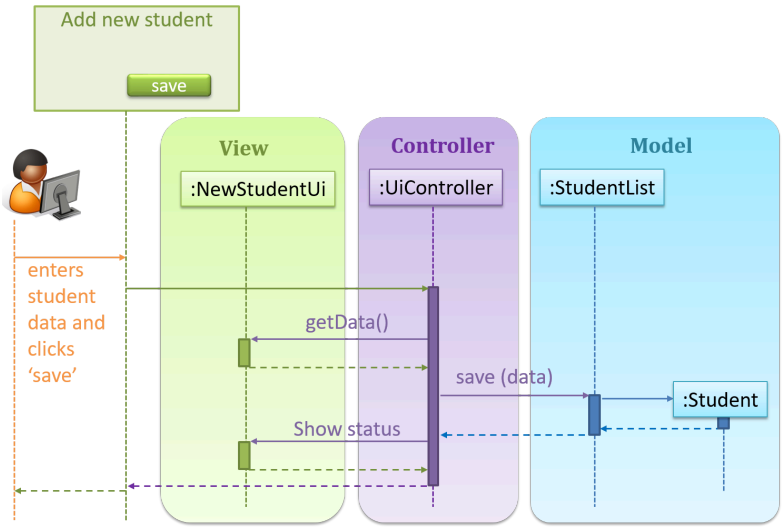
Observer pattern

What ⚡⚡⚡

Context

An object (possibly more than one) is interested in being notified when a change happens to another object. That is, some objects want to ‘observe’ another object.

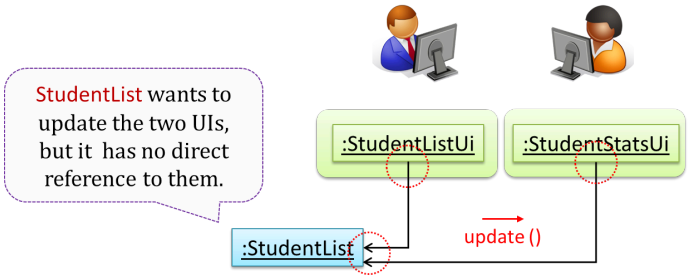
Consider this scenario from a student management system where the user is adding a new student to the system.



Now, assume the system has two additional views used in parallel by different users:

- `StudentListUi` : that accesses a list of students and
- `StudentStatsUi` : that generates statistics of current students.

When a student is added to the database using `NewStudentUi` shown above, both `StudentListUi` and `StudentStatsUi` should get updated automatically, as shown below.



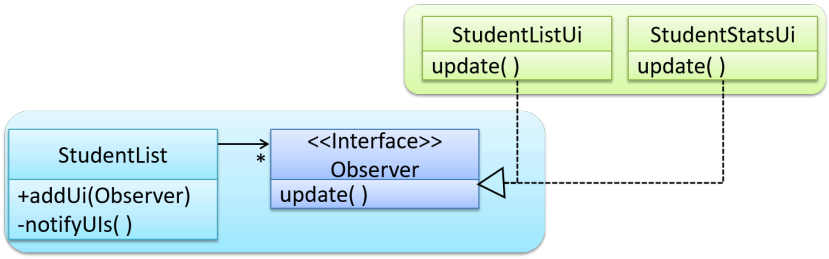
However, the `StudentList` object has no knowledge about `StudentListUi` and `StudentStatsUi` (note the direction of the navigability) and has no way to inform those objects. This is an example of the type of problem addressed by the Observer pattern.

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.



Here is the Observer pattern applied to the student management system.

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.

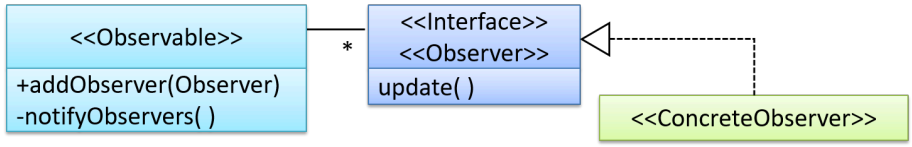
```
// StudentList class
public void notifyUIs() {
    // for each observer in the list
    for (Observer o: observerList) {
        o.update();
    }
}
```

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.

Here is the generic description of the observer pattern:



- `<<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.

In a GUI application, how is the Controller notified when the "save" button is clicked? UI frameworks such as JavaFX have inbuilt support for the Observer pattern.

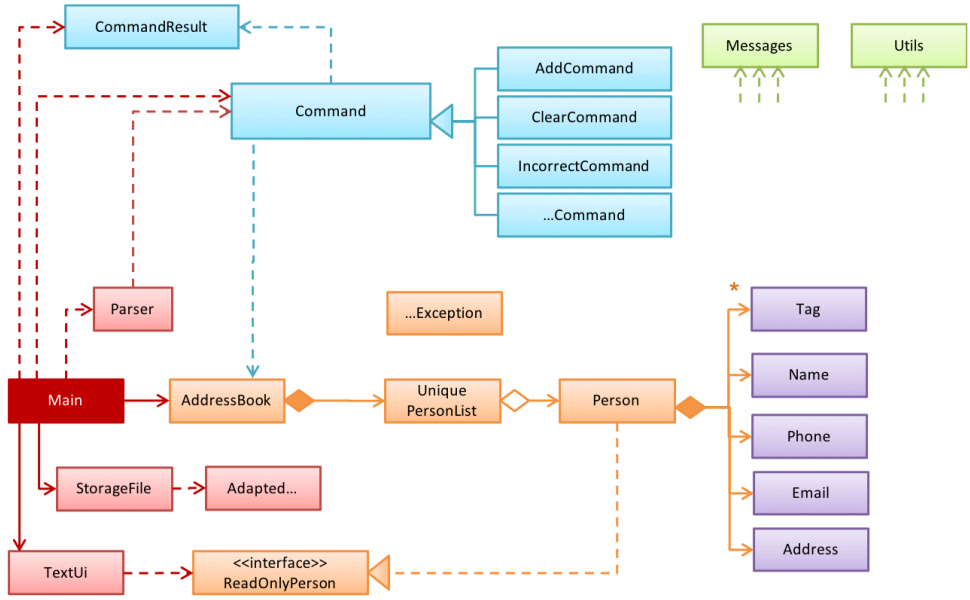
Design approaches

Multi-level design

What ⚡⚡

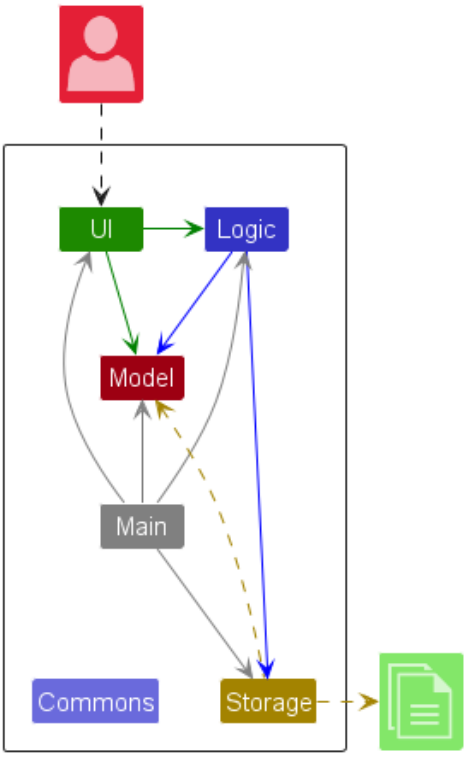
In a smaller system, the design of the entire system can be shown in one place.

This class diagram of se-edu/addressbook-level2 depicts the design of the entire software.

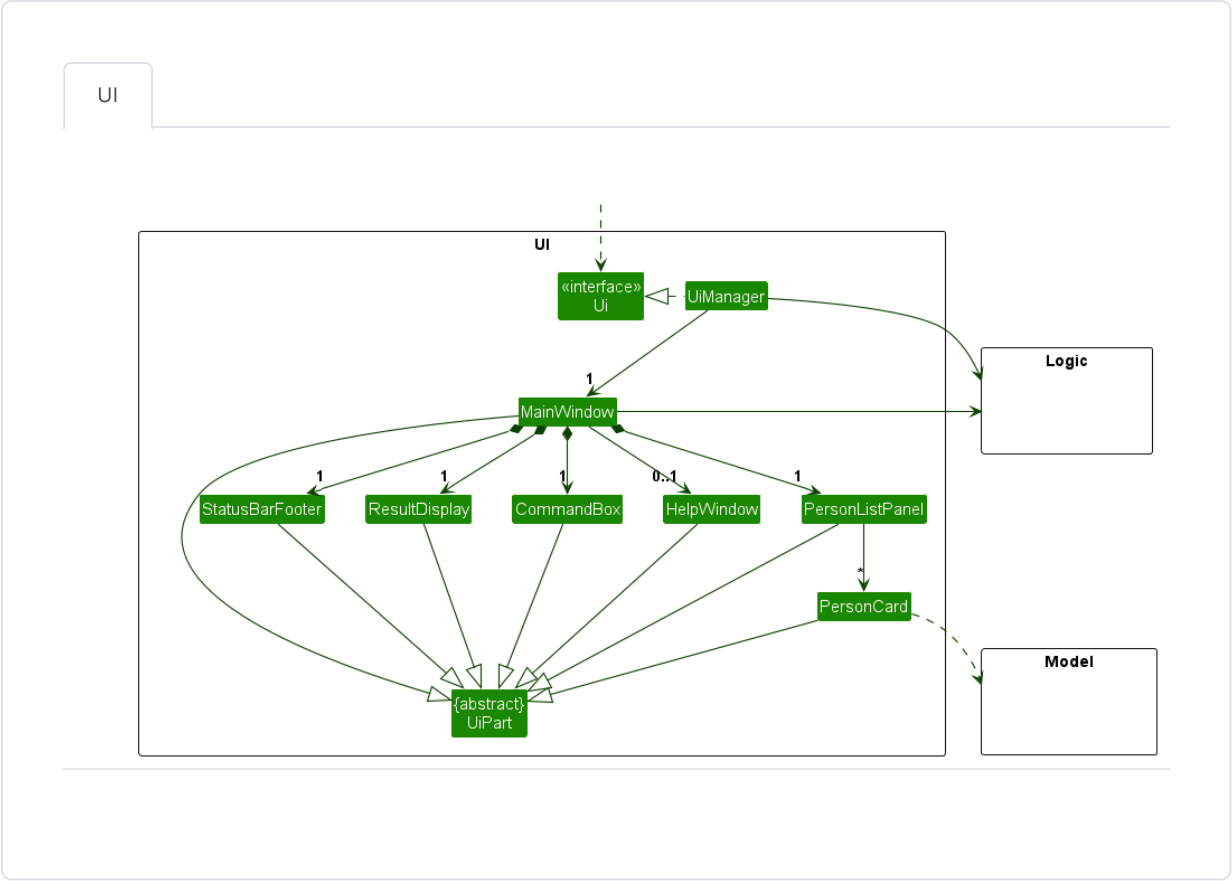


The design of bigger systems needs to be done/shown at multiple levels.

This architecture diagram of se-edu/addressbook-level3 depicts the high-level design of the software.



Here are examples of lower level designs of some components of the same software:



Top-down and bottom-up design

What ⚡⚡⚡

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. 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 design

What ⚡⚡⚡

***Agile design* can be contrasted with *full upfront design* in the following way**:

> **Agile designs are emergent, they're not defined up front.** Your overall system design will emerge over time, evolving to fulfill new requirements and take advantage of new technologies as appropriate. Although you will often do **some initial architectural modeling at the very beginning** of a project, this will be just enough to get your team going. This approach does not produce a fully documented set of models in place before you may begin coding. -- adapted from agilemodeling.com

SECTION: IMPLEMENTATION

IDEs

Introduction

What ⚡

Professional software engineers often write code using *Integrated Development Environments (IDEs)*. IDEs support most development-related work within the same tool (hence, the term *integrated*).

An IDE generally consists of:

- A source code editor that includes features such as syntax coloring, auto-completion, easy code navigation, error highlighting, and code-snippet generation.
- A compiler and/or an interpreter (together with other build automation support) that facilitates the compilation/linking/running/deployment of a program.
- A debugger that allows the developer to execute the program one step at a time to observe the run-time behavior in order to locate bugs.
- Other tools that aid various aspects of coding e.g., support for automated testing, drag-and-drop construction of UI components, version management support, simulation of the target runtime platform, modeling support, AI-assisted coding help, collaborative coding with others.

Examples of popular IDEs:

- Java: Eclipse, IntelliJ IDEA, NetBeans
- C#, C++: Visual Studio
- Swift: XCode
- Python: PyCharm
- Multiple languages: VS Code

Some web-based IDEs have appeared in recent times too e.g., Amazon's Cloud9 IDE.

Some experienced developers, in particular those with a UNIX background, prefer lightweight yet powerful text editors with scripting capabilities (e.g., Emacs) over heavier IDEs.

Debugging

What ⚡⚡⚡

***Debugging* is the process of discovering defects in the program.** Here are some approaches to debugging:

- 👎 **Bad** -- **By inserting temporary print statements**: This is an ad-hoc approach in which print statements are inserted in the program to print information relevant to debugging, such as variable values. e.g., `Exiting process() method, x is 5.347` . This approach is not recommended due to these reasons:
  - Incurs extra effort when inserting and removing the print statements.
  - These extraneous program modifications increase the risk of introducing errors into the program.
  - These print statements, if not removed promptly after the debugging, may even appear unexpectedly in the production version.
- 👎 **Bad** -- **By manually tracing through the code**: Otherwise known as 'eye-balling', this approach doesn't have the cons of the previous approach, but it too is not recommended (other than as a 'quick try') due to these reasons:
  - It is a difficult, time consuming, and error-prone technique.
  - If you didn't spot the error while writing the code, you might not spot the error when reading the code either.
- 👍 **Good** -- **Using a debugger**: A debugger tool allows you to pause the execution, then step through the code one statement at a time while examining the internal state if necessary. Most IDEs come with an inbuilt debugger. **This is the recommended approach for debugging**.

Code quality

Introduction

Basic ⚡⚡⚡

> ❝ Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live. ❞ -- Martin Golding

Production code needs to be of high quality. Given how the world is becoming increasingly dependent on software, poor quality code is something no one can afford to tolerate.

Guideline: Maximize readability

Introduction ⚡

> ❝ Programs should be written and polished until they acquire publication quality. ❞ --Niklaus Wirth

Among various dimensions of code quality, such as run-time efficiency, security, and robustness, one of the most important is readability (aka understandability). This is because in any non-trivial software project, code needs to be read, understood, and modified by other developers later on. Even if you do not intend to pass the code to someone else, code quality is still important because you will become a 'stranger' to your own code someday.

Basic

Avoid Long Methods ⚡⚡

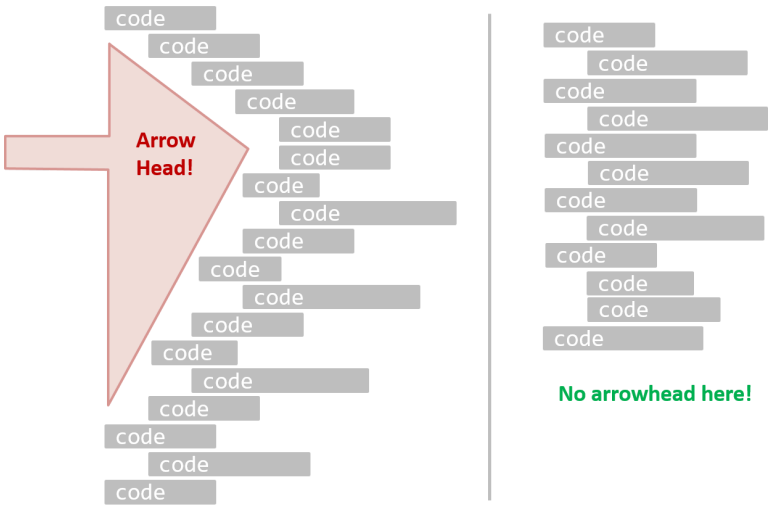
Avoid long methods as they often contain more information than what the reader can process at a time. Consider if shortening is possible when a method goes beyond 30 LoC. The bigger the haystack, the harder it is to find a needle.

Avoid Deep Nesting ⚡⚡

> If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program. --Linux 1.3.53 Coding Style

Avoid deep nesting -- the deeper the nesting, the harder it is for the reader to keep track of the logic.

In particular, avoid *arrowhead* style code.



A real code example:

Bad

```
int subsidy() {
    int subsidy;
    if (!age) {
        if (!sub) {
            if (!notFullTime) {
                subsidy = 500;
```

```
            } else {
                subsidy = 250;
            }
        } else {
            subsidy = 250;
        }
    } else {
        subsidy = -1;
    }
    return subsidy;
}
```

Good

```
int calculateSubsidy() {
    int subsidy;
    if (isSenior) {
        subsidy = REJECT_SENIOR;
    } else if (isAlreadySubsidized) {
        subsidy = SUBSIDIZED_SUBSIDY;
    } else if (isPartTime) {
        subsidy = FULLTIME_SUBSIDY * RATIO;
    } else {
        subsidy = FULLTIME_SUBSIDY;
    }
    return subsidy;
}
```

Avoid Complicated Expressions

Avoid complicated expressions, especially those having many negations and nested parentheses. If you must evaluate complicated expressions, have it done in steps (i.e., calculate some intermediate values first and use them to calculate the final value).

Bad

```
return ((length < MAX_LENGTH) || (previousSize != length))
        && (typeCode == URGENT);
```

Good

```
boolean isWithinSizeLimit = length < MAX_LENGTH;
boolean isSameSize = previousSize != length;
boolean isValidCode = isWithinSizeLimit || isSameSize;

boolean isUrgent = typeCode == URGENT;

return isValidCode && isUrgent;
```

> The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. -- Edsger Dijkstra

Avoid Magic Numbers

Avoid *magic numbers* in your code. When the code has a number that does not explain the meaning of the number, it is called a "magic number" (as in "the number appears as if by magic"). Using a *named constant* makes the code easier to understand because the name tells us more about the meaning of the number.

👎 Bad

```
return 3.14236;
...
return 9;
```

👍 Good

```
static final double PI = 3.14236;
static final int MAX_SIZE = 10;
...
return PI;
...
return MAX_SIZE - 1;
```

Similarly, you can have 'magic' values of other data types.

👎 Bad

```
return "Error 1432"; // A magic string!
```

Avoid any *magic literals* in general, not just magic numbers.

Make the Code Obvious ⚡⚡

Make the code as explicit as possible, even if the language syntax allows them to be implicit. Here are some examples:

- [`Java`] Use explicit type conversion instead of implicit type conversion.
- [`Java` , `Python`] Use parentheses/braces to show groupings even when they can be skipped.
- [`Java` , `Python`] Use enumerations when a certain variable can take only a small number of finite values. For example, instead of declaring the variable 'state' as an integer and using values 0, 1, 2 to denote the states 'starting', 'enabled', and 'disabled' respectively, declare 'state' as type `SystemState` and define an enumeration `SystemState` that has values `'STARTING'` , `'ENABLED'` , and `'DISABLED'` .

Intermediate

Structure Code Logically ⚡⚡⚡

Lay out the code so that it adheres to the logical structure. The code should read like a story. Just like how you use section breaks, chapters and paragraphs to organize a story, use classes, methods, indentation and line spacing in your code to group related segments of the code. For example, you can use blank lines to separate groups of related statements.

Sometimes, the correctness of your code does not depend on the order in which you perform certain intermediary steps. Nevertheless, this order may affect the clarity of the story you are trying to tell. Choose the order that makes the story most readable.

👎 Bad

```
statement A1
statement A2
statement A3
statement B1
statement C1
statement B2
statement C2
```

👍 Good

```
statement A1
statement A2
statement A3

statement B1
statement B2

statement C1
statement C2
```

Do Not 'Trip Up' Reader ⚡⚡⚡

Avoid things that would make the reader go 'huh?', such as,

- unused parameters in the method signature
- similar things that 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

Practice KISSing ⚡⚡⚡

Do not try to write 'clever' code. "Keep it simple, stupid" (KISS), as the old adage goes. For example, do not dismiss the brute-force yet simple solution in favor of a complicated one because of some 'supposed benefits' such as 'better reusability' unless you have a strong justification.

> ❝ Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. ❞ -- Brian W. Kernighan

> ❝ Programs must be written for people to read, and only incidentally for machines to execute. ❞ -- Abelson and Sussman

Avoid Premature Optimizations ⚡⚡⚡

Optimizing code prematurely has several drawbacks:

- **You may not know which parts are the real performance bottlenecks**. This is especially the case when the code undergoes transformations (e.g., compiling, minifying, transpiling, etc.) before it becomes an executable. Ideally, you should use a profiler tool to identify the actual bottlenecks of the code first, and optimize only those parts.
- **Optimizing can complicate the code**, affecting correctness and readability.
- **Hand-optimized code can be harder for the compiler to optimize** (the simpler the code, the easier it is for the compiler to optimize). In many cases, a compiler can do a better job of optimizing the runtime code if you don't get in the way by trying to hand-optimize the source code.

Make it work, make it right, make it fast is popular saying in the industry, which means in most cases, getting the code to perform correctly should take priority over optimizing it. If the code doesn't work correctly, it has no value no matter how fast/efficient it is.

> ❝ Premature optimization is the root of all evil in programming. ❞ -- Donald Knuth

Of course, there are cases in which optimizing takes priority over other things e.g., when writing code for resource-constrained environments. This guideline is simply a caution that you should optimize only when it is really needed.

SLAP Hard ⚡⚡⚡

Avoid having multiple levels of abstraction within a code fragment. Note: The book *The Productive Programmer* (by Neal Ford) calls this the *Single Level of Abstraction Principle* (SLAP) while the book *Clean Code* (by Robert C. Martin) calls this *One Level of Abstraction per Function*.

👎 **Bad** (`readData();` and `salary = basic * rise + 1000;` are at different levels of abstraction)

```
readData();
salary = basic * rise + 1000;
tax = (taxable ? salary * 0.07 : 0);
displayResult();
```

👍 **Good** (all statements are at the same level of abstraction)

```
readData();
processData();
displayResult();
```

Also ensure that the code is written at the highest level of abstraction possible.

👎 **Bad** (all statements are at a low levels of abstraction)

```
low-level statement A1
low-level statement A2
low-level statement A3
low-level statement B1
low-level statement B2
if condition X :
    low-level statement C1
    low-level statement C2
```

👍 **Good** (all statements are at the same high level of abstraction)

```
high-level step A
high-level step B
if condition X:
  high-level step C
```

That said, it is sometimes possible to pack *two* levels of abstraction into the code without affecting readability that much, provided each step in the higher-level logic is clearly marked using comments and separated (e.g., using a blank line) from adjacent steps.

📦 Example: The following pseudocode has two levels of abstraction.

```
//high-level step A
low-level statement A1
low-level statement A2
low-level statement A3

//high-level step B
low-level statement B1
low-level statement B2

if condition X :
    //high-level step C
    low-level statement C1
    low-level statement C2
```

Advanced

Make the Happy Path Prominent ⚡⚡⚡

The *happy path* should be clear and prominent in your code. Restructure the code to make the happy path (i.e., the execution path taken when everything goes well) less-nested as much as possible. It is the 'unusual' cases that should be nested. Someone reading the code should not get distracted by alternative paths taken when error conditions happen. One technique that could help in this regard is the use of guard clauses.

📦 The following example shows how guard clauses can be used to reduce the nesting of the happy path.

👎 **Bad**

```
if (!isUnusualCase) {  //detecting an unusual condition
    if (!isErrorCase) {
        start();    //main path
        process();
        cleanup();
        exit();
    } else {
        handleError();
    }
} else {
    handleUnusualCase(); //handling that unusual condition
}
```

In the code above,

- unusual condition detections are separated from their handling.
- the main path is nested deeply.

👍 **Good**

```
if (isUnusualCase) { //Guard Clause
    handleUnusualCase();
    return;
}

if (isErrorCase) { //Guard Clause
```

```
    handleError();
    return;
}

start();
process();
cleanup();
exit();
```

In contrast, the above code

- deals with unusual conditions as soon as they are detected so that the reader doesn't have to remember them for long.
- keeps the main path un-indented.

The following pseudocode example shows how to reduce the nesting of the happy path inside a loop using a `continue` statement:

Bad

```
for (condition1)
    if (condition2)
        statement A
        statement B
        statement C
        statement D
statement E
```

Good

```
for (condition1)
    if (not condition2)
        continue
    statement A
    statement B
    statement C
    statement D
statement E
```

Guideline: Follow a standard

Introduction ⚡⚡

One essential way to improve code quality is to follow a consistent style. That is why software engineers usually follow a strict coding standard (aka *style guide*).

The aim of a coding standard is to make the entire codebase look like it was written by one person. A coding standard is usually specific to a programming language and specifies guidelines such as the locations of opening and closing braces, indentation styles and naming styles (e.g., whether to use *Hungarian style*, *Pascal casing*, *Camel casing*, etc.). It is important that the whole team/company uses the same coding standard and that the standard is generally not inconsistent with typical industry practices. If a company's coding standard is very different from what is typically used in the industry, new recruits will take longer to get used to the company's coding style.

IDEs can help to enforce some parts of a coding standard e.g., indentation rules.

Basic ⚡⚡⚡

Go through the Java coding standard at @SE-EDU and learn the *basic* style rules.

Intermediate ⚡⚡⚡

Go through the Java coding standard at @SE-EDU and learn the *intermediate* style rules.

Guideline: Name well

Introduction ⚡⚡

Proper naming improves the readability of code. It also reduces bugs caused by ambiguities regarding the intent of a variable or a method.

> “There are only two hard things in Computer Science: cache invalidation and naming things.” -- Phil Karlton

Basic

Use Nouns for Things and Verbs for Actions ⚡⚡

> ❝ Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. ❞
>
> -- Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*

Use nouns for classes/variables and verbs for methods/functions.

Name for a	👎 Bad	👍 Good
Class	`CheckLimit`	`LimitChecker`
Method	`result()`	`calculate()`

Distinguish clearly between single-valued and multi-valued variables.

👍 Good

```
Person student;
ArrayList<Person> students;
```

Use Standard Words ⚡⚡⚡

Use correct spelling in names. Avoid 'texting-style' spelling. **Avoid foreign language words, slang, and names that are only meaningful within specific contexts/times** e.g., terms from private jokes, a TV show currently popular in your country.

Intermediate

Use Name to Explain ⚡⚡⚡

A name is not just for differentiation; it should explain the named entity to the reader accurately and at a sufficient level of detail.

👎 Bad	👍 Good
`processInput()` (what 'process'?)	`removeWhiteSpaceFromInput()`
`flag`	`isValidInput`
`temp`	

If a name has multiple words, they should be in a sensible order.

👎 Bad	👍 Good
`bySizeOrder()`	`orderBySize()`

Imagine going to the doctor's and saying "My eye1 is swollen"! Don't use numbers or case to distinguish names.

👎 Bad	👎 Bad	👍 Good
`value1`, `value2`	`value`, `Value`	`originalValue`, `finalValue`

Not Too Long, Not Too Short ⚡⚡⚡

While it is preferable not to have lengthy names, names that are 'too short' are even worse. If you must abbreviate or use acronyms, do it consistently. Explain their full meaning at an obvious location.

Avoid Misleading Names ⚡⚡⚡

Related things should be named similarly, while unrelated things should NOT.

📦 Example: Consider these variables

- `colorBlack` : hex value for color black
- `colorWhite` : hex value for color white
- `colorBlue` : number of times blue is used
- `hexForRed` : hex value for color red

This is misleading because `colorBlue` is named similar to `colorWhite` and `colorBlack` but has a different purpose while `hexForRed` is named differently but has a very similar purpose to the first two variables. The following is better:

- `hexForBlack` `hexForWhite` `hexForRed`
- `blueColorCount`

Avoid misleading or ambiguous names (e.g., those with multiple meanings), similar sounding names, hard-to-pronounce ones (e.g., avoid ambiguities like "is that a lowercase L, capital I or number 1?", or "is that number 0 or letter O?"), almost similar names.

👎 Bad	👍 Good	Reason
`phase0`	`phaseZero`	Is that zero or letter O?
`rwrLgtDirn`	`rowerLegitDirection`	Hard to pronounce
`right` `left` `wrong`	`rightDirection` `leftDirection` `wrongResponse`	`right` is for 'correct' or 'opposite of 'left'?
`redBooks` `readBooks`	`redColorBooks` `booksRead`	`red` and `read` (past tense) sounds the same
`FiletMignon`	`egg`	If the requirement is just a name of a food, `egg` is a much easier to type/say choice than `FiletMignon`

Guideline: Avoid unsafe shortcuts

Introduction ⚡⚡

It is safer to use language constructs in the way they are meant to be used, even if the language allows shortcuts. Such coding practices are common sources of bugs. Know them and avoid them.

Basic

Use the Default Branch ⚡⚡

Always include a default branch in `case` statements. This ensures that all possible outcomes have been considered at the branching point.

Furthermore, use the `default` branch for the intended default action and not just to execute the last option. If there is no default action, you can use the `default` branch to detect errors (i.e., if execution reached the `default` branch, raise a suitable error). This also applies to the final `else` of an `if-else` construct. That is, the final `else` should mean 'everything else', and not the final option. Do not use `else` when an `if` condition can be explicitly specified, unless there is absolutely no other possibility.

👎 Bad

```
if (red) print "red";
else print "blue";
```

👍 Good

```
if (red) print "red";
else if (blue) print "blue";
else error("incorrect input");
```

Don't Recycle Variables or Parameters ⚡⚡

- **Use one variable for one purpose.** Do not reuse a variable for a different purpose other than its intended one, just because the data type is the same.
- **Do not *reuse* formal parameters as local variables** inside the method.

👎 Bad

```
double computeRectangleArea(double length, double width) {
    length = length * width;  // parameter reused as a variable
    return length;
}
```

👍 Good

```
double computeRectangleArea(double length, double width) {
    double area;
    area = length * width;
    return area;
}
```

Avoid Empty Catch Blocks ⚡⚡

Avoid empty `catch` statements, as they are a way to ignore errors silently (which is not a good thing). In cases when it is unavoidable, at least give a comment to explain why the `catch` block is left empty.

Delete Dead Code ⚡⚡

Get rid of unused code the moment it becomes redundant. You might feel reluctant to delete code you have painstakingly written, even if you have no use for that code anymore ("I spent a lot of time writing that code; what if I need it again?"). Consider all code as baggage you have to carry. If you need that code again, simply recover it from the revision control tool you are using. Deleting code you wrote previously is a sign that you are improving.

Intermediate

Minimize Scope of Variables ⚡⚡⚡

Minimize global variables. Global variables may be the most convenient way to pass information around, but they do create implicit links between code segments that use the global variable. Avoid them as much as possible.

Define variables in the least possible scope. For example, if the variable is used only within the `if` block of the conditional statement, it should be declared inside that `if` block.

> **The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used.** -- *Effective Java*, by Joshua Bloch

Minimize Code Duplication ⚡⚡⚡

Code duplication, especially when you copy-paste-modify code, often indicates a poor quality implementation. While it may not be possible to have zero duplication, always think twice before duplicating code; most often there is a better alternative.

This guideline is closely related to the DRY Principle.

Guideline: Comment minimally, but sufficiently

Introduction ⚡⚡

> Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' Improve the code and then document it to make it even clearer. -- Steve McConnell, Author of *Clean Code*

Some think commenting heavily increases the 'code quality'. That is not so. Avoid writing comments to explain bad code. Improve the code to make it self-explanatory.

Basic

Do Not Repeat the Obvious ⚡⚡

Do not repeat in comments information that is already obvious from the code. If the code is self-explanatory, a comment may not be needed.

👎 **Bad**

```
//increment x
x++;

//trim the input
trimInput();
```

Write to the Reader ⚡⚡

Write comments targeting other programmers reading the code. Do not write comments as if they are private notes to yourself. Instead, One type of comment that is almost always useful is the *header comment* that you write for a class or an operation to explain its purpose.

👎 **Bad** Reason: this comment will only make sense to the person who wrote it

```
// a quick trim function used to fix bug I detected overnight
void trimInput() {
    ....
}
```

👍 **Good**

```
/** Trims the input of leading and trailing spaces */
void trimInput() {
    ....
}
```

Intermediate

Explain WHAT and WHY, not HOW ⚡⚡⚡

Comments should explain the *WHAT* and *WHY* aspects of the code, rather than the *HOW* aspect.

✓ **WHAT: The specification of what the code is *supposed* to do.** The reader can compare such comments to the implementation to verify if the implementation is correct.

📦 Example: This method is possibly buggy because the implementation does not seem to match the comment. In this case, the comment could help the reader to detect the bug.

```
/** Removes all spaces from the {@code input} */
void compact(String input) {
    input.trim();
}
```

✓ **WHY: The rationale for the current implementation.**

📦 Example: Without this comment, the reader will not know the reason for calling this method.

```
// Remove spaces to comply with IE23.5 formatting rules
compact(input);
```

✕ **HOW: The explanation for how the code works.** This should already be apparent from the code, if the code is self-explanatory. Adding comments to explain the same thing is redundant.

📦 Example:

👎 **Bad** Reason: Comment explains how the code works.

```
// return true if both left end and right end are correct
//    or the size has not incremented
return (left && right) || (input.size() == size);
```

👍 **Good** Reason: The code is now self-explanatory -- the comment is no longer needed.

```
boolean isSameSize = (input.size() == size);
return (isLeftEndCorrect && isRightEndCorrect) || isSameSize;
```

Refactoring

What ⚡⚡

The process of restructuring code in small steps without modifying its external behavior is called *refactoring*. Refactoring is needed because the first version of the code you write may not be of production quality. It is OK to first concentrate on making the code work, rather than worry over the quality of the code, as long as you improve the quality later.

- **Refactoring is not rewriting**: Discarding poorly-written code entirely and re-writing it from scratch is not refactoring because refactoring needs to be done in small steps.
- **Refactoring is not bug fixing**: By definition, refactoring is different from bug fixing or any other modifications that alter the external behavior (e.g. adding a feature) of the component in concern.

Refactoring code can have many secondary benefits e.g.

- hidden bugs become easier to spot
- improve performance (sometimes, simpler code runs faster than complex code because simpler code is easier for the compiler to optimize).

Given below are two common refactorings (more).

Refactoring Name: **Consolidate Duplicate Conditional Fragments**

Situation: The same fragment of code is in all branches of a conditional expression.

Method: Move it outside of the expression.

📦 Example:

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
```

→

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Refactoring Name: **Extract Method**

Situation: You have a code fragment that can be grouped together.

Method: Turn the fragment into a method whose name explains the purpose of the method.

📦 Example:

```
void printOwing() {
    printBanner();

    // print details
    System.out.println("name:    " + name);
    System.out.println("amount    " + getOutstanding());
}
```

↓

```
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printDetails(double outstanding) {
    System.out.println("name:    " + name);
    System.out.println("amount    " + outstanding);
```

```
}
```

Some IDEs have builtin support for basic refactorings such as automatically renaming a variable/method/class in all places it has been used.

Refactoring, even if done with the aid of an IDE, may still result in regressions. Therefore, each small refactoring should be followed by regression testing.

How ⚡⚡⚡

There are refactoring catalogs listing various refactorings. Given below are some commonly used refactorings.

1. Consolidate Conditional Expression
2. Decompose Conditional
3. Inline Method
4. Remove Double Negative
5. Replace Magic Literal
6. Replace Nested Conditional with Guard Clauses
7. Replace Parameter with Explicit Methods
8. Reverse Conditional
9. Split Loop
10. Split Temporary Variable

When ⚡⚡⚡

One way to identify refactoring opportunities is by *code smells*.

> A *code smell* is a surface indication that usually corresponds to a deeper problem in the system. First, a smell is by definition something that's quick to spot. Second, smells don't always indicate a problem.
> --adapted from https://martinfowler.com/bliki/CodeSmell.html

An example (from the same source as above) is the code smell *data class* i.e., a class with all data and no behavior. When you encounter the such a class, you can explore if refactoring it to move the corresponding behavior into that class is appropriate. Some more examples:

- Long Method
- Large Class
- Primitive Obsession
- Temporary Field
- Shotgun Surgery

Periodic refactoring is a good way to pay off the *technical debt* a codebase has accumulated.

> Software systems are prone to the build up of **cruft** - deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further. Technical Debt is a metaphor, coined by Ward Cunningham, that frames how to think about dealing with this cruft, thinking of it like a financial debt. The extra effort that it takes to add new features is the interest paid on the debt.
> --https://martinfowler.com/bliki/TechnicalDebt.html

While it is important to refactor frequently so as to avoid the accumulation of 'messy' code (aka technical debt), an important question is how much refactoring is *too much* refactoring? **It is too much refactoring when the benefits no longer justify the cost.** The costs and the benefits depend on the context. That is why some refactorings are 'opposites' of each other (e.g., *extract method* vs *inline method*).

Documentation

Introduction

What ⚡⚡

Developer-to-developer documentation can be in one of two forms:

1. **Documentation for *developer-as-user***: Software components are written by developers and reused by other developers, which means there is a need to document how such components are to be used. Such documentation can take several forms:
   - API documentation: APIs expose functionality in small-sized, independent and easy-to-use chunks, each of which can be documented systematically.
   - Tutorial-style instructional documentation: In addition to explaining functions/methods independently, some higher-level explanations of how to use an API can be useful.

- 📦 Example of API Documentation: String API.
- 📦 Example of tutorial-style documentation: Java Internationalization Tutorial.

2. **Documentation for *developer-as-maintainer***: There is a need to document how a system or a component is designed, implemented and tested so that other developers can maintain and evolve the code. Writing documentation of this type is harder because of the need to explain complex internal details. However, given that readers of this type of documentation usually have access to the source code itself, only *some* information needs to be included in the documentation, as code (and code comments) can also serve as a complementary source of information.

- 📦 An example: se-edu/addressbook-level4 Developer Guide.

Another view proposed by Daniele Procida in this article is as follows:

> **There is a secret that needs to be understood in order to write good software documentation: there isn't one thing called documentation, there are four. They are: tutorials, how-to guides, explanation and technical reference.** They represent four different purposes or functions, and require four different approaches to their creation. Understanding the implications of this will help improve most software documentation - often immensely. ...
>
> | | |
> |---|---|
> | **TUTORIALS**

A tutorial:

• is learning-oriented
• allows the newcomer to get started
• is a lesson

Analogy: teaching a small child how to cook | **HOW-TO GUIDES**

A how-to guide:

• is goal-oriented
• shows how to solve a specific problem
• is a series of steps

Analogy: a recipe in a cookery book |
> | **EXPLANATION**

An explanation:

• is understanding-oriented
• explains
• provides background and context

Analogy: an article on culinary social history | **REFERENCE**

A reference guide:

• is information-oriented
• describes the machinery
• is accurate and complete

Analogy: a reference encyclopedia article |

Software documentation (applies to both user-facing and developer-facing) is best kept in a text format for ease of version tracking. **A writer-friendly source format is also desirable** as non-programmers (e.g., technical writers) may need to author/edit such documents. As a result, formats such as Markdown, AsciiDoc, and PlantUML are often used for software documentation.

Guidelines

Guideline: Go top-down, not bottom-up

What ⚡⚡

When writing project documents, a top-down breadth-first explanation is easier to understand than a bottom-up one.

Why ⚡⚡⚡

The main advantage of the top-down approach is that the document is structured like an upside down tree (root at the top) and **the reader can travel down a path she is interested in until she reaches the component she is interested to learn in-depth**, without having to read the entire document or understand the whole system.

How ⚡⚡

📦 To explain a system called `SystemFoo` with two sub-systems, `FrontEnd` and `BackEnd`, start by describing the system at the highest level of abstraction, and progressively drill down to lower level details. An outline for such a description is given below.

[First, explain what the system is, in a black-box fashion (no internal details, only the external view).]

> `SystemFoo` is a ....

[Next, explain the high-level architecture of `SystemFoo`, referring to its major components only.]

> `SystemFoo` consists of two major components: `FrontEnd` and `BackEnd`.
>
> The job of `FrontEnd` is to ... while the job of `BackEnd` is to ...
>
> And this is how `FrontEnd` and `BackEnd` work together ...

[Now you can drill down to `FrontEnd`'s details.]

> `FrontEnd` consists of three major components: `A`, `B`, `C`
>
> `A`'s job is to ...
> `B`'s job is to...
> `C`'s job is to...
>
> And this is how the three components work together ...

[At this point, further drill down to the internal workings of each component. A reader who is not interested in knowing the nitty-gritty details can skip ahead to the section on `BackEnd`.]

> In-depth description of `A`
>
> In-depth description of `B`
>
> ...

[At this point drill down to the details of the `BackEnd`.]

> ...

Guideline: Aim for comprehensibility

What ⚡⚡

Technical documents exist to help others understand technical details. Therefore, **it is not enough for the documentation to be accurate and comprehensive; it should also be comprehensible**.

How ⚡⚡⚡

Here are some tips on writing effective documentation.

- **Use plenty of diagrams**: It is not enough to explain something in words; complement it with visual illustrations (e.g. a UML diagram).
- **Use plenty of examples**: When explaining algorithms, show a running example to illustrate each step of the algorithm, in parallel to worded explanations.
- **Use simple and direct explanations**: Convoluted explanations and fancy words will annoy readers. Avoid long sentences.
- **Get rid of statements that do not add value**: For example, 'We made sure our system works perfectly' (who didn't?), 'Component X has its own responsibilities' (of course it has!).
- **It is not a good idea to have separate sections for each type of artifact**, such as 'use cases', 'sequence diagrams', 'activity diagrams', etc. Such a structure, coupled with the indiscriminate inclusion of diagrams without justifying their need, indicates a failure to understand the purpose of documentation. Include diagrams when they are needed to explain something. If you want to provide additional diagrams for completeness' sake, include them in the appendix as a reference.

Guideline: Document minimally, but sufficiently

What ⚡⚡

Aim for 'just enough' developer documentation.

- Writing and maintaining developer documents is an overhead. You should try to minimize that overhead.
- If the readers are developers who will eventually read the code, the documentation should complement the code and should provide only just enough guidance to get started.

How ⚡⚡⚡

Anything that is already clear in the code need not be described in words. Instead, **focus on providing higher level information that is not readily visible in the code or comments**.

Refrain from duplicating chunks of text. When describing several similar algorithms/designs/APIs, etc., do not simply duplicate large chunks of text. Instead, **describe the similarities in one place and emphasize only the differences in other places.** It is very annoying to see pages and pages of similar text without any indication as to how they differ from each other.

Tools

JavaDoc

What ⚡⚡

JavaDoc is a tool for generating API documentation in HTML format from comments in the source code. In addition, modern IDEs use JavaDoc comments to generate explanatory tooltips.

📦 An example method header comment in JavaDoc format:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url An absolute URL giving the base location of the image.
 * @param name The location of the image, relative to the url argument.
 * @return The Image at the specified URL.
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

📦 Generated HTML documentation:

getImage
`public Image getImage(URL url, String name)`
Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

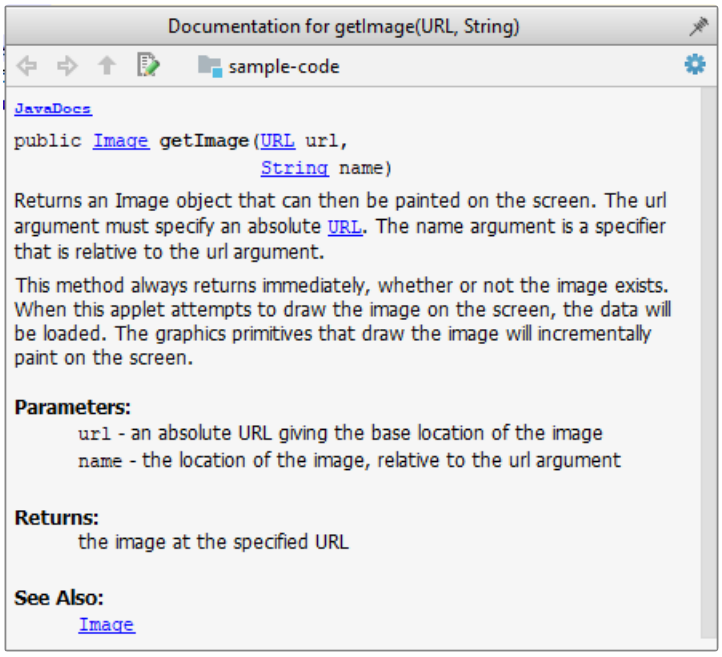
Parameters:
`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

Returns:
the image at the specified URL.

See Also:
`Image`

Tooltip generated by IntelliJ IDE:



How ϟϟϟ

In the absence of more extensive guidelines (e.g., given in a coding standard adopted by your project), you can follow the two examples below in your code.

A minimal JavaDoc comment example for methods:

```
/**
 * Returns lateral location of the specified position.
 * If the position is unset, NaN is returned.
 *
 * @param x X coordinate of position.
 * @param y Y coordinate of position.
 * @param zone Zone of position.
 * @return Lateral location.
 * @throws IllegalArgumentException If zone is <= 0.
 */
public double computeLocation(double x, double y, int zone)
    throws IllegalArgumentException {
    // ...
}
```

A minimal JavaDoc comment example for classes:

```
package ...

import ...

/**
```

```
 * Represents a location in a 2D space. A <code>Point</code> object corresponds to
 * a coordinate represented by two integers e.g., <code>3,6</code>
 */
public class Point {
    // ...
}
```

Error handling

Introduction

What

> Well-written applications include error-handling code that allows them to recover gracefully from unexpected errors. When an error occurs, the application may need to request user intervention, or it may be able to recover on its own. In extreme cases, the application may log the user off or shut down the system. -- Microsoft

Exceptions

What

Exceptions are used to deal with *'unusual' but not entirely unexpected* situations that the program might encounter at runtime.

Exception:

> The term *exception* is shorthand for the phrase "exceptional event." An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. –- Java Tutorial (Oracle Inc.)

Examples:

- A network connection encounters a timeout due to a slow server.
- The code tries to read a file from the hard disk but the file is corrupted and cannot be read.

How

Most languages allow code that encountered an "exceptional" situation to encapsulate details of the situation in an *Exception* object and *throw*/*raise* that object so that another piece of code can *catch* it and deal with it. This is especially useful when the code that encountered the unusual situation does not know how to deal with it.

The extract below from the -- Java Tutorial (with slight adaptations) explains how exceptions are typically handled.

When an error occurs at some point in the execution, the code being executed creates an *exception object* and hands it off to the runtime system. The exception object contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing* an exception.

After a method throws an exception, the runtime system attempts to find something to handle it in the *call stack*. The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch* the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the program terminates.

Advantages of exception handling in this way:

- The ability to propagate error information through the call stack.
- The separation of code that deals with 'unusual' situations from the code that does the 'usual' work.

When

In general, use exceptions only for 'unusual' conditions. Use normal `return` statements to pass control to the caller for conditions that are 'normal'.

Assertions

What

***Assertions* are used to define assumptions about the program state so that the runtime can verify them.** An assertion failure indicates a possible bug in the code because the code has resulted in a program state that violates an assumption about how the code *should* behave.

> An assertion can be used to express something like *when the execution comes to this point, the variable `v` cannot be null*.

If the runtime detects an *assertion failure*, it typically takes some drastic action such as terminating the execution with an error message. This is because an assertion failure indicates a possible bug and the sooner the execution stops, the safer it is.

> In the Java code below, suppose you set an assertion that `timeout` returned by `Config.getTimeout()` is greater than `0`. Now, if `Config.getTimeout()` returns `-1` in a specific execution of this line, the runtime can detect it as an *assertion failure* -- i.e., an assumption about the expected behavior of the code turned out to be wrong which could potentially be the result of a bug -- and take some drastic action such as terminating the execution.
>
> ```
> int timeout = Config.getTimeout();
> // set assertion here ...
> ```

How

Use the `assert` keyword to define assertions.

> 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";
> ...
> ```

Assertions can be disabled without modifying the code.

> `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.** This could create a situation where you think all assertions are being verified as `true` while in fact they are not being verified at all. Therefore, remember to enable assertions when you run the program if you want them to be in effect.

Enable assertions in IntelliJ (how?) and get an assertion to fail temporarily (e.g. insert an `assert false` into the code temporarily) to confirm assertions are being verified.

Java `assert` vs JUnit assertions: Both check for a given condition but JUnit assertions are more powerful and customized for testing. In addition, JUnit assertions are not disabled by default. Use JUnit assertions in test code and Java `assert` in functional code.

When

It is recommended that assertions be used liberally in the code. Their impact on performance is low, and worth the additional safety they provide.

Do not use assertions to do *work* because assertions can be disabled. If not, your program will stop working when assertions are not enabled.

> The code below will not invoke the `writeFile()` method when assertions are disabled. If that method is performing some work that is necessary for your program, your program will not work correctly when assertions are disabled.

```
...
assert writeFile() : "File writing is supposed to return true";
```

Assertions are suitable for verifying assumptions about *Internal Invariants*, *Control-Flow Invariants*, *Preconditions*, *Postconditions*, and *Class Invariants*. Refer to Programming with Assertions (second half) to learn more.

Exceptions and assertions are two complementary ways of handling errors in software but they serve different purposes. Therefore, both assertions and exceptions should be used in code.

- The raising of an exception indicates an unusual condition created by the user (e.g. user inputs an unacceptable input) or the environment (e.g., a file needed for the program is missing).
- An assertion failure indicates the programmer made a mistake in the code (e.g., a null value is returned from a method that is not supposed to return null under any circumstances).

Logging

What ⚡⚡

***Logging* is the deliberate recording of certain information during a program execution for future reference.** Logs are typically written to a log file but it is also possible to log information in other ways e.g. into a database or a remote server.

***Logging* can be useful for troubleshooting problems**. A good logging system records some system information regularly. When bad things happen to a system e.g. an unanticipated failure, their associated log files may provide indications of what went wrong and actions can then be taken to prevent it from happening again.

💡 **A log file is like the black box of an airplane**; they don't prevent problems but they can be helpful in understanding what went wrong after the fact.

How ⚡⚡⚡

Most programming environments come with logging systems that allow sophisticated forms of logging. They have features such as the ability to enable and disable logging easily or to change the logging intensity.

📦 This sample Java code uses Java's default logging mechanism.

First, import the relevant Java package:

```
import java.util.logging.*;
```

Next, create a `Logger`:

```
private static Logger logger = Logger.getLogger("Foo");
```

Now, you can use the `Logger` object to log information. Note the use of a logging level for each message. When running the code, the logging level can be set to `WARNING` so that log messages specified as having `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");
```

Defensive programming

What ⚡⚡

A defensive programmer codes under the assumption "if you leave room for things to go wrong, they *will* go wrong". Therefore, a defensive programmer proactively tries to eliminate any room for things to go wrong.

📦 Consider a method `MainApp#getConfig()` that returns a `Config` object containing configuration data. A typical implementation is given below:

```
class MainApp {
    Config config;

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

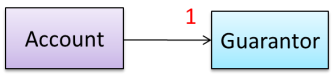
If the returned `Config` object is not meant to be modified, a defensive programmer might use a more *defensive* implementation given below. This is more defensive because even if the returned `Config` object is modified (although it is not meant to be), it will not affect the `config` object inside the `MainApp` object.

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

Enforcing Compulsory Associations ⚡⚡⚡

Consider two classes, `Account` and `Guarantor`, with an association as shown in the following diagram:

📦 Example:



Here, the association is compulsory i.e., an `Account` object should always be linked to a `Guarantor`. One way to implement this is to simply use a reference variable, like this:

```
class Account {
    Guarantor guarantor;

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

However, what if someone else used the `Account` class like this?

```
Account a = new Account();
a.setGuarantor(null);
```

This results in an `Account` without a `Guarantor`! In a real banking system, this could have serious consequences! The code here did not try to prevent such a thing from happening. You can make the code more defensive by proactively enforcing the multiplicity constraint, like this:

```
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;
    }
    // ...
}
```

When ⚡⚡⚡

It is not necessary to be 100% defensive all the time. While defensive code may be less prone to be misused or abused, such code can also be more complicated and slower to run.

The suitable degree of defensiveness depends on many factors such as:

- 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

Integration

Introduction

What

Combining parts of a software product to form a whole is called *integration*. It is also one of the most troublesome tasks and it rarely goes smoothly.

Approaches

'Late and One Time' vs 'Early and Frequent'

In terms of timing and frequency, there are two general approaches to integration: *late and one-time*, *early and frequent*.

***Late and one-time*: wait till all components are completed and integrate all finished components near the end of the project.**

✕ This approach is not recommended because integration often causes many component incompatibilities (due to previous miscommunications and misunderstandings) to surface which can lead to delivery delays i.e., Late integration → incompatibilities found → major rework required → cannot meet the delivery date.

***Early and frequent*: integrate early and evolve each part in parallel, in small steps, re-integrating frequently.**

A *walking skeleton* can be written first. This can be done by one developer, possibly the one in charge of integration. After that, all developers can flesh out the skeleton in parallel, adding one feature at a time. After each feature is done, simply integrate the new code into the main system.

Big-Bang vs Incremental Integration

***Big-bang integration*: integrate all (or too many) components at the same time.** More generally, integrating too many changes at the same time.

✕ Big-bang is not recommended because it will uncover too many problems at the same time which could make debugging and bug-fixing more complex than when problems are uncovered incrementally.

***Incremental integration*: integrate a few components at a time.** More generally, integrating changes gradually. This approach is better than big-bang integration because it surfaces integration problems in a more manageable way.

Build Automation

What

Build automation tools automate the steps of the build process, usually by means of build scripts.

In a non-trivial project, building a product from its source code can be a complex multistep process. For example, it can include steps such as: pull code from the revision control system, compile, link, run automated tests, automatically update release documents (e.g., build number), package into a distributable, push to repo, deploy to a server, delete temporary files created during building/testing, email developers of the new build, and so on. Furthermore, this build process can be done 'on demand', it can be scheduled (e.g., every day at midnight) or it can be triggered by various events (e.g., triggered by a code push to the revision control system).

Some of these build steps such as compiling, linking and packaging, are already automated in most modern IDEs. For example, several steps happen automatically when the 'build' button of the IDE is clicked. Some IDEs even allow customization of this build process to some extent.

However, most big projects use specialized build tools to automate complex build processes.

Some popular build tools relevant to Java developers: Gradle, Maven, Apache Ant, GNU Make

Some other build tools: Grunt (JavaScript), Rake (Ruby)

Some build tools also serve as *dependency management tools*. Modern software projects often depend on third party libraries that evolve constantly. That means developers need to download the correct version of the required libraries and update them regularly. Therefore, dependency management is an important part of build automation. Dependency management tools can automate that aspect of a project.

Maven and Gradle, in addition to managing the build process, can play the role of dependency management tools too.

Continuous Integration and Continuous Deployment

An extreme application of build automation is called *continuous integration (CI)* in which integration, building, and testing happens automatically after each code change.

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.

Some examples of CI/CD tools: Travis, Jenkins, Appveyor, CircleCI, GitHub Actions

Reuse

Introduction

What ⚡⚡

Reuse is a major theme in software engineering practices. **By reusing tried-and-tested components, the robustness of a new software system can be enhanced while reducing the manpower and time requirement.** Reusable components come in many forms; it can be reusing a piece of code, a subsystem, or a whole software.

When ⚡⚡⚡

While you may be tempted to use many libraries/frameworks/platforms that seem to crop up on a regular basis and promise to bring great benefits, note that **there are costs associated with reuse**. Here are some:

- The reused code **may be an overkill** (think *using a sledgehammer to crack a nut*), increasing the size of, and/or degrading the performance of, your software.
- The reused software **may not be mature/stable enough** to be used in an important product. That means the software can change drastically and rapidly, possibly in ways that break your software.
- Non-mature software has the **risk of dying off** as fast as they emerged, leaving you with a dependency that is no longer maintained.
- The license of the reused software (or its dependencies) **restrict how you can use/develop your software**.
- The reused software **might have bugs, missing features, or security vulnerabilities** that are important to your product, but not so important to the maintainers of that software, which means those flaws will not get fixed as fast as you need them to.
- **Malicious code can sneak into your product** via compromised dependencies.

APIs

What ⚡⚡

An *Application Programming Interface (API)* specifies the interface through which other programs can interact with a software component. It is a contract between the component and its clients.

> 📦 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.
>
> 📦 The GitHub API is a collection of web request formats that the GitHub server accepts and their corresponding responses. You can write a program that interacts with GitHub through that API.

When developing large systems, if you define the API of each component early, the development team can develop the components in parallel because the future behavior of the other components are now more predictable.

Libraries

What ⚡

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`

How ⚡⚡

These are the typical steps required to use a library:

1. Read the documentation to confirm that its functionality fits your needs.

2. Check the license to confirm that it allows reuse in the way you plan to reuse it. For example, some libraries might allow non-commercial use only.
3. Download the library and make it accessible to your project. Alternatively, you can configure your dependency management tool to do it for you.
4. Call the library API from your code where you need to use the library's functionality.

Frameworks

What ⚡⚡

The overall structure and execution flow of a specific category of software systems can be very similar. The similarity is an opportunity to reuse at a high scale.

📦 Running example:

IDEs for different programming languages are similar in how they support editing code, organizing project files, debugging, etc.

A *software framework* is a reusable implementation of a software (or part thereof) providing *generic* functionality that can be selectively customized to produce a *specific* application.

📦 Running example:

Eclipse is an IDE framework that can be used to create IDEs for different programming languages.

Some frameworks provide a complete implementation of a *default* behavior which makes them immediately usable.

📦 Running example:

Eclipse is a fully functional Java IDE out-of-the-box.

A framework facilitates the adaptation and customization of some desired functionality.

📦 Running example:

The Eclipse plugin system can be used to create an IDE for different programming languages while reusing most of the existing IDE features of Eclipse.

e.g., https://marketplace.eclipse.org/content/pydev-python-ide-eclipse

Some frameworks cover only a specific component or an aspect.

📦 JavaFX is a framework for creating Java GUIs. Tkinter is a GUI framework for Python.

📦 More examples of frameworks

- Frameworks for web-based applications: Drupal (PHP), Django (Python), Ruby on Rails (Ruby), Spring (Java)
- Frameworks for testing: JUnit (Java), unittest (Python), Jest (JavaScript)

Frameworks vs Libraries ⚡⚡⚡

Although both frameworks and libraries are reuse mechanisms, there are notable differences:

- **Libraries are meant to be used 'as is' while frameworks are meant to be customized/extended.** e.g., writing plugins for Eclipse so that it can be used as an IDE for different languages (C++, PHP, etc.), adding modules and themes to Drupal, and adding test cases to JUnit.

- **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"** (i.e., don't call us, we'll call you!). That is, you write code that will be called by the framework, e.g., writing test methods that will be called by the JUnit framework. In the case of libraries, your code calls libraries.

Platforms

What ⚡⚡⚡

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.

📦 Technically, an operating system can be called a platform. For example, Windows PC is a platform for desktop applications while iOS is a platform for mobile applications.

📦 **Two well-known examples of platforms are JavaEE and .NET**, both of which sit above the operating systems layer, and are used to develop enterprise applications. Infrastructure services such as connection pooling, load balancing, remote code execution, transaction management, authentication, security, messaging etc. are done similarly in most enterprise applications. Both JavaEE and .NET provide these services to applications in a customizable way without developers having to implement them from scratch every time.

- JavaEE (Java Enterprise Edition) is both a framework and a platform for writing enterprise applications. The runtime used by JavaEE applications is the JVM (Java Virtual Machine) that can run on different Operating Systems.
- .NET is a similar platform and framework. Its runtime is called CLR (Common Language Runtime) and it is usually used on Windows machines.

SECTION: QUALITY ASSURANCE

Quality assurance

Introduction

What ⚡⚡

Software *Quality Assurance* (QA) is the process of ensuring that the software being built has the required levels of quality.

While testing is the most common activity used in QA, there are other complementary techniques such as *static analysis, code reviews,* and *formal verification*.

Validation vs Verification ⚡⚡⚡

Quality Assurance = Validation + Verification

QA involves checking two aspects:

1. Validation: are you *building the right system* i.e., are the requirements correct?
2. Verification: are you *building the system right* i.e., are the requirements implemented correctly?

Whether something belongs under validation or verification is not that important. What is more important is that both are done, instead of limiting to only verification (i.e., remember that the requirements can be wrong too).

Code reviews

What ⚡⚡⚡

Code review is the systematic examination of code with the intention of finding where the code can be improved.

Reviews can be done in various forms. Some examples below:

- **Pull Request reviews**
  - Project Management Platforms such as GitHub and BitBucket allow the new code to be proposed as *Pull Requests* and provide the ability for others to review the code in the PR.
- **In pair programming**
  - As pair programming involves two programmers working on the same code at the same time, there is an implicit review of the code by the other member of the pair.
- **Formal inspections**
  - Inspections involve a group of people systematically examining project artifacts to discover defects. Members of the inspection team play various roles during the process, such as:
    - 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 review over testing:

- It can detect functionality defects as well as other problems such as coding standard violations.
- It can verify non-code artifacts and incomplete code.
- It does not require test drivers or stubs.

Disadvantages:

- It is a manual process and therefore, error prone.

Static analysis

What ⚡⚡⚡

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

Static analysis of code can find useful information such as unused variables, unhandled exceptions, style errors, and statistics. Most modern IDEs come with some inbuilt static analysis capabilities. For example, an IDE can highlight unused variables as you type the code into the editor.

The term *static* in static analysis refers to the fact that the code is analyzed without executing the code. **In contrast, *dynamic analysis* requires the code to be executed to gather additional information about the code** e.g., performance characteristics.

Higher-end static analysis tools (static analyzers) can perform more complex analysis such as locating potential bugs, memory leaks, inefficient code structures, etc.

Some example static analyzers for Java: CheckStyle, PMD, FindBugs

***Linters* are a subset of static analyzers** that specifically aim to locate areas where the code can be made 'cleaner'.

Formal verification

What ⚡⚡⚡

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:

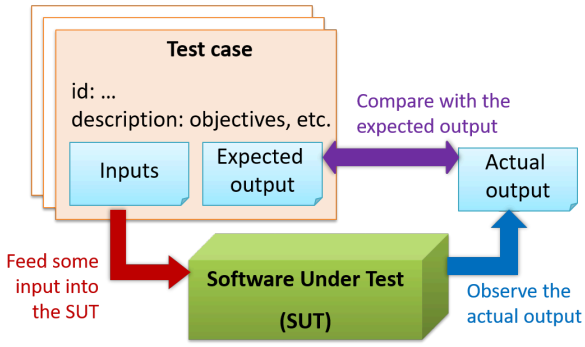
- It only proves the compliance with the specification, but not the actual utility of the software.
- It requires highly specialized notations and knowledge which makes it an expensive technique to administer. Therefore, **formal verifications are more commonly used in safety-critical software such as flight control systems**.

Testing

Introduction

What ⚡⚡

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. –- source: IEEE



When testing, you execute a set of *test cases*. A test case specifies how to perform a test. At a minimum, it specifies the input to the *software under test (SUT)* and the expected behavior.

Example: A minimal test case for testing a browser:

- **Input** – Start the browser using a blank page (vertical scrollbar disabled). Then, load `longfile.html` located in the `test data` folder.
- **Expected behavior** – The scrollbar should be automatically enabled upon loading `longfile.html` .

Test cases can be determined based on the specification, reviewing similar existing systems, or comparing to the past behavior of the SUT.

For each test case you should do the following:

1. Feed the input to the SUT
2. Observe the actual output
3. Compare actual output with the expected output

A test case *failure* is a mismatch between the expected behavior and the actual behavior. A failure indicates a potential *defect* (or a bug) -- we say 'potential' because the error could be in the test case itself.

Example: In the browser example above, a test case failure is implied if the scrollbar remains disabled after loading `longfile.html` . The defect/bug causing that failure could be an uninitialized variable.

Testability ⚡⚡⚡

***Testability* is an indication of how easy it is to test an SUT.** As testability depends a lot on the design and implementation, you should try to increase the testability when you design and implement software. The higher the testability, the easier it is to achieve better quality software.

Testing types

Regression testing

What ⚡⚡

When you modify a system, the modification may result in some unintended and undesirable effects on the system. Such an effect is called a *regression*.

***Regression testing* is the re-testing of the software to detect regressions.** The typical way to detect regressions is retesting all related components, even if they had been tested before.

Regression testing is more effective when it is done frequently, after each small change. However, doing so can be prohibitively expensive if testing is done manually. Hence, **regression testing is more practical when it is automated**.

Developer testing

What ⚡⚡

***Developer testing* is the testing done by the developers themselves** as opposed to dedicated testers or end-users.

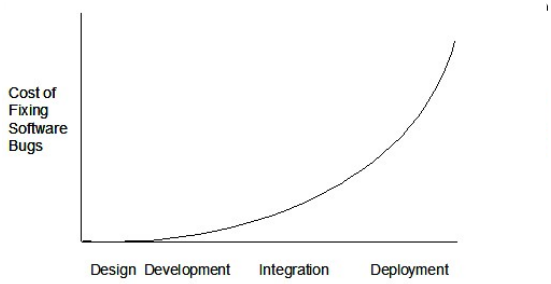
Why ⚡⚡⚡

Delaying testing until the full product is complete has a number of disadvantages:

- **Locating the cause of a test case failure is difficult due to the larger search space;** in a large system, the search space could be millions of lines of code, written by hundreds of developers! The failure may also be due to multiple inter-related bugs.
- **Fixing a bug found during such testing could result in major rework**, especially if the bug originated from the design or during requirements specification i.e., a faulty design or faulty requirements.
- **One bug might 'hide' other bugs**, which could emerge only after the first bug is fixed.
- **The delivery may have to be delayed** if too many bugs are found during testing.

Therefore, it is better to do early testing, as hinted by the popular rule of thumb given below, also illustrated by the graph below it.

> The earlier a bug is found, the easier and cheaper to have it fixed.



Such early testing software is usually, and often by necessity, done by the developers themselves i.e., developer testing.

Unit testing

What ⚡

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

In OOP code, it is common to write one or more unit tests for each public method of a class.

Here are the code skeletons for a `Foo` class containing two methods and a `FooTest` class that contains unit tests for those two methods.

```
class Foo {
    String read() {
        // ...
    }

    void write(String input) {
        // ...
    }

}
```

```
class FooTest {

    @Test
    void read() {
        // a unit test for Foo#read() method
    }

    @Test
    void write_emptyInput_exceptionThrown() {
        // a unit tests for Foo#write(String) method
    }

    @Test
    void write_normalInput_writtenCorrectly() {
        // another unit tests for Foo#write(String) method
    }
}
```

Stubs ⚡⚡⚡

A proper unit test requires the *unit* to be tested in isolation so that bugs in the dependencies cannot influence the test i.e., bugs outside of the unit should not affect the unit tests.

If a `Logic` class depends on a `Storage` class, unit testing the `Logic` class requires isolating the `Logic` class from the `Storage` class.

***Stubs* can isolate the SUT 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 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.

Consider the code below:

```
class Logic {
    Storage s;

    Logic(Storage s) {
        this.s = s;
    }

    String getName(int index) {
        return "Name: " + s.getName(index);
    }
}
```

```
interface Storage {
    String getName(int index);
}
```

```
class DatabaseStorage implements Storage {

    @Override
    public String getName(int index) {
        return readValueFromDatabase(index);
    }

    private String readValueFromDatabase(int index) {
        // retrieve name from the database
    }
}
```

Normally, you would use the `Logic` class as follows (note how the `Logic` object depends on a `DatabaseStorage` object to perform the `getName()` operation):

```
Logic logic = new Logic(new DatabaseStorage());
String name = logic.getName(23);
```

You can test it like this:

```
@Test
void getName() {
    Logic logic = new Logic(new DatabaseStorage());
    assertEquals("Name: John", logic.getName(5));
}
```

However, this `logic` object being tested is making use of a `DataBaseStorage` object which means a bug in the `DatabaseStorage` class can affect the test. Therefore, this test is not testing `Logic` *in isolation from its dependencies* and hence it is not a pure unit test.

Here is a stub class you can use in place of `DatabaseStorage` :

```
class StorageStub implements Storage {

    @Override
    public String getName(int index) {
        if (index == 5) {
            return "Adam";
        } else {
            throw new UnsupportedOperationException();
        }
    }
}
```

Note how the `StorageStub` has the same interface as `DatabaseStorage` , but is so simple that it is unlikely to contain bugs, and is pre-configured to respond with a hard-coded response, presumably, the correct response `DatabaseStorage` is expected to return for the given test input.

Here is how you can use the stub to write a unit test. This test is not affected by any bugs in the `DatabaseStorage` class and hence is a pure unit test.

```
@Test
void getName() {
    Logic logic = new Logic(new StorageStub());
    assertEquals("Name: Adam", logic.getName(5));
}
```

In addition to Stubs, there are other type of replacements you can use during testing, e.g., *Mocks*, *Fakes*, *Dummies*, *Spies*.

Integration testing

What ⚡⚡

***Integration testing* : testing whether different parts of the software *work together* (i.e., integrates) as expected.** Integration tests aim to discover bugs in the 'glue code' related to how components interact with each other. These bugs are often the result of misunderstanding what the parts are supposed to do vs what the parts are actually doing.

📦 Suppose a class `Car` uses classes `Engine` and `Wheel` . If the `Car` class assumed a `Wheel` can support a speed of up to 200 mph but the actual `Wheel` can only support a speed of up to 150 mph, it is the integration test that is supposed to uncover this discrepancy.

How ⚡⚡⚡

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

Suppose a class `Car` uses classes `Engine` and `Wheel`. Here is how you would go about doing 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 the `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 minimize the need for stubs.

Here's how a hybrid unit+integration approach could be applied to the same example used above:

(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 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).

Note that you no longer need stubs for `Engine` and `Wheel`. The downside is that `Car` is never tested in isolation of its dependencies. Given that its dependencies are already unit tested, the risk of bugs in `Engine` and `Wheel` affecting the testing of `Car` can be considered minimal.

System testing

What ⚡⚡

***System testing*: take the *whole system* and test it *against the system specification*.**

System testing is typically done by a testing team (also called a QA team).

System test cases are based on the specified external behavior of the system. Sometimes, system tests go beyond the bounds defined in the specification. This is useful when testing that the system fails 'gracefully' when pushed beyond its limits.

Suppose the SUT is a browser that is supposedly capable of handling web pages containing up to 5000 characters. Given below is a test case to test if the SUT fails gracefully if pushed beyond its limits.

```
Test case: load a web page that is too big
* Input: loads a web page containing more than 5000 characters.
* Expected behavior: aborts the loading of the page
  and shows a meaningful error message.
```

This test case would fail if the browser attempted to load the large file anyway and crashed.

System testing includes testing against non-functional requirements too. Here are some examples:

- *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.

Alpha and beta testing

What ⚡⚡⚡

***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.**

An *open beta release* is the release of not-yet-production-quality-but-almost-there software to the general population. For example, Google's Gmail was in 'beta' for many years before the label was finally removed.

Dogfooding

What ⚡⚡⚡

***Dogfooding* is when creators use their own product** in order to experience how end users experience the product. The term is supposedly derived from the phrase "eating our own dogfood". Dogfooding is different from regular testing in that you *become* an end user, rather than *pretend to be* an end user.

For example, suppose a company produces an email client software. Then, getting some of the employees to use that software for their day-to-day emailing would be dogfooding. Such longer-term, consistent, and authentic use of the software can point to areas of improvement that regular testing (which is often short-term and 'simulated') might not encounter.

Note that dogfooding to be useful, observations need to be deliberately collected and processed i.e., just using the product itself is not enough.

Exploratory versus scripted testing

What ⚡⚡⚡

Here are two alternative approaches to testing a software: *Scripted* testing and *Exploratory* testing.

1. **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.

2. **Exploratory testing:** Devise test cases on-the-fly, creating new test cases based on the results of the past test cases.

Exploratory testing is 'the simultaneous learning, test design, and test execution' [source: bach-et-explained] whereby the nature of the follow-up test case is decided based on the behavior of the previous test cases. In other words, running the system and trying out various operations. It is called *exploratory testing* because testing is driven by observations during testing. Exploratory testing usually starts with areas identified as error-prone, based on the tester's past experience with similar systems. One tends to conduct more tests for those operations where more faults are found.

📦 Here is an example thought process behind a segment of an exploratory testing session:

> "Hmm... looks like feature x is broken. This usually means feature n and k could be broken too; you need to look at them soon. But before that, you should give a good test run to feature y because users can still use the product if feature y works, even if x doesn't work. Now, if feature y doesn't work 100%, you have a major problem and this has to be made known to the development team sooner rather than later..."

💡 **Exploratory testing is also known as *reactive testing, error guessing technique, attack-based testing,* and *bug hunting*.**

When ⚡⚡⚡

Which approach is better – **scripted or exploratory? A mix is better.**

The success of exploratory testing depends on the tester's prior experience and intuition. Exploratory testing should be done by experienced testers, using a clear strategy/plan/framework. Ad-hoc exploratory testing by unskilled or inexperienced testers without a clear strategy is not recommended for real-world non-trivial systems. While **exploratory testing may allow us to detect some problems in a relatively short time, it is not prudent to use exploratory testing as the sole means of testing a critical system**.

Scripted testing is more systematic, and hence, likely to discover more bugs given sufficient time, while exploratory testing would aid in quick error discovery, especially if the tester has a lot of experience in testing similar systems.

> In some contexts, you will achieve your testing mission better through a more scripted approach; in other contexts, your mission will benefit more from the ability to create and improve tests as you execute them. I find that most situations benefit from a mix of scripted and exploratory approaches. --[source: bach-et-explained]

Acceptance testing

What ⚡⚡

***Acceptance testing* (aka *User Acceptance Testing (UAT)*: test the system to ensure it meets the user requirements.**

Acceptance tests 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. Successful completion of UAT is often a prerequisite to the project sign-off.

Acceptance vs System Testing ⚡⚡⚡

Acceptance testing comes after system testing. Similar to system testing, acceptance testing involves testing the whole system.

Some differences between system testing and acceptance testing:

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

Note: *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

Requirement specification versus system specification

The requirement specification need not be the same as the system specification. Some example differences:

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)

However, **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. Some examples:

- The system might work on the testbed environments but might not work the same way in the deployment environment, due to subtle differences between the two environments.
- The system might conform to the system specification but could fail to solve the problem it was supposed to solve for the user, due to flaws in the system design.

Test automation

What ⚡⚡

An automated test case can be run programmatically and the result of the test case (pass or fail) is determined programmatically. Compared to manual testing, automated testing reduces the effort required to run tests repeatedly and increases precision of testing (because manual testing is susceptible to human errors).

Automated Testing of CLI Apps ⚡⚡

A simple way to semi-automate testing of a CLI (Command Line Interface) app is by using input/output re-direction. Here are the high-level steps:

- First, you feed the app with a sequence of test inputs that is stored in a file while redirecting the output to another file.
- Next, you compare the actual output file with another file containing the expected output.

Let's assume you are testing a CLI app called `AddressBook`. Here are the detailed steps:

1. Store the test input in the text file `input.txt`.

2. Store the output you expect from the SUT in another text file `expected.txt`.

3. Run the program as given below, which will redirect the text in `input.txt` as the input to `AddressBook` and similarly, will redirect the output of `AddressBook` to a text file `output.txt`. Note that this does not require any changes in `AddressBook` code.

   ```
   java AddressBook < input.txt > output.txt
   ```

   - The way to run a CLI program differs based on the language.
     e.g., In Python, assuming the code is in `AddressBook.py` file, use the command
     `python AddressBook.py < input.txt > output.txt`

   - If you are using Windows, use a normal MS-DOS terminal (i.e., `cmd.exe`) to run the app, not a PowerShell window.

4. Next, you compare `output.txt` with the `expected.txt`. This can be done using a utility such as Windows' `FC` (i.e., File Compare) command, Unix's `diff` command, or a GUI tool such as *WinMerge*.

   ```
   FC output.txt expected.txt
   ```

Note that the above technique is only suitable when testing CLI apps, and only if the exact output can be predetermined. If the output varies from one run to the other (e.g., it contains a time stamp), this technique will not work. In those cases, you need more sophisticated ways of automating tests.

Test Automation Using Test Drivers ⚡⚡⚡

A test driver is the code that 'drives' the SUT for the purpose of testing i.e., invoking the SUT with test inputs and verifying if the behavior is as expected.

`PayrollTest` 'drives' the `Payroll` class by sending it test inputs and verifies if the output is as expected.

```
public class PayrollTest {
    public static void main(String[] args) throws Exception {

        // test setup
        Payroll p = new Payroll();

        // test case 1
        p.setEmployees(new String[]{"E001", "E002"});
        // automatically verify the response
        if (p.totalSalary() != 6400) {
            throw new Error("case 1 failed ");
```

```
        }

        // test case 2
        p.setEmployees(new String[]{"E001"});
        if (p.totalSalary() != 2300) {
            throw new Error("case 2 failed ");
        }

        // more tests...

        System.out.println("All tests passed");
    }
}
```

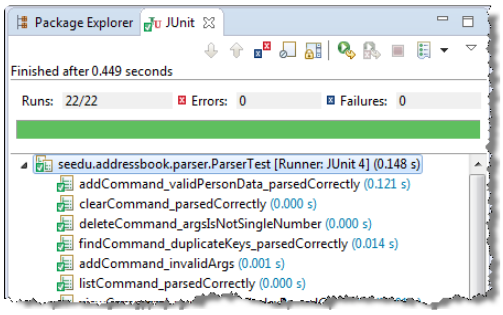
Test Automation Tools

JUnit is a tool for automated testing of Java programs. Similar tools are available for other languages and for automating different types of testing.

This is an automated test for a `Payroll` class, written using JUnit libraries.

```
16      // other test methods
17
18      @Test
19      public void testTotalSalary() {
20          Payroll p = new Payroll();
21
22          // test case 1
23          p.setEmployees(new String[]{"E001", "E002"});
24          assertEquals(6400, p.totalSalary());
25
26          // test case 2
27          p.setEmployees(new String[]{"E001"});
28          assertEquals(2300, p.totalSalary());
29
30          // more tests...
31      }
```

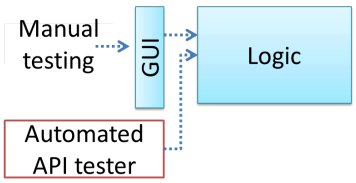
Most modern IDEs have integrated support for testing tools. The figure below shows the JUnit output when running some JUnit tests using the Eclipse IDE.



Automated Testing of GUIs

If a software product has a GUI (Graphical User Interface) component, all product-level testing (i.e., the types of testing mentioned above) need to be done using the GUI. However, **testing the GUI is much harder than testing the CLI (Command Line Interface) or API**, for the following reasons:

- Most GUIs can support a large number of different operations, many of which can be performed in any arbitrary order.
- GUI operations are more difficult to automate than API testing. Reliably automating GUI operations and automatically verifying whether the GUI behaves as expected is harder than calling an operation and comparing its return value with an expected value. Therefore, automated regression testing of GUIs is rather difficult.
- The appearance of a GUI (and sometimes even behavior) can be different across platforms and even environments. For example, a GUI can behave differently based on whether it is minimized or maximized, in focus or out of focus, and in a high resolution display or a low resolution display.



Moving as much logic as possible out of the GUI can make GUI testing easier. That way, you can bypass the GUI to test the rest of the system using automated API testing. While this still requires the GUI to be tested, the number of such test cases can be reduced as most of the system will have been tested using automated API testing.

There are testing tools that can automate GUI testing.

📦 Some tools used for automated GUI testing:

- **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

Test coverage

What ⚡⚡

***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.

Here are some examples of different coverage criteria:

- **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`] [see note 1]
- `(x < 44) == false` : [e.g., `x == 100`]
- `(x > 2) == false` : [e.g., `x == 0`]

Note 1: A case where both conditions are `true` is needed because most execution environments use a *short circuiting* behavior for compound boolean expressions e.g., given an expression `c1 && c2` , `c2` will not be evaluated if `c1` is `false` (as the final result is going to be `false` anyway).

- **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)*.

📦 Consider the following Java method.

```
void findRate(int input) {
```

```
    if (input == 0) {
        return 0;
    }
    cap = 100/input;
    if (cap < 0) {
        return -1;
    } else {
        return cap;
    }
}
```

It has 3 paths, as follows:

1. enter -> `2` -> `3` -> exit (can be triggered by input `0`)
2. enter -> `2` -> `5` -> `6` -> `7` -> exit (can be triggered by input `-5`)
3. enter -> `2` -> `5` -> `6` -> `9` -> exit (can be triggered by input `8`)

So, to achieve 100% path coverage, we need at least 3 test cases (e.g., `0`, `-5`, `8`).

A loop can increase the path count greatly.

```
void sayHello(List<String> names) {
    for (String n : names) {
        System.out.println(n);
    }
}
```

The number of paths through this method is very large, as each possible length of `names` produces a unique path.

1. enter -> `2` -> exit (if `names` is empty)
2. enter -> `2` -> `3` -> exit (if `names` has one entry)
3. enter -> `2` -> `3` -> `2` -> `3` -> exit (if `names` has two entries) 1 ...

So, achieving 100% path coverage of this method will be extremely difficult.

- **Entry/exit coverage** measures coverage in terms of possible *calls to* and *exits* from the operations in the SUT.
  Entry points refer to all places from which the method is called from the rest of the code i.e., all places where the control is handed over to the method in concern.
  Exit points refer to points at which the control is returned to the caller e.g., return statements, throwing of exceptions.

How ⚡⚡⚡

Measuring coverage is often done using *coverage analysis tools*. Most IDEs have inbuilt support for measuring test coverage, or at least have plugins that can measure test coverage.

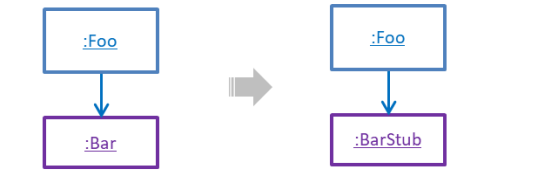
Coverage analysis can be useful in improving the quality of testing e.g., if a set of test cases does not achieve 100% branch coverage, more test cases can be added to cover missed branches.

Dependency injection

What ⚡⚡⚡

***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 from its dependencies so that it can be tested in isolation.

A `Foo` object normally depends on a `Bar` object, but you can inject a `BarStub` object so that the `Foo` object no longer depends on a `Bar` object. Now you can test the `Foo` object in isolation from the `Bar` object.



Normal dependency After dependency injection

TDD

What ⚡⚡⚡

***Test-Driven Development(TDD)* advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments**. In TDD you first define the precise behavior of the SUT using test code, and then update the SUT to match the specified behavior. While TDD has its fair share of detractors, there are many who consider it a good way to reduce defects. One big advantage of TDD is that it guarantees the code is testable.

Test case design

Introduction

What ⚡⚡⚡

Except for trivial SUTs, exhaustive testing is not practical because such testing often requires a massive/infinite number of test cases.

📦 Consider the test cases for adding a string object to a collection:

- Add an item to an empty collection.
- Add an item when there is one item in the collection.
- Add an item when there are 2, 3, ... n items in the collection.
- Add an item that has an English, a French, a Spanish, ... word.
- Add an item that is the same as an existing item.
- Add an item immediately after adding another item.
- Add an item immediately after system startup.
- ...

Exhaustive testing of this operation can take many more test cases.

> Program testing can be used to show the presence of bugs, but never to show their absence! --Edsger Dijkstra

Every test case adds to the cost of testing. In some systems, a single test case can cost thousands of dollars e.g. on-field testing of flight-control software. Therefore, **test cases need to be designed to make the best use of testing resources.** In particular:

- **Testing should be *effective*** i.e., it finds a high percentage of existing bugs e.g., a set of test cases that finds 60 defects is more effective than a set that finds only 30 defects in the same system.

- **Testing should be *efficient*** i.e., it has a high rate of success (bugs found/test cases) a set of 20 test cases that finds 8 defects is more efficient than another set of 40 test cases that finds the same 8 defects.

For testing to be E&E, each new test you add should be targeting a potential fault that is not already targeted by existing test cases. There are test case design techniques that can help us improve the E&E of testing.

Positive vs Negative Test Cases ⚡⚡

A *positive test case* is when the test is designed to produce an expected/valid behavior. On the other hand, a *negative test case* is designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message.

📦 Consider the testing of the method `print(Integer i)` which prints the value of `i`.

- A positive test case: `i == new Integer(50);`
- A negative test case: `i == null;`

Black Box vs Glass Box ⚡⚡

Test case design can be of three types, based on how much of the SUT's internal details are considered when designing test cases:

- ***Black-box* (aka *specification-based or responsibility-based*) approach**: test cases are designed exclusively based on the SUT's specified external behavior.

- ***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.

- ***Gray-box* approach**: test case design uses *some* important information about the implementation. For example, if the implementation of a sort operation uses different algorithms to sort lists shorter than 1000 items and lists longer than 1000 items, more meaningful test cases can then be added to verify the correctness of both algorithms.

Black-box and white-box testing

Equivalence partitions

What ⚡⚡

Consider the testing of the following operation.

`isValidMonth(m)` : returns `true` if `m` (an `int`) is in the range [1..12]

It is inefficient and impractical to test this method for all integer values `[-MIN_INT to MAX_INT]`. Fortunately, there is no need to test all possible input values. For example, if the input value `233` fails to produce the correct result, the input `234` is likely to fail too; there is no need to test both.

In general, **most SUTs do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways.** That means a range of inputs is treated the same way inside the SUT. ***Equivalence partitioning (EP)* is a test case design technique that uses the above observation to improve the E&E of testing.**

Equivalence partition (aka equivalence class): A group of test inputs that are likely to be processed by the SUT in the same way.

By dividing possible inputs into equivalence partitions you can,

- **avoid testing too many inputs from one partition.** Testing too many inputs from the same partition is unlikely to find new bugs. This increases the efficiency of testing by reducing redundant test cases.
- **ensure all partitions are tested.** Missing partitions can result in bugs going unnoticed. This increases the effectiveness of testing by increasing the chance of finding bugs.

Basic ⚡⚡

Equivalence partitions (EPs) are usually derived from the specifications of the SUT.

These could be EPs for the isValidMonth example:

- [MIN_INT … 0]: **below** the range that produces `true` (produces `false`)
- [1 … 12]: the range that produces `true`
- [13 … MAX_INT]: **above** the range that produces `true` (produces `false`)

When the SUT has multiple inputs, you should identify EPs for each input.

Consider the method `duplicate(String s, int n): String` which returns a `String` that contains `s` repeated `n` times.

Example EPs for `s`:

- zero-length strings
- string containing whitespaces
- …

Example EPs for `n`:

- `0`
- negative values
- …

An EP may not have adjacent values.

📦 Consider the method `isPrime(int i): boolean` that returns true if `i` is a prime number.

EPs for `i`:

- prime numbers
- non-prime numbers

Some inputs have only a small number of possible values and a potentially unique behavior for each value. In those cases, you have to consider each value as a partition by itself.

📦 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.

Note that the EP technique is merely a heuristic and not an exact science, especially when applied manually (as opposed to using an automated program analysis tool to derive EPs). The partitions derived depend on how one 'speculates' the SUT to behave internally. Applying EP under a glass-box or gray-box approach can yield more precise partitions.

📦 Consider the EPs given above for the method `isValidMonth`. A different tester might use these EPs instead:

- [1 … 12]: the range that produces `true`
- [all other integers]: the range that produces `false`

📦 Some more examples:

Specification	Equivalence partitions
`isValidFlag(String s): boolean` Returns `true` if `s` is one of [`"F"`, `"T"`, `"D"`]. The comparison is case-sensitive.	[`"F"`] [`"T"`] [`"D"`] [`"f"`, `"t"`, `"d"`] [any other string][null]
`squareRoot(String s): int` Pre-conditions: `s` is a `String` that represents a positive integer e.g., `"23"`. Returns the square root of `s` if the square root is an integer; returns `0` otherwise.	[`s` does not represent a valid number] [`s` is a negative integer] [`s` has an integer square root] [`s` does not have an integer square root]

Intermediate ⚡⚡⚡

When deciding EPs of OOP methods, you need to identify the EPs of all data participants that can potentially influence the behaviour of the method, such as,

- the target object of the method call
- input parameters of the method call
- other data/objects accessed by the method such as global variables. This category may not be applicable if using the black box approach (because the test case designer using the black box approach will not know how the method is implemented).

📦 Consider this method in the `DataStack` class: `push(Object o): boolean`

- Adds `o` to the top of the stack if the stack is not full.
- Returns `true` if the push operation was a success.
- Throws
  - `MutabilityException` if the global flag `FREEZE==true`.
  - `InvalidValueException` if `o` is null.

EPs:

- `DataStack` object: [full] [not full]
- `o` : [null] [not null]
- `FREEZE` : [true][false]

Consider a simple Minesweeper app. What are the EPs for the `newGame()` method of the `Logic` component?

As `newGame()` does not have any parameters, the only obvious participant is the `Logic` object itself.

Note that if the glass-box or the grey-box approach is used, other associated objects that are involved in the method might also be included as participants. For example, the `Minefield` object can be considered as another participant of the `newGame()` method. Here, the black-box approach is assumed.

Next, let us identify equivalence partitions for each participant. Will the `newGame()` method behave differently for different `Logic` objects? If yes, how will it differ? In this case, yes, it might behave differently based on the game state. Therefore, the equivalence partitions are:

- `PRE_GAME` : before the game starts, minefield does not exist yet
- `READY` : a new minefield has been created and the app is waiting for the player's first move
- `IN_PLAY` : the current minefield is already in use
- `WON` , `LOST` : let us assume that `newGame()` behaves the same way for these two values

Consider the `Logic` component of the Minesweeper application. What are the EPs for the `markCellAt(int x, int y)` method? The partitions in **bold** represent valid inputs.

- `Logic` : PRE_GAME, **READY**, **IN_PLAY**, WON, LOST
- `x` : [MIN_INT..-1] **[0..(W-1)]** [W..MAX_INT] (assuming a minefield size of WxH)
- `y` : [MIN_INT..-1] **[0..(H-1)]** [H..MAX_INT]
- `Cell` at `(x,y)` : **HIDDEN**, MARKED, CLEARED

Boundary value analysis

What ⚡⚡

***Boundary Value Analysis (BVA)* is a test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.** This is not surprising, as the end points of boundaries are often used in branching instructions, etc., where the programmer can make mistakes.

The `markCellAt(int x, int y)` operation could contain code such as `if (x > 0 && x <= (W-1))` which involves the boundaries of x's equivalence partitions.

BVA suggests that when picking test inputs from an equivalence partition, values near boundaries (i.e., boundary values) are more likely to find bugs.

Boundary values are sometimes called *corner cases*.

How ⚡⚡⚡

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. The number of values to pick depends on other factors, such as the cost of each test case.

Some examples:

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 No specific boundary No specific boundary
[non-empty Stack] (assuming a fixed size stack)	Stack with: no elements, **one element**, two elements, **no empty spaces**, only one empty space

Combining test inputs

Why ⚡⚡

An SUT can take multiple inputs. You can select values for each input (using equivalence partitioning, boundary value analysis, or some other technique).

📦 An SUT that takes multiple inputs and some values chosen for each input:

- Method to test: `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test:

Input	Valid values to test	Invalid values to test
participation	0, 1, 19, 20	21, 22
projectGrade	A, B, C, D, F	
isAbsent	true, false	
examScore	0, 1, 69, 70,	71, 72

Testing all possible combinations is effective but not efficient. If you test all possible combinations for the above example, you need to test 6x5x2x6=360 cases. Doing so has a higher chance of discovering bugs (i.e., effective) but the number of test cases will be too high (i.e., not efficient). Therefore, **you need smarter ways to combine test inputs that are both effective and efficient.**

Test Input Combination Strategies ⚡⚡

Given below are some basic strategies for generating a set of test cases by combining multiple test inputs.

📦 Let's assume the SUT has the following three inputs and you have selected the given values for testing:

SUT: `foo(char p1, int p2, boolean p3)`

Values to test:

Input	Values
p1	a, b, c
p2	1, 2, 3

Input	Values
p3	T, F

The *all combinations* strategy generates test cases for each unique combination of test inputs.

📦 This strategy generates 3x3x2=18 test cases.

Test Case	p1	p2	p3
1	a	1	T
2	a	1	F
3	a	2	T
...	...	...	...
18	c	3	F

The *at least once* strategy includes each test input at least once.

📦 This strategy generates 3 test cases.

Test Case	p1	p2	p3
1	a	1	T
2	b	2	F
3	c	3	VV/IV

VV/IV = Any Valid Value / Any Invalid Value

The *all pairs* strategy creates test cases so that for any given pair of inputs, all combinations between them are tested. It is based on the observation that a bug is rarely the result of more than two interacting factors. The resulting number of test cases is lower than the *all combinations* strategy, but higher than the *at least once* approach.

📦 This strategy generates 9 test cases:

▶ See steps

Test Case	p1	p2	p3
1	a	1	T
2	a	2	T
3	a	3	F
4	b	1	F

Test Case	p1	p2	p3
5	b	2	T
6	b	3	F
7	c	1	T
8	c	2	F
9	c	3	T

A variation of this strategy is to test all pairs of inputs but only for inputs that could influence each other.

📦 Testing all pairs between p1 and p3 only while ensuring all p2 values are tested at least once:

Test Case	p1	p2	p3
1	a	1	T
2	a	2	F
3	b	3	T
4	b	VV/IV	F
5	c	VV/IV	T
6	c	VV/IV	F

The *random* strategy generates test cases using one of the other strategies and then picks a subset randomly (presumably because the original set of test cases is too big).

There are other strategies that can be used too.

Heuristic: Each Valid Input at Least Once in a Positive Test Case ⚡⚡⚡

Consider the following scenario.

SUT: `printLabel(String fruitName, int unitPrice)`

Selected values for `fruitName` (invalid values are underlined):

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

Selected values for `unitPrice` :

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

Suppose these are the test cases being considered.

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

It looks like the test cases were created using the *at least once* strategy. After running these tests, can you confirm that the square-format label printing is done correctly?

- Answer: No.
- Reason: `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.

In this case, a useful heuristic to apply is **each valid input must appear at least once in a positive test case**. `Cherry` is a valid test input and you must ensure that it appears at least once in a positive test case. Here are the updated test cases after applying that heuristic.

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	Dog	-1	Error message “invalid fruit"

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

Heuristic: Test Invalid Inputs Individually Before Combining Them ⚡⚡⚡

To verify the SUT is handling a certain invalid input correctly, it is better to test that invalid input without combining it with other invalid inputs. For example, consider the test case 4 of test cases designed in [Heuristic: each valid input at least once in a positive test case]. After running that test case, can you be sure that the error message “invalid fruit” is caused by the invalid fruitName `Dog` ?

- Answer: No
- Reason: Because it could have been (incorrectly) triggered by the other invalid unitPrice of `-1` in that test case, due to a bug in the code.

Therefore, if that test case was intended to verify that the invalid fruitName `Dog` triggers the "invalid fruit" error message, it is better not to include the invalid unitPrice `-1` in that test case at the same time. If the invalid value `-1` needs to be tested, we should test it in a separate test case.

After applying the above insight to our running example, you get the following test cases.

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

This is not to say *never* have more than one invalid input in a test case. In fact, an SUT might work correctly when only one invalid input is given but not when a certain combination of multiple invalid inputs is given. Hence, it is still useful to have test cases with multiple invalid inputs, *after* you already have confirmed that the SUT works when only one invalid input is given.

Test invalid inputs individually before combining them is the heuristic we learned here. As a test case with multiple invalid inputs *by itself* does not confirm that the SUT works for each of those invalid inputs, you are better off testing the SUT with one-invalid-input-at-a-time first, and if you can afford more test cases, also testing with combinations of invalid inputs.

Mix ⚡⚡⚡

Consider the calculateGrade scenario given below:

- SUT: `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test: invalid values are underlined
  - participation: 0, 1, 19, 20, 21, 22
  - projectGrade: A, B, C, D, F
  - isAbsent: true, false
  - examScore: 0, 1, 69, 70, 71, 72

To get the first cut of test cases, let's apply the *at least once* strategy.

Test cases for calculateGrade V1

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV/IV	69	...
4	20	D	VV/IV	70	...
5	21	F	VV/IV	71	Err Msg

Case No.	participation	projectGrade	isAbsent	examScore	Expected
6	22	VV/IV	VV/IV	72	Err Msg

VV/IV = Any Valid or Invalid Value, Err Msg = Error Message

Next, let's apply the *each valid input at least once in a positive test case* heuristic. Test case 5 has a valid value for `projectGrade=F` that doesn't appear in any other positive test case. Let's replace test case 5 with 5.1 and 5.2 to rectify that.

Test cases for calculateGrade V2

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV/IV	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

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

Next, you have to apply the *no more than one invalid input in a test case* heuristic. Test cases 5.2 and 6 don't follow that heuristic. Let's rectify the situation as follows:

Test cases for calculateGrade V3

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV	VV	VV	Err Msg
5.3	22	VV	VV	VV	Err Msg
6.1	VV	VV	VV	71	Err Msg
6.2	VV	VV	VV	72	Err Msg

Next, you can assume that there is a dependency between the inputs `examScore` and `isAbsent` such that an absent student can only have `examScore=0`. To cater for the hidden invalid case arising from this, you can add a new test case where `isAbsent=true`

and `examScore!=0`. In addition, test cases 3-6.2 should have `isAbsent=false` so that the input remains valid.

Test cases for calculateGrade V4

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	false	69	...
4	20	D	false	70	...
5.1	VV	F	false	VV	...
5.2	21	VV	false	VV	Err Msg
5.3	22	VV	false	VV	Err Msg
6.1	VV	VV	false	71	Err Msg
6.2	VV	VV	false	72	Err Msg
7	VV	VV	true	!=0	Err Msg

More

Testing Based on Use Cases ⚡⚡⚡

Use cases can be used for system testing and acceptance testing. For example, the main success scenario can be one test case while each variation (due to extensions) can form another test case. However, note that use cases do not specify the exact data entered into the system. Instead, it might say something like `user enters his personal data into the system`. Therefore, the tester has to choose data by considering equivalence partitions and boundary values. The combinations of these could result in one use case producing many test cases.

To increase the E&E of testing, high-priority use cases are given more attention. For example, a scripted approach can be used to test high-priority test cases, while an exploratory approach is used to test other areas of concern that could emerge during testing.

SECTION: PROJECT MANAGEMENT

Revision Control (Using Git & GitHub)

Introduction to Revision Control

Given below is a general introduction to revision control, adapted from bryan-mercurial-guide:

> **Revision control is the process of managing multiple versions of a piece of information.** In its simplest form, this is something that many people do by hand: every time you modify a file, save it under a new name that contains a number, each one higher than the number of the preceding version.
>
> Manually managing multiple versions of even a single file is an error-prone task, though, so software tools to help automate this process have long been available. The earliest automated revision control tools were intended to help a single user to manage revisions of a single file. Over the past few decades, the scope of revision control tools has expanded greatly; they now manage multiple files, and help multiple people to work together. The best modern revision control tools have no problem coping with thousands of people working together on projects that consist of hundreds of thousands of files.
>
> There are a number of reasons why you or your team might want to use an automated revision control tool for a project.
>
> - **It will track the history and evolution of your project**, so you don't have to. For every change, you'll have a log of who made it; why they made it; when they made it; and what the change was.
> - **It makes it easier for you to collaborate** when you're working with other people. For example, when people more or less simultaneously make potentially incompatible changes, the software will help you to identify and resolve those conflicts.
> - **It can help you to recover from mistakes**. If you make a change that later turns out to be an error, you can revert to an earlier version of one or more files. In fact, a good revision control tool will even help you to efficiently figure out exactly when a problem was introduced.
> - **It will help you to work simultaneously on, and manage the drift between, multiple versions of your project.**
>
> Most of these reasons are equally valid, at least in theory, whether you're working on a project by yourself, or with a hundred other people.

A revision is the state of a piece of information at a specific point in time, resulting from changes made to it e.g., if you modify the code and save the file, you have a new *revision* (or a new *version*) of that file. Some seem to use this term interchangeably with *version* while others seem to distinguish the two -- here, let us treat them as the same, for simplicity.
Revision Control Software (RCS) are the software tools that automate the process of Revision Control i.e., managing revisions of software artifacts. RCS are also known as *Version Control Software (VCS)*, and by a few other names.

Git is the most widely used RCS today. Other RCS tools include Mercurial, Subversion (SVN), Perforce, CVS (Concurrent Versions System), Bazaar, TFS (Team Foundation Server), and Clearcase.

Github is a web-based project hosting platform for projects using Git for revision control. Other similar services include GitLab, BitBucket, and SourceForge.

Putting a Folder Under Git's Control

Normally, we use Git to manage a revision history of a specific folder, which gives us the ability to revision-control any file in that folder and its subfolders.

To put a folder under the control of Git, we *initialise* a repository (short name: repo) in that folder. This way, we can initialise repos in different folders, to revision-control different clusters of files independently of each other e.g., files belonging to different projects.

You can follow the hands-on practical below to learn how to initialise a repo in a folder.

Initialising a repo results in two things:

a) **First, Git now recognises this folder as a Git repository**, which means it can now help you track the version history of files inside this folder.

b) **Second, Git created a hidden subfolder named** `.git` inside the `things` folder. This folder will be used by Git to store metadata about this repository.

A Git-controlled folder is divided into two main parts:

1. **The repository** – stored in the hidden `.git` subfolder, which contains all the metadata and history.
2. **The working directory** – everything else in that folder, where you create and edit files.

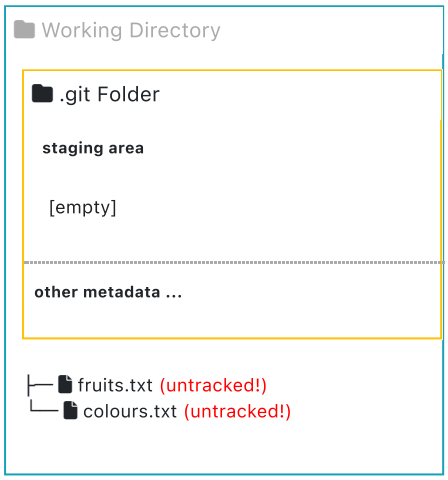
Specifying What to include in a Snapshot

Git considers new files that you add to the working directory as 'untracked' i.e., Git is aware of them, but they are not yet under Git's control. The same applies to files that existed in the working folder at the time you initialised the repo.

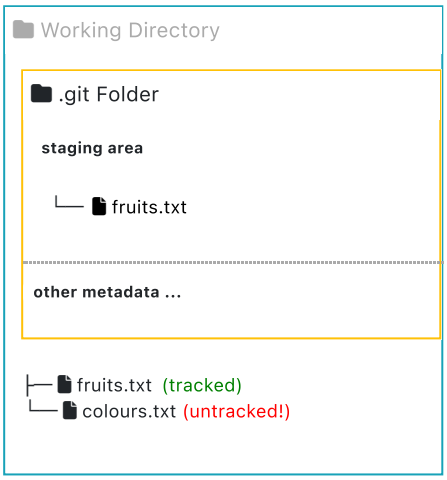
A Git repo has an internal space called the staging area which it uses to build the next snapshot. Another name for the staging area is the **index** .

We can stage an untracked file to tell Git that we want its current version to be included in the next snapshot (in Git terminology, such a snapshot is called a **commit**). Once you stage an untracked file, it becomes **'tracked'** (i.e., under Git's control). A staged file can be **unstaged** to indicate that we no longer want it to be included in the next snapshot.

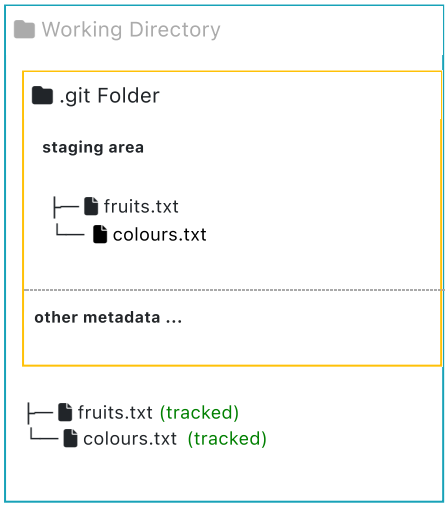
In the example below, you can see how staging files change the status of the repo as you go from (a) to (c).



(a) State of the repo, just after initialisation, and creating two files. Both are untracked.

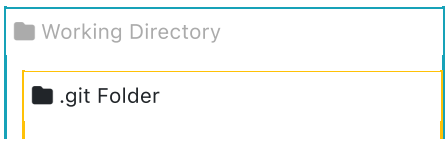


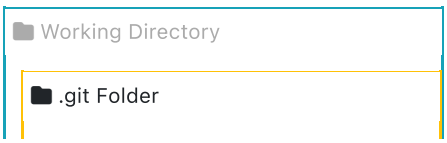
(b) State after staging `fruits.txt` .



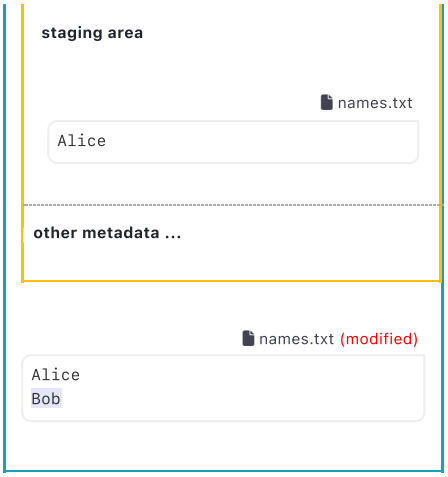
(c) State after staging `colours.txt` .

If you modify a staged file, it goes into the 'modified' state i.e., the file contains modifications that are not present in the copy that is waiting (in the staging area) to be included in the next snapshot. If you wish to include these new changes in the next snapshot, you need to stage the file again, which will overwrite the copy of the file that was previously in the staging area.
The example below shows how the status of a file changes when it is modified *after* it was staged.

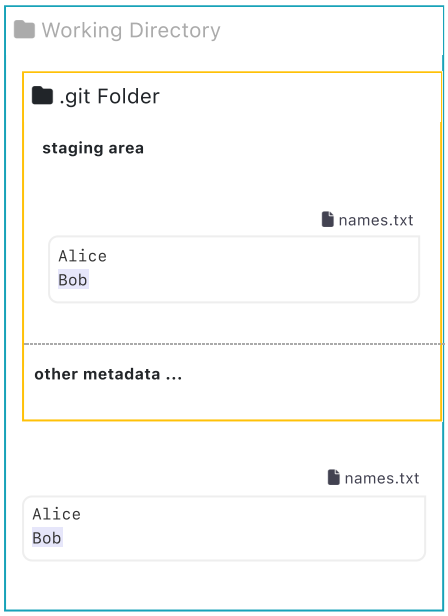




(a) The file names.txt is staged. The copy in the staging area is an exact match to the one in the working directory.



(b) State after adding a line to the file. Git indicates it as 'modified' because it now differs from the version in the staged area.



(c) After staging the file again, the staging area is updated with the latest copy of the file, and it is no longer marked as 'modified'.

Staging applies regardless of whether a file is currently tracked.

- Staging an untracked file will both begin tracking the file and include it in the next snapshot.
- **Staging an already tracked file will simply mark its current changes for inclusion in the next commit.**

Git does not track empty folders. It tracks only folders that contain tracked files.

PRO-TIP: Applying a Git command to multiple files in one go

When a Git command expects a list of files or paths as a parameter (as the `git add` command does), these parameters are known as **pathspecs — patterns that tell Git which files or directories to operate on.** Pathspecs can be simple file names, directory names, or more complex patterns.

Here are some common ways to write them, shown with examples using the `git add <pathspec>` command:

- **Specify multiple files, separated by spaces:**

```
git add f1.txt f2.txt data/lists/f3.txt  # stages the specified three files
```

- **Use a glob pattern:**

```
git add '*.txt'  # stages all .txt files in the current directory
```

Quoting the glob pattern is recommended so your shell doesn't expand it before Git sees it.

- **Use `.` to indicate 'all in the current directory and subdirectories':**

```
git add .  # stages all files in current directory and its subdirectories
```

- **Specific directory, to indicate 'this directory and its subdirectories':**

```
git add path/to/dir  # stages all files in path/to/dir and its subdirectories
```

- **Negated pathspecs, to indicate 'except these':**

```
git add . ':!*.log'  # stage everything except .log files
```

Git supports combining these features — for example, you could add all `.txt` files except those in a certain folder using:

```
git add '*.txt' ':!docs/*.txt'
```

Saving a Snapshot

Saving a snapshot is called committing and a saved snapshot is called a commit.

A Git commit is a full snapshot of your working directory based on the files you have staged, more precisely, a record of the exact state of all files in the staging area (index) at that moment -- even the files that have not changed since the last commit. This is in contrast to other revision control software that only store the delta in a commit. Consequently, a Git commit has all the information it needs to recreate the snapshot of the working directory at the time the commit was created.
A commit also includes metadata such as the author, date, and an optional **commit message** describing the change.

A Git commit is a snapshot of *all* tracked files, not simply a delta of *what changed since the last commit*.

Examining the Revision History

Git commits form a timeline, as each corresponds to a point in time when you asked Git to take a snapshot of your working directory. Each commit links to at least one previous commit, forming a structure that we can traverse.
A timeline of commits is called a branch. By default, Git names the initial branch `master` -- though many now use `main` instead. You'll learn more about branches in future lessons. For now, just be aware that the commits you create in a new repo will be on a branch called `master` (or `main`) by default.

Git can show you the list of commits in the Git history.

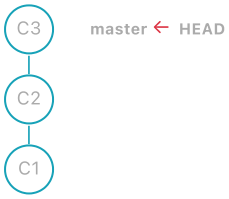
The Git data model consists of two types of entities: objects and refs (short for *references*). In this lesson, you will encounter examples of both.

A Git revision graph is a visualisation of a repo's revision history, consisting of one or more branches. First, let us learn to work with simpler revision graphs consisting of one branch, such as the one given below.

- **Nodes in the revision graph represent commits. A commit is one of four main types of Git objects**. For completeness, the other three are:
  - **blob** (short for **binary large object**): stores the contents of a file
  - **tree**: represents a directory and records the hierarchy of its contents by referencing blobs and other trees
  - **tag** (specifically, *annotated* tag): a label-like object that can store additional metadata and point to a specific commit
- **A commit is identified by its SHA value**. A SHA (Secure Hash Algorithm) value is a unique identifier generated by Git to represent each commit. It is produced by using SHA-1 (i.e., one of the algorithms in the SHA family of cryptographic hash functions) on the entire content of the commit. It's a 40-character hexadecimal string (e.g., `f761ea63738a67258628e9e54095b88ea67d95e2`) that acts like a fingerprint, ensuring that every commit can be referenced unambiguously. That is, **every commit has a unique SHA-1 hash value**.
- A commit is a full snapshot of the working directory, constructed based on the previous commit, and the changes staged. That means **each commit (except the initial commit) is based on a another 'parent' commit**. Some commits can have multiple parent commits -- we'll cover that later.

Edges in the revision graph represent links between a commit and its parent commit(s). In some revision graph visualisations, you might see arrows (instead of lines) showing how each commit points to its parent commit.

Git uses refs to name and keep track of various points in a repository's history. These refs are essentially 'named-pointers' that can serve as bookmarks to reach a certain point in the revision graph using the ref name.



In the revision graph above, there are two refs `master` and `← HEAD`.

- `master` is a branch ref. **A branch ref points to the latest commit on a branch**. In this visualisation, the commit shown alongside the ref is the one it points to i.e., `C3`.
  When you create a new commit, the branch ref of the current branch moves to the new commit.
- `← HEAD` **is a special ref that typically points to the current branch and moves along with that branch ref.** In this example, it is pointing to the `master` branch.
  In certain cases, the `HEAD` may point directly to a specific commit instead of a branch. This situation is called a "detached `HEAD`", which will be covered in a later lesson.

Remote Repositories

A repo you have on your computer is called a local repo. A remote repo is a repo hosted on a remote computer and allows remote access. Some use cases for remote repositories:

a) as a backup of your local repo
b) as an intermediary repo to work on the same files from multiple computers
c) for sharing the revision history of a codebase among team members of a multi-person project

It is possible to set up a Git remote repo on your own server, but an easier option is to use a remote repo hosting service such as GitHub.

Creating a Repo on GitHub

You can create a remote repository based on an existing local repository, to serve as a remote copy of your local repo. For example, suppose you created a local repo and worked with it for a while, but now you want to upload it onto GitHub. The first step is to create an empty repository on GitHub.

Linking a Local Repo With a Remote Repo

A Git `remote` is a reference to a repository hosted elsewhere, usually on a server like GitHub, GitLab, or Bitbucket. It allows your local Git repo to communicate with another remote copy — for example, to upload locally-created commits that are missing in the remote copy.

By *adding a remote*, you are informing the local repo details of a remote repo it can communicate with, for example, where the repo exists and what name to use to refer to the remote.

The URL you use to connect to a remote repo depends on the protocol — HTTPS or SSH:

- **HTTPS URLs use the standard web protocol** and start with `https://github.com/` (for GitHub users). e.g.,
  ```
  https://github.com/username/repo-name.git
  ```
- **SSH URLs use the secure shell protocol** and start with `git@github.com:` . e.g.,
  ```
  git@github.com:username/repo-name.git
  ```

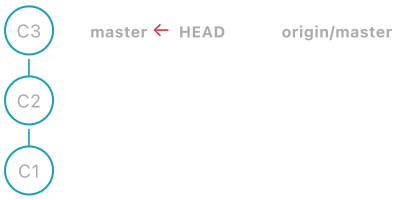
A Git repo can have multiple remotes. You simply need to specify different names for each remote (e.g., `upstream` , `central` , `production` , `other-backup` ...).

Updating the Remote Repo

You can `push` content of one repository to another, usually from your local repo to a remote repo. Pushing transfers recorded Git history (such as past commits), but it does not transfer unstaged changes or untracked files.

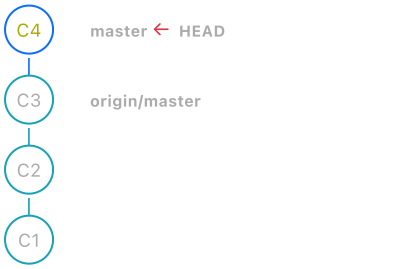
- **To push, you need to have write-access to the remote repo**.
- **Pushing is performed one branch at a time**; you must specify which branch you want to push.

You can configure Git to `track` a pairing between a local branch and a remote branch, so in future you can push from the same local branch to the corresponding remote branch without needing to specify them again. For example, you can set your local `master` branch to *track* the `master` branch on the remote repo `origin` i.e., local `master` branch will track the upstream branch `origin/master` .



In the revision graph above, you see a new type of ref (`origin/master`). This is a **`remote-tracking branch` ref that represents the state of a corresponding branch in a remote repository** (if you previously set up the branch to 'track' a remote branch). In this example, the `master` branch in the remote `origin` is also at the commit `C3` (which means you have not created new commits after you pushed to the remote).

If you now create a new commit `C4` , the state of the revision graph will be as follows:



Explanation: When you create `C4` , the current branch `master` moves to `C4` , and `HEAD` moves along with it. However, the `master` branch in the remote `origin` remains at `C3` (because you have not pushed `C4` yet). That is, the remote-tracking branch `origin/master` is *one commit* ***behind*** the local branch `master` (or, the local branch is *one commit* ***ahead***). The `origin/master` ref will move to `C4` only after you push your local branch to the remote again.

The push command can be used repeatedly to send further updates to another repo e.g., to update the remote with commits you created since you pushed the first time.

Note that you can push between two repos only if those repos have a shared history among them (i.e., one should have been created by copying the other).

Omitting Files from Revision Control

You can specify which files Git should `ignore` from revision control. While you can always omit files from revision control simply by not staging them, having an 'ignore-list' is more convenient, especially if there are files inside the working folder that are not suitable for revision control (e.g., temporary log files) or files you want to prevent from accidentally including in a commit (files containing confidential information).

A repo-specific ignore-list of files can be specified in a `.gitignore` file, stored in the root of the repo folder.

The `.gitignore` file itself can be either revision controlled or ignored.

- To version control it (the more common choice – which allows you to track how the `.gitignore` file changes over time), simply commit it as you would commit any other file.
- To ignore it, simply add its name to the `.gitignore` file itself.

The `.gitignore` file supports file patterns e.g., adding `temp/*.tmp` to the `.gitignore` file prevents Git from tracking any `.tmp` files in the `temp` directory.

Files recommended to be omitted from version control

- **Binary files** *generated* when building your project e.g., `*.class`, `*.jar`, `*.exe`
  Reasons:
  1. no need to version control these files as they can be generated again from the source code
  2. Revision control systems are optimized for tracking text-based files, not binary files.
- **Temporary files** e.g., log files generated while testing the product
- **Local files** i.e., files specific to your own computer e.g., local settings of your IDE (`.idea/`)
- **Sensitive content** i.e., files containing sensitive/personal information e.g., credential files, personal identification data (especially if there is a possibility of those files getting leaked via the revision control system).

Duplicating a Remote Repo on the Cloud

A `fork` is a copy of a remote repository created on the same hosting service such as GitHub, GitLab, or Bitbucket. On GitHub, you can fork a repository from another user or organisation into your own space (i.e., your user account or an organisation you have sufficient access to). **Forking** is particularly useful if you want to experiment with a repo but don't have write permissions to the original -- you can fork it and work on your own remote copy without affecting the original repository.

Forking is not a Git feature, but a feature provided by hosted Git services like GitHub, GitLab, or Bitbucket.

GitHub does not allow you to fork the same repo more than once to the same destination. If you want to re-fork, you need to delete the previous fork.

Creating a Local Copy of a Repo

You can `clone` a repository to create a full copy of it on your computer. This copy includes the entire revision history, branches, and files of the original, so it behaves just like the original repository. For example, you can clone a repository from a hosting service like GitHub to your computer, giving you a complete local version to work with.

Cloning a repo automatically creates a remote named `origin` which points to the repo you cloned from.

The repo you cloned from is often referred to as the `upstream` repo.

Downloading Data Into a Local Repo

There are two steps to bringing over changes from a remote repository into a local repository: *fetch* and *merge*.

- **`Fetch` is the act of downloading the latest changes from the remote repository, but without applying them to your current branch yet.** It updates metadata in your repo so that it knows what has changed in the remote repo, but your own local branch remains untouched.
- **`Merge` is what you do after fetching, to actually incorporate the fetched changes into your local branch.** It combines your local branch with the changes from the corresponding branch from the remote repo.

`Pull` is a shortcut that combines fetch and merge — it fetches the latest changes from the remote and immediately merges them into your current branch. In practice, Git users typically use the pull instead of the fetch-then-merge.

> ℹ **pull = fetch + merge**

You can pull from any number of remote repos, provided the repos involved have a shared history. This can be useful when the upstream repo you forked from has some new commits that you wish to bring over to your copies of the repo (i.e., your fork and your local repo).

Examining a Commit

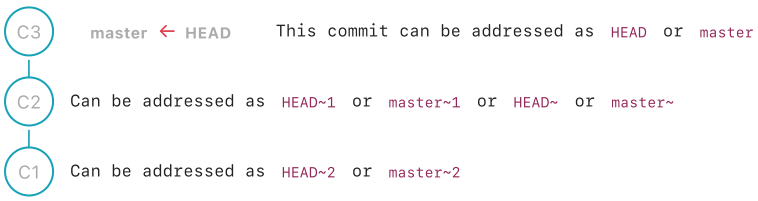
When you examine a commit, normally what you see is the 'changes made since the previous commit'. This does not mean that a Git commit contains only the changes made since the previous commit. As you recall, a Git commit contains a full snapshot of the working directory. However, tools used to examine commits typically show only the changes, as that is the more informative part.

Git shows changes included in a commit by *dynamically calculating* the difference between the snapshots stored in the target commit and the parent commit. This is because Git commits store snapshots of the working directory, not changes themselves.

Although each commit represents a copy of the entire working directory, Git uses space efficiently in two main ways:

1. **Reuse of unchanged data:** If a file hasn't changed since a previous commit, the commit simply points to the already stored version of that file instead of making another copy. This means only new or changed files take up extra space, while unchanged files are reused.
2. **Compression:** Git also compresses all the files and data it stores using an algorithm (zlib). So, even the objects that are stored (whether reused or new) take up less disk space because they are saved in a compressed format.

To address a specific commit, you can use its SHA (e.g., `e60deaeb2964bf2ebc907b7416efc890c9d4914b`). In fact, just the first few characters of the SHA is enough to uniquely address a commit (e.g., `e60deae`), provided the partial SHA is long enough to uniquely identify the commit (i.e., only one commit has that partial SHA).
Naturally, a commit can be addressed using any ref pointing to it too (e.g., `HEAD`, `master`).
Another related technique is to use the `<ref>~<n>` notation (e.g., `HEAD~1`) to address the commit that is `n` commits prior to the commit pointed by `<ref>` i.e., "start with the commit pointed by `<ref>` and go back `n` commits".
A related alternative notation is `HEAD^`, `HEAD^^`, `HEAD^^^`, ... to mean `HEAD~1`, `HEAD~2`, `HEAD~3` etc.



Git uses the `diff` format to show file changes in a commit. The diff format was originally developed for Unix. It was later extended with headers and metadata to show changes between file versions and commits. Here is an example diff showing the changes to a file.

```
diff --git a/fruits.txt b/fruits.txt
index 7d0a594..f84d1c9 100644
--- a/fruits.txt
+++ b/fruits.txt
@@ -1,6 +1,6 @@
-apples
+apples, apricots
 bananas
 cherries
 dragon fruits
-elderberries
 figs
@@ -20,2 +20,3 @@
 oranges
+pears
 raisins
diff --git a/colours.txt b/colours.txt
new file mode 100644
index 0000000..55c8449
--- /dev/null
+++ b/colours.txt
@@ -0,0 +1 @@
+a file for colours
```

A Git diff can consist of multiple **`file diffs`**, one for each changed file. Each file diff can contain one or more **`hunk`** i.e., a localised group of changes within the file — including lines added, removed, or left unchanged (included for context).

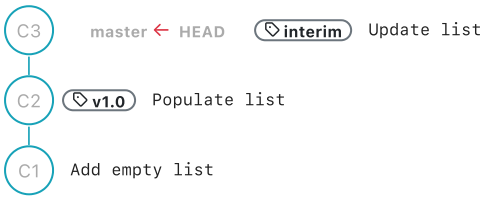
PRO-TIP: Use Git Aliases to Work Faster

The Git `alias` feature allows you to create custom shortcuts for frequently used Git commands. This saves time and reduces typing, especially for long or complex commands. Once an alias is defined, you can use the alias just like any other Git command e.g., use `git lodg` as an alias for `git log --oneline --decorate --graph`.

You can also create shell-level aliases using your shell configuration (e.g., `.bashrc`, `.zshrc`) to make even shorter aliases. This lets you create shortcuts for any command, including Git commands, and even combine them with other tools. e.g., instead of the Git alias `git lodg`, you can define a shorter shell-level alias `glodg`.

Tagging Commits

Git lets you `tag` commits with names, making them easy to reference later. This is useful when you want to mark specific commits -- such as releases or key milestones (e.g., `v1.0` or `v2.1`). Using tags to refer to commits is much more convenient than using SHA hashes. In the diagram below, (v1.0) and (interim) are tags.



A tag stays fixed to the commit. Unlike branch refs or `HEAD`, tags do not move automatically as new commits are made. As you see below, after adding a new commit, tags stay in the previous commits while `master ← HEAD` has moved to the new commit.

Git supports two kinds of tags:

1. **A `lightweight tag`** is just a ref that points directly to a commit, like a branch that doesn't move.
2. **An `annotated tag`** is a full Git object that stores a reference to a commit along with metadata such as the tagger's name, date, and a message.

Annotated tags are generally preferred for versioning and public releases, while lightweight tags are often used for less formal purposes, such as marking a commit for your own reference.

If you need to change what a tag points to, you must delete the old one and create a new tag with the same name. This is because tags are designed to be fixed references to a specific commit, and there is no built-in mechanism to 'move' a tag.

! **Tags are different from commit messages**, in purpose and in form. A commit message is a description of the commit that is *part of* the commit itself. A tag is a short name for a commit, which you can use to address a commit.

Pushing commits to a remote does not push tags automatically. You need to push tags specifically.

Comparing Points of History

Git's `diff` feature can show you what changed between two points in the revision history. Given below are some use cases.

Usage 1: Examining changes in the working directory

Example use case: To verify the next commit will include exactly what you intend it to include.

Usage 2: Comparing two commits at different points of the revision graph

Example use case: Suppose you're trying to improve the performance of a piece of software by experimenting with different code

tweaks. You commit after each change (as you should). After several commits, you now want to review the overall effect of all those changes on the code.

Usage 3: Examining changes to a specific file

Example use case: Similar to other use cases but when you are interested in a specific file only.

Traversing to a Specific Commit

Suppose you added a new feature to a software product, and while testing it, you noticed that another feature added two commits ago doesn't handle a certain edge case correctly. Now you're wondering: *did the new feature break the old one, or was it already broken?* Can you go back to the moment you committed the old feature and test it in isolation, and come back to the present after you found the answer? With Git, you can.

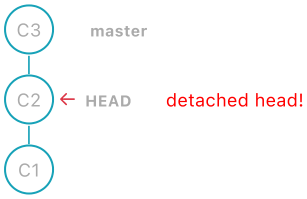
To view the working directory at a specific point in history, you can `check out` a commit created at that point.

When you check out a commit, Git:

1. **Updates your working directory to match the snapshot in that commit**, overwriting current files as needed.
2. **Moves the `HEAD` ref to that commit**, marking it as the current state you're viewing.

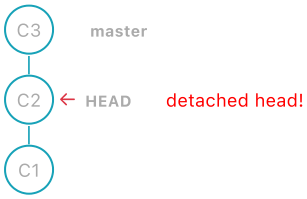
→

[check out commit `C2` ...]



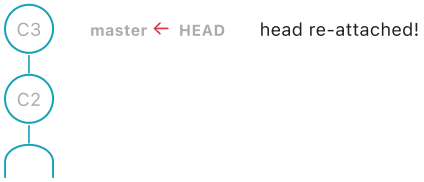
Checking out a specific commit puts you in a `"detached HEAD"` state: i.e., the `HEAD` no longer points to a branch, but directly to a commit (see the above diagram for an example). This isn't a problem by itself, but any commits you make in this state *can* be lost, unless certain follow-up actions are taken. It is perfectly fine to be in a detached state if you are only examining the state of the working directory at that commit.

To get out of a "detached HEAD" state, you can simply check out a branch, which "re-attaches" `HEAD` to the branch you checked out.



→

[check out `master` ...]



If there are uncommitted changes in the working directory, Git proceeds with a checkout *only* if it can preserve those changes.

- Example 1: There is a new file in the working directory that is not committed yet.
  → Git will proceed with the checkout and will keep the uncommitted file as well.
- Example 2: There is an uncommitted change to a file that conflicts with the version of that file in the commit you wish to check out.
  → Git will abort the checkout, and the repo will remain in the current commit.

The Git `stash` feature temporarily sets aside uncommitted changes you've made (in your working directory and staging area), without committing them. This is useful when you're in the middle of some work, but need to switch to another state (e.g., checkout a previous commit), and your current changes are not yet ready to be committed or discarded. You can later reapply the stashed changes when you're ready to resume that work.

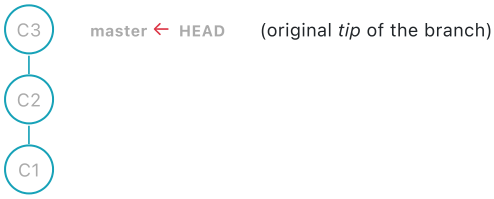
Rewriting History to Start Over

Suppose you realise your last few commits have gone in the wrong direction, and you want to go back to an earlier commit and continue from there — as if the "bad" commits never happened. Git's *reset* feature can help you do that.

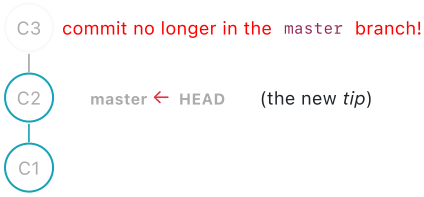
Git `reset` moves the `tip` of the current branch to a specific commit, optionally adjusting your staged and unstaged changes to match. **This effectively rewrites the branch's history** by discarding any commits that came after that point.

Resetting is different from the *checkout* feature:

- Reset: Lets you start over from a past state. It rewrites history by moving the branch ref to a new location.
- Checkout: Lets you explore a past state *without* rewriting history. It just moves the `HEAD` ref.



→
[reset to `C2` ...]



There are three types of resets: soft, mixed, hard. All three move the branch pointer to a new commit, but they vary based on what happens to the staging area and the working directory.

- **`soft reset`** : Moves the cumulative changes from the discarded commits into the staging area, waiting to be committed again. Any staged and unstaged changes that existed before the reset will remain untouched.
- **`mixed reset`** : Cumulative changes from the discarded commits, and any existing staged changes, are moved into the working directory.
- **`hard reset`** : All staged and unstaged changes are discarded. Both the working directory and the staging area are aligned with the target commit (as if no changes were done after that commit).

Rewriting history can cause your local repo to `diverge` from its remote counterpart. For example, if you discard earlier commits and create new ones in their place, and you've already pushed the original commits to a remote repository, your local branch history will no longer match the corresponding remote branch. Git refers to this as a diverged history.

To protect the integrity of the remote, Git will reject attempts to push a diverged branch using a normal push. If you want **to *overwrite* the remote history with your local version, you must perform a `force push`** .

Controlling What Goes Into a Commit

Crafting a commit involves two aspects:

1. **What changes to include in it**: deciding what changes belong together in a single commit — this is about *commit granularity*, ensuring each commit represents a meaningful, self-contained change.
2. **How to include those changes**: carefully staging just those changes — this is about using Git's tools to precisely control what ends up in the commit.

SIDEBAR: Guidelines on what to include in a commit

A good commit represents a single, logical unit of change — something that can be described clearly in one sentence. For example, fixing a specific bug, adding a specific feature, or refactoring a specific function. If each commit tells a clear story about why the change was made and what it achieves, your repository history becomes a valuable narrative of the project's development. Here are some (non-exhaustive) guidelines:

- **No more than one change per commit**: Avoid lumping unrelated changes into one commit, as this makes the history harder to understand, review, or revert (if each commit contains one standalone change, to reverse that change can be done by deleting or reverting that specific commit entirely, without affecting any other changes).
- **Make the commit standalone**: Don't split a single logical change across multiple commits unnecessarily, as this can clutter the history and make it harder to follow the evolution of an idea or feature.
- **Small enough to review easily, but large enough to stand on its own**: For example, fixing the same typo in five files can be one commit — splitting it into five separate commits is excessive. Conversely, implementing a big feature may be too much for one commit — instead, break it down into a series of commits, each containing a meaningful yet standalone step towards the final goal.

Git can let you choose not just which files, but which specific changes within those files, to include in a commit. Most Git tools — including the command line and many GUIs — let you interactively select which "hunks" or even individual lines of a file to stage. This allows you to separate unrelated changes and avoid committing unnecessary edits. If you make multiple changes in the same file, you can selectively stage only the parts that belong to the current logical change.

This level of control is particularly useful when:

- You noticed and fixed a small, unrelated issue while working on something else.
- You experimented with multiple approaches in the same file and now want to commit only the final, clean solution.
- You want your commit history to clearly separate concerns, even when the edits touch the same files.

Writing Good Commit Messages ⚡⚡⚡

Every commit you make in Git also includes a `commit message` that explains the change. While one-line messages are fine for small or obvious changes, as your revision history grows, good commit messages become an important source of information — for example, to understand the rationale behind a specific change made in the past.

A commit message is meant to explain the intent behind the changes, not just what was changed. The code (or diff) already shows what changed. Well-written commit messages make collaboration, code reviews, debugging, and future maintenance easier by helping you and others quickly understand the project's history without digging into the code of every commit.

A complete commit message can include a short summary line (the `subject`) followed by a more detailed `body` if needed. The subject line should be a concise description of the change, while the body can elaborate on the context, rationale, side effects, or other details if the change is more complex.

A commit message has the following structure (note how the subject and the body are separated by a blank line):

```
Subject line
<blank line>
Body

# lines starting with '#' are ignored (they will not be included in the commit message)
```

Here is an example commit message:

```
Find command: make matching case-insensitive

Find command is case-sensitive.

A case-insensitive find is more user-friendly because users cannot be
expected to remember the exact case of the keywords.

Let's,
```

```
* update the search algorithm to use case-insensitive matching
* add a script to migrate stress tests to the new format
```

Following a style guide makes your commit messages more consistent and fit-for-purpose. Many teams adopt established guidelines. These style guides typically contain common conventions that Git users follow when writing commit messages. For example:

- Keep the subject line (the first line) under 50–72 characters.
- Write the subject in the imperative mood (e.g., `Fix typo in README` rather than `Fixed typo` or `Fixes typo`).
- Leave a blank line between the subject and the body, if you include a body.
- Wrap the body at around 72 characters per line for readability.

PRO-TIP: Configure Git to use your preferred text editor

Git will use the default text editor when it needs you to write a commit message. However, Git can be configured to use a different text editor of your choice.

Reorganising Commits

Git has a powerful tool called interactive rebasing which lets you review and reorganise your recent commits. With it, you can reword commit messages, change their order, delete commits, combine several commits into one (squash), or split a commit into smaller pieces. This feature is useful for tidying up a commit history that has become messy — for example, when some commits are out of order, poorly described, or include changes that would be clearer if split up or combined.

! **Rebasing rewrites history.** It is not recommended to rebase commits you have already shared with others.

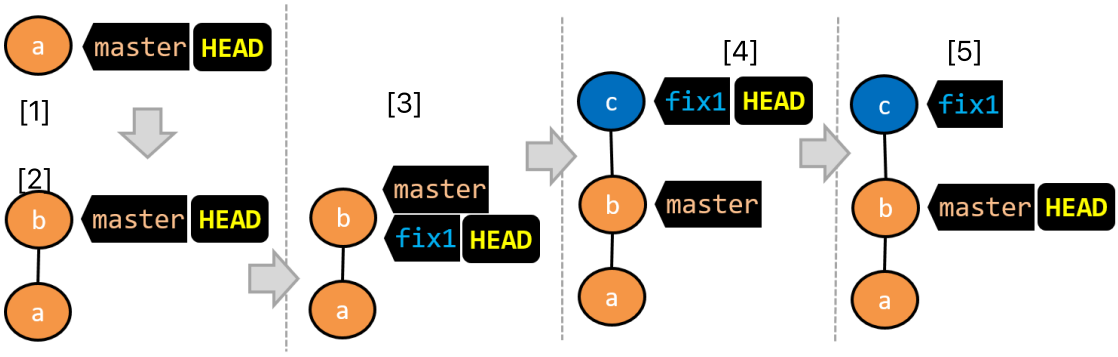
Creating Branches

Git branches let you develop multiple versions of your work in parallel — effectively creating diverged timelines of your repository's history. For example, one team member can create a new branch to experiment with a change, while the rest of the team continues working on another branch. Branches can have meaningful names, such as `master`, `release`, or `draft`.

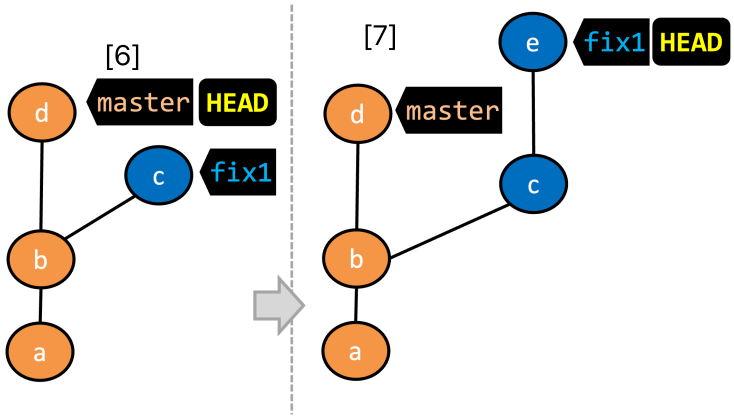
A Git branch is simply a ref (a named label) that points to a commit and automatically moves forward as you add new commits to that branch. As you've seen before, the `HEAD` ref indicates which branch you're currently working on, by pointing to the corresponding branch ref.
When you add a commit, it goes into the branch you are currently on, and the branch ref (together with the `HEAD` ref) moves to the new commit.

Git creates a branch named `master` by default (Git can be configured to use a different name e.g., `main`).

Given below is an illustration of how branch refs move as branches evolve. Refer to the text below it for explanations of each stage.



[1] There is only one branch (i.e., `master`) and there is only one commit on it. The `HEAD` ref is pointing to the `master` branch (as we are currently on that branch).
[2] A new commit has been added. The `master` and the `HEAD` refs have moved to the new commit.
[3] A new branch `fix1` has been added. The repo has switched to the new branch too (hence, the `HEAD` ref is attached to the `fix1` branch).
[4] A new commit (`c`) has been added. The current branch ref `fix1` moves to the new commit, together with the `HEAD` ref.
[5] The repo has switched back to the `master` branch. Hence, the `HEAD` has moved back to `master` branch's tip.
At this point, the repo's working directory reflects the code at commit `b` (not `c`).



[6] A new commit (`d`) has been added. The `master` and the `HEAD` refs have moved to that commit.
[7] The repo has switched back to the `fix1` branch and added a new commit `e` to it. Note how the branch ref `fix1` (together with `HEAD`) has moved to the new commit `e` while the branch ref `master` still points to `d` .

! Note that appearance of the revision graph (colors, positioning, orientation etc.) varies based on the Git client you use, and might not match the exact diagrams given above.

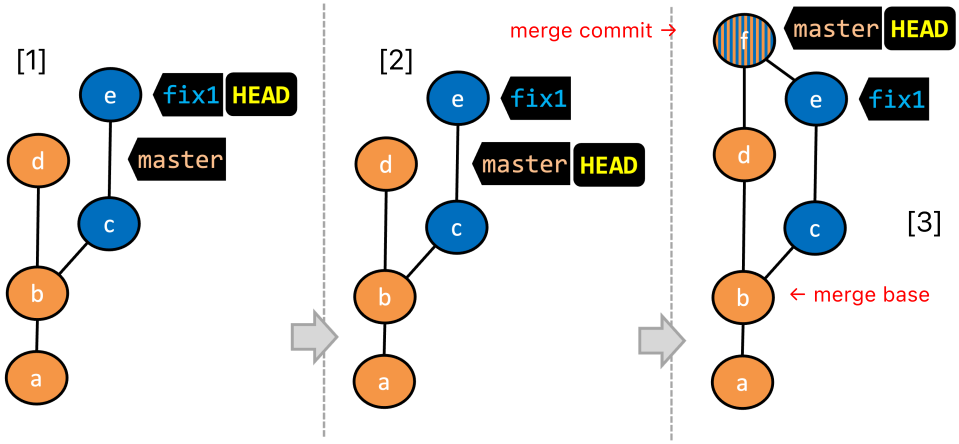
You can also start a branch from an earlier commit, instead of the latest commit in the current branch. For that, simply check out the commit you wish to start from.

Merging Branches

Merging combines the changes from one branch into another, bringing their diverged timelines back together.

When you merge, Git looks at the two branches and figures out how their histories have diverged since their **merge base** (i.e., the most recent common ancestor commit of two branches). It then applies the changes from the other branch onto your current branch, creating a new commit. **The new commit created when merging is called a merge commit — it records the result of combining both sets of changes.**

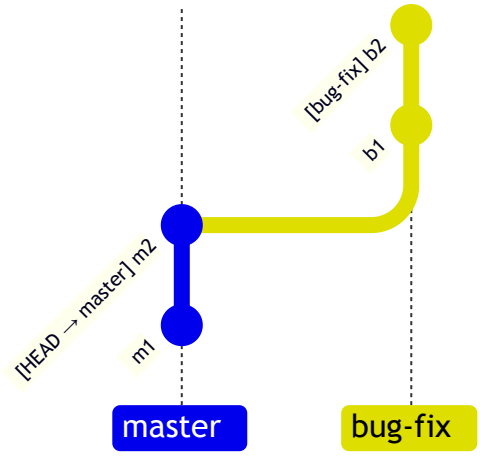
Given below is an illustration of how such a merge looks like in the revision graph:



[1] We are on the `fix1` branch (as indicated by `HEAD`).
[2] We have switched to the `master` branch (thus, `HEAD` is now pointing to `master` ref).
[3] The `fix1` branch has been merged into the `master` branch, creating a *merge commit* `f` . The repo is still on the `master` branch.

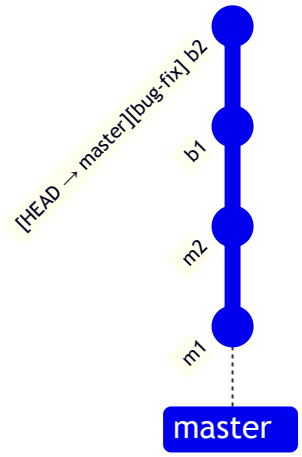
A merge commit has two parent commits e.g., in the above example, the merge commit `f` has both `d` and `e` as parent commits. **The parent commit on the receiving branch is considered the first parent and the other is considered the second parent** e.g., in the example above, `fix1` branch is being merged into the `master` branch (i.e., the receiving branch) -- accordingly, `d` is the first parent and `e` is the second parent.

When the branch you're merging into hasn't diverged — meaning it hasn't had any new commits since the merge base — **Git simply moves the branch pointer forward to include all the new commits**, keeping the history clean and linear. This is **called a fast-forward merge** because Git simply "fast-forwards" the branch pointer to the tip of the other branch. The result looks as if all the changes had been made directly on one branch, without any branching at all.



→

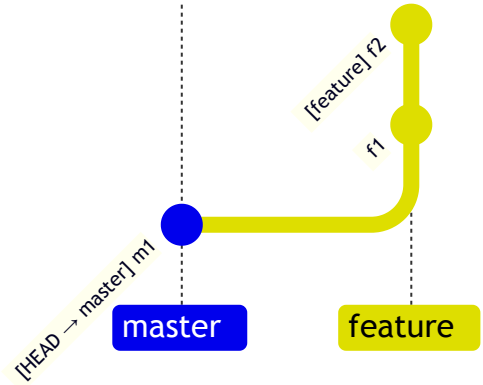
[merge `bug-fix`]



In the example above, the `master` branch has not changed since the merge base (i.e., `m2`). Hence, merging the branch `bug-fix` onto `master` can be done by fast-forwarding the `master` branch ref to the tip of the `bug-fix` branch (i.e., `b2`).

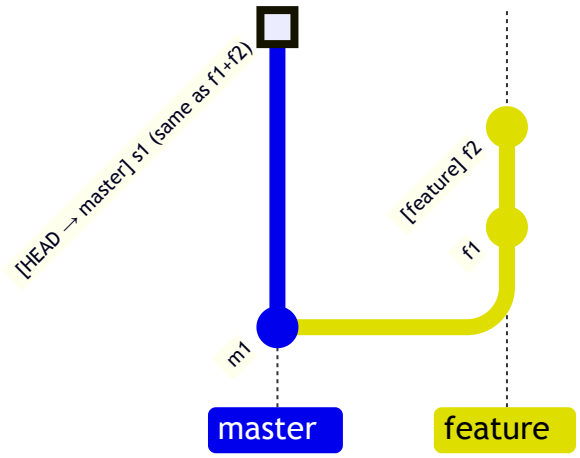
It is possible to force Git to create a merge commit even if fast forwarding is possible. This is useful if you prefer the revision graph to visually show when each branch was merged to the main timeline.

A squash merge combines all the changes from a branch into a single commit on the receiving branch, without preserving the full commit history of the branch being merged. This is especially useful when the feature branch contains many small or experimental commits that would clutter the main branch's history. By squashing, you retain the final state of the changes while presenting them as one cohesive unit, making the project history easier to read and manage. It also helps maintain a linear, simplified commit log on the main branch.



→

[squash merge...]



In the example above, the branch `feature` has been squash merged onto the `master` branch, creating a single 'squashed' commit `s1` that combines all the commits in `feature` branch.

Resolving Merge Conflicts

A merge conflict happens when Git can't automatically combine changes from two branches because the same parts of a file were modified differently in each branch. When this happens, **Git pauses the merge and marks the conflicting sections in the affected files so you can resolve them yourself.** Once you've reviewed and fixed the conflicts, you can tell Git they're resolved and complete the merge.

More generally, a conflict occurs when Git cannot automatically reconcile different changes made to the same part of a file -- branch merge conflicts is just one example.

Renaming Branches

Local branches can be renamed easily. Renaming a branch simply changes the branch reference (i.e., the name used to identify the branch) — it is just a cosmetic change.

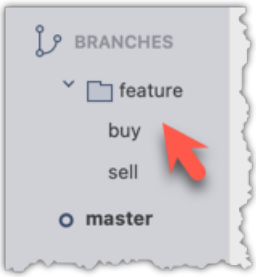
SIDEBAR: Branch naming conventions

Branch names can contain lowercase letters, numbers, `/`, dashes (`-`), underscores (`_`), and dots (`.`). You can use uppercase letters too, but many teams avoid them for consistency.

A common branch naming convention is to prefix it with `<category>/`. Some examples:

- `feature/login-form` — for new features (`origin/feature/login-form` could be the matching remote-tracking branch)
- `bugfix/profile-photo` — for fixing bugs
- `hotfix/payment-crash` — for urgent production fixes
- `release/2.0` — for prepping a release
- `experiment/ai-chatbot` — for "just trying stuff"

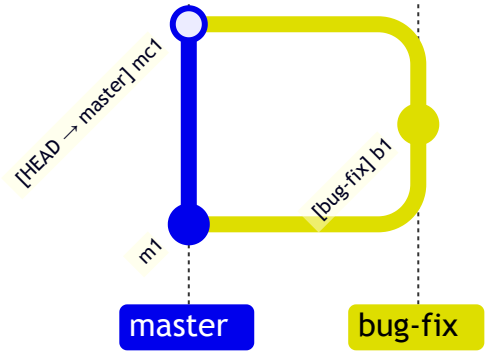
Although forward-slash (`/`) in the prefix doesn't mean folders, some tools treat it kind of like a path so you can group related branches when you run git branch. Shown below is an example of how Sourcetree groups branches with the same prefix.



Deleting Branches

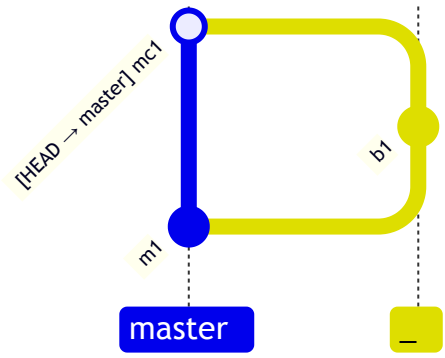
Deleting a branch deletes the corresponding branch ref from the revision history (it does not delete any commits). The impact of the loss of the branch ref depends on whether the branch has been merged.

When you delete a branch that has been merged, the commits of the branch will still exist in the history and will be safe. Only the branch ref is lost.



→

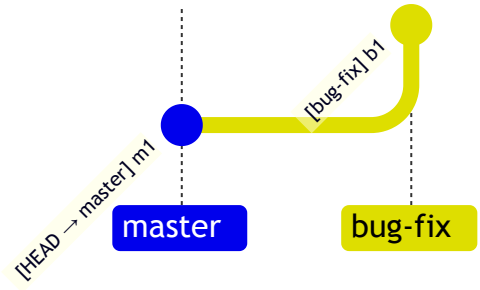
[delete branch `bug-fix`]



In the above example, the only impact of the deletion is the loss of the branch ref `bug-fix` . All commits remain reachable (via the `master` branch), and there is no other impact on the revision history.

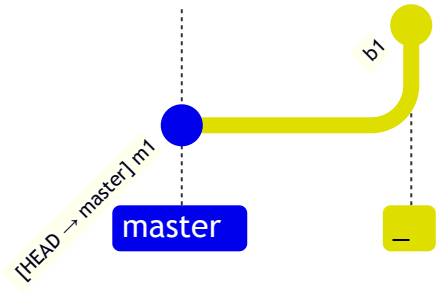
In fact, some prefer to delete the branch soon after merging it, to reduce branch references cluttering up the revision history.

When you delete a branch that has not been merged, the loss of the branch ref can render some commits unreachable (unless you know their commit IDs or they are reachable through other refs), putting them at risk of being lost eventually.



→

[delete branch `bug-fix`]



In the above example, the commit `b1` is no longer reachable, unless we know its commit ID (i.e., the `SHA`).

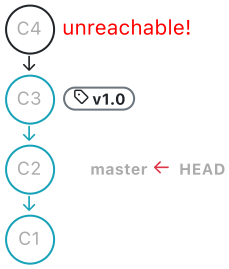
SIDEBAR: What makes a commit 'unreachable'?

Recall that a commit only has a pointer to its parent commit (not its descendent commits).

A commit is considered `reachable` if you can get to it by starting at a branch, tag, or other ref and walking backward through its parent commits. This is the normal state for commits — they are part of the visible history of a branch or tag.

When no branch, tag, or ref points to a commit (directly or indirectly), it becomes `unreachable`. This often happens when you delete a branch or rewrite history (e.g., with reset or rebase), leaving some commits **"orphaned"** (or **"dangling"**) without a ref pointing to them.

In the example below, `C4` is unreachable (i.e., cannot be reached by starting at any of the three refs: v1.0 or master or ← HEAD), but the other three are all reachable.

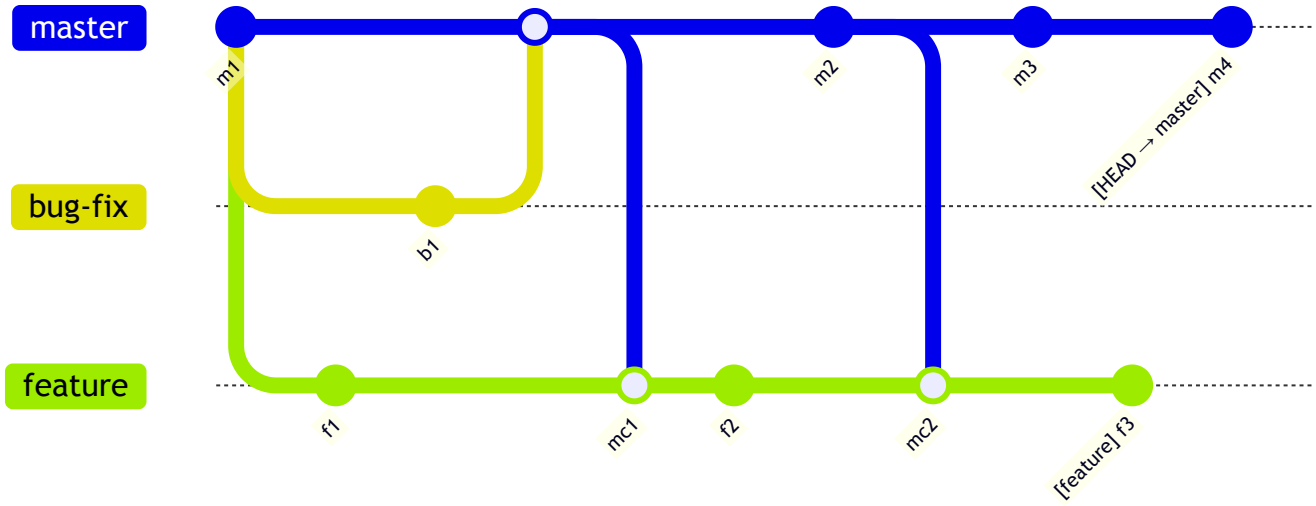


Unreachable commits are not deleted immediately — Git keeps them for a while before cleaning them up. By default, Git retains unreachable commits for at least **30 days**, during which they can still be recovered if you know their SHA. **After that, they will be garbage-collected, and will be lost for good.**

Merging to Sync Branches

When working in parallel branches, you'll often need to `sync` (short for synchronise) one branch with another. For example, while developing a feature in one branch, you might want to bring in a recent bug fix from another branch that your branch doesn't yet have.

The simplest way to sync branches is to merge — that is, to sync a branch `b1` with changes from another branch `b2`, you merge `b2` into `b1`. In fact, you can merge them periodically to keep one branch up to date with the other.



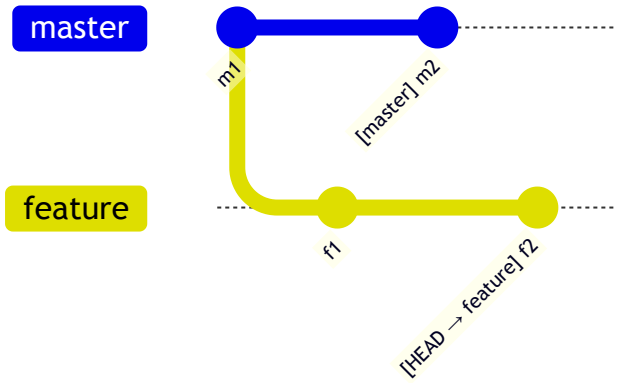
In the example above, you can see how the `feature` branch is merging the `master` branch periodically to keep itself in sync with the changes being introduced to the `master` branch.

Rebasing to Sync Branches ⚡⚡⚡

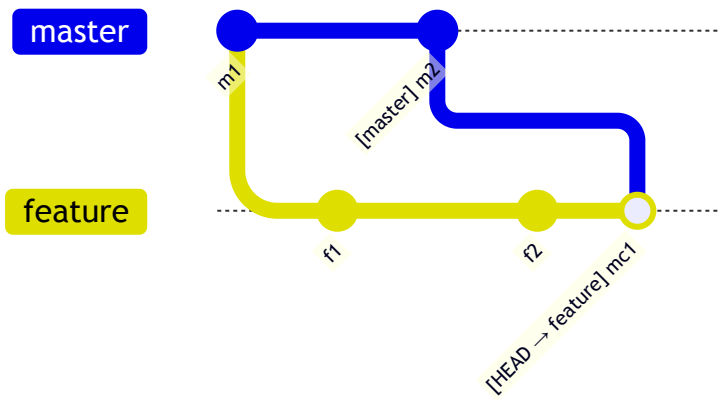
Rebasing is another way to synchronise one branch with another, while keeping the history cleaner and more linear. Instead of creating a merge commit to combine the branches, rebasing moves the entire sequence of commits from your branch and "replays" them on top of another branch. This effectively **moves the base of your branch to the tip of the other branch** (i.e., it 're-bases' it — hence the name), as if you had started your work from there in the first place.

Rebasing is especially useful when you want to update your branch with the latest changes from a main branch, but you prefer an uncluttered history with fewer merge commits.

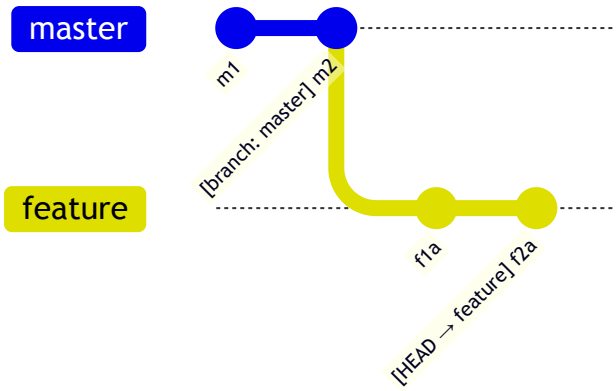
Suppose we have the following revision graph, and we want to sync the `feature` branch with `master`, so that changes in commit `m2` become visible to the `feature` branch.



If we merge the `master` branch to the `feature` branch as given below, `m2` becomes visible to the `feature` branch. However, it creates a merge commit.



Instead of merging, if we *rebased* the `feature` branch on the `master` branch, we would get the following.



Note how the rebasing changed the base of the `feature` branch from `m1` to `m2`. As a result, changes done in `m2` are now visible to the `feature` branch. But there is no merge commit, and the revision graph is simpler.

Also note how the first commit in the feature branch, previously shown as `f1`, is now shown as `f1a` after the rebase. Although both commits contain the same changes, other details -- such as the parent commit -- are different, making them two distinct Git objects with different SHA values. Similarly, `f2` and `f2a` are also different. Thus, the history of the entire `feature` branch has changed after the rebase.

Because rebasing rewrites the commit history of your branch, you should avoid rebasing branches that you've already published, and are potentially used by others -- rewriting published history can cause confusion and conflicts for those using the previous version of the commits.

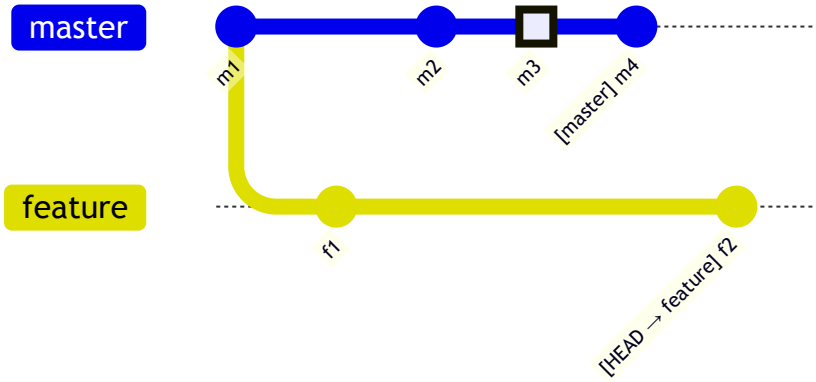
Copying Specific Commits ⚡⚡⚡

Cherry-picking is another way to synchronise branches, by applying specific commits from one branch onto another.

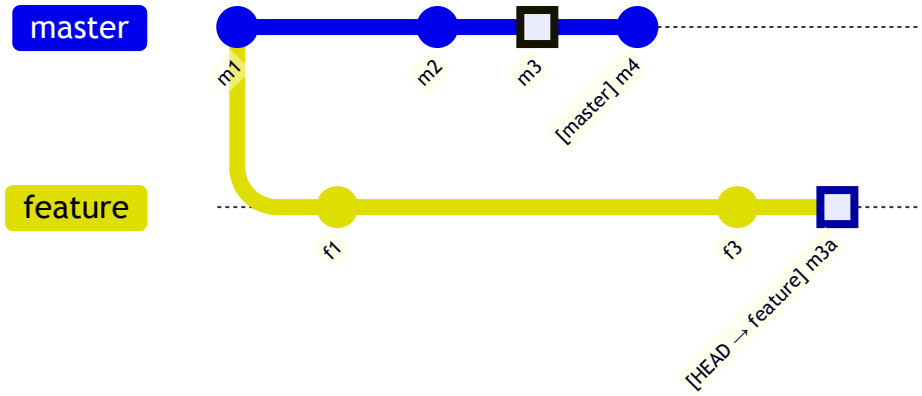
Unlike merging or rebasing — which bring over all changes since the branches diverged — cherry-picking lets you choose individual commits and apply just those, one at a time, to your current branch. This is useful when you want to bring over a bug fix or a small feature from another branch without merging the entire branch history.

Because cherry-picking copies only the chosen commits, **it creates new commits on your branch with the same changes but different SHA values**.

Suppose we have the following revision graph, and we want to bring over the changes introduced in `m3` (in the `master` branch) onto the `feature` branch.



After cherry-picking `m3` onto the `feature` branch, the revision graph should look like the following:



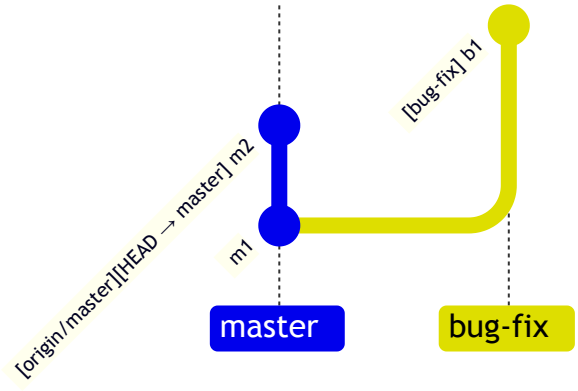
Note how it makes the changes done in `m3` available (from now on) in the `feature` branch, with minimal changes to the revision graph. Also note that the new commit `m3a` contains the same changes as `m3`, but it will be a different Git object with a different SHA value.

Cherry-picking is another Git operation that can result in conflicts i.e., if the changes in the cherry-picked commit conflict with the changes in the receiving branch.

Pushing Branches to a Remote

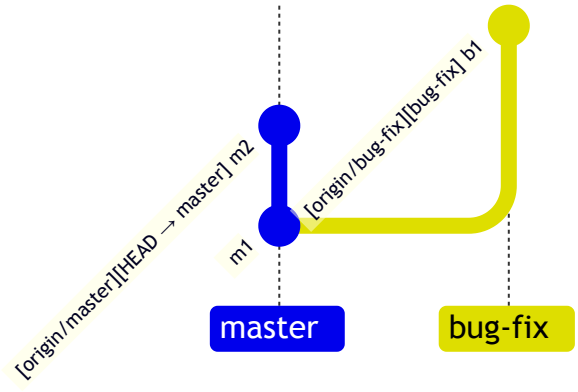
Pushing a copy of local branches to the corresponding remote repo makes those branches available remotely.

In a previous lesson, we saw how to push the default branch to a remote repository and have Git set up tracking between the local and remote branches using a remote-tracking reference. **Pushing any other local branch to a remote works the same way as pushing the default branch — you simply specify the target branch instead of the default branch.** Pushing any new commits in any local branch to a corresponding remote branch is done similarly as well.



[`bug-fix` branch does not exist in the remote `origin`]

→



[after pushing `bug-fix` branch to origin,
and setting up a remote-tracking branch]

Pulling Branches from a Remote

Sometimes we need to create a local copy of a branch from a remote repository, make further changes to it, and keep it synchronised with the remote branch. Let's explore how to handle this in a few common use cases:

Use case 1: Working with branches that already existed in the remote repo when you cloned it to your computer.

When you clone a repository,

1. **Git checks out the default branch.** You can start working on this branch immediately. This branch is tracking the default branch in the remote, which means you can easily synchronise changes in this branch with the remote by pulling and pushing.
2. **Git also *fetches* all the other branches from the remote.** These other branches are not immediately available as local branches, but they are visible as remote-tracking branches.
   You can think of remote-tracking branches as read-only references to the state of those branches in the remote repository at the time of cloning. They allow you to see what work has been done on those branches without yet making local copies of them.
   To work on one of these branches, you can create a new local branch based on the remote-tracking branch. Once you do this, your local branch will usually be configured to track the corresponding branch on the remote, so you can easily synchronise your work later.

Use case 2: Working with branches that were added to the remote repository after you cloned it e.g., a branch someone else pushed to the remote after you cloned.

Simply fetch to update your local repository with information about the new branch. After that, you can create a local copy of it and work with it just as you did in Use Case 1.

Fetching was covered in Lesson T3L3. Downloading Data Into a Local Repo.

Deleting Branches from a Remote

To delete a branch in a remote repository, you simply tell Git to remove the reference to that branch from the remote. This does not delete the branch from your local repository — it only removes it from the remote, so others won't see it anymore. This is useful for cleaning up clutter in the remote repo e.g., delete old or merged branches that are no longer needed on the remote.

Renaming Branches in a Remote

Git does not have a way to rename remote branches in place. Instead, you create a new branch with the desired name and delete the old one. This involves renaming your local branch to the new name, pushing it to the remote (which effectively creates a new remote branch), and then removing the old branch from the remote. This ensures the remote reflects the updated name while preserving the commit history and any work already done on the branch.

While Git cannot rename a remote branch in place, **GitHub allows you to rename a branch in a remote repo.** If you use this approach, the local repo still needs to be updated to reflect the change.

Creating PRs

A pull request (PR for short) is a mechanism for contributing code to a remote repo i.e., "I'm *requesting* you to *pull* my proposed changes to your repo". It's feature provided by RCS platforms such as GitHub. For this to work, the two repos must have a shared history. The most common case is sending PRs from a fork to its upstream repo.

Suppose you want to propose some changes to a GitHub repo (e.g., samplerepo-pr-practice) as a pull request (PR).

The next step of the PR lifecycle is the PR *review*. The members of the repo that received your PR can now review your proposed changes.

- If they like the changes, they can *merge* the changes to their repo, which also closes the PR automatically.
- If they don't like it at all, they can simply close the PR too i.e., they reject your proposed change.
- In most cases, they will add comments to the PR to suggest further changes. When that happens, GitHub will notify you.

You can update the PR along the way too. Suppose PR reviewers suggested a certain improvement to your proposed code. To update your PR as per the suggestion, you can simply modify the code in your local repo, commit the updated code to the same branch as before, and push to your fork as you did earlier. The PR will auto-update accordingly.

Sending PRs using the `master` branch is less common than sending PRs using separate branches. For example, suppose you wanted to propose two bug fixes that are not related to each other. In that case, it is more appropriate to send two separate PRs so that each fix can be reviewed, refined, and merged independently. But if you send PRs using the `master` branch only, both fixes (and any other change you do in the `master` branch) will appear in the PRs you create from it.

It is possible to create PRs within the same repo too e.g., you can create a PR from branch `feature-x` to the `master` branch, within the same repo. Doing so will allow the code to be reviewed by other developers (using PR review mechanism) before it is merged.

Reviewing PRs ⚡⚡

PR reviews are a collaborative process in which project members examine and provide feedback on PRs submitted to a remote repo. After an initial review, the reviewer may suggest improvements or identify issues, prompting the submitter to refine and update their code in the PR. This review-refine-update cycle can repeat several times, with reviewers reassessing each new iteration until all feedback is addressed and the code meets the team's expectations. Once approved, the PR can be merged, making the changes an official part of the codebase.

Merging PRs ⚡⚡

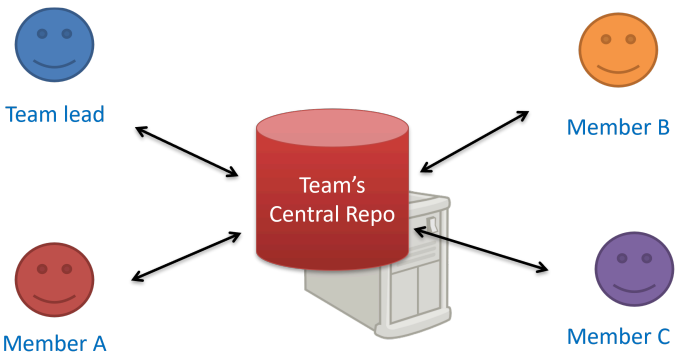
A project member with sufficient access to the remote repo can merge a PR, incorporating proposed changes into the main codebase. **Merging a PR** is similar to performing a Git merge in a local repo, except that it occurs in the remote repository.

After a PR is merged, you need to sync other related repos. Merging a PR simply merges the code in the upstream remote repository in which it was merged. The PR author (and other members of the repo) needs to pull the merged code from the upstream repo to their local repos and push the new code to their respective forks to sync the fork with the upstream repo.

DRCS vs CRCS ⚡⚡⚡

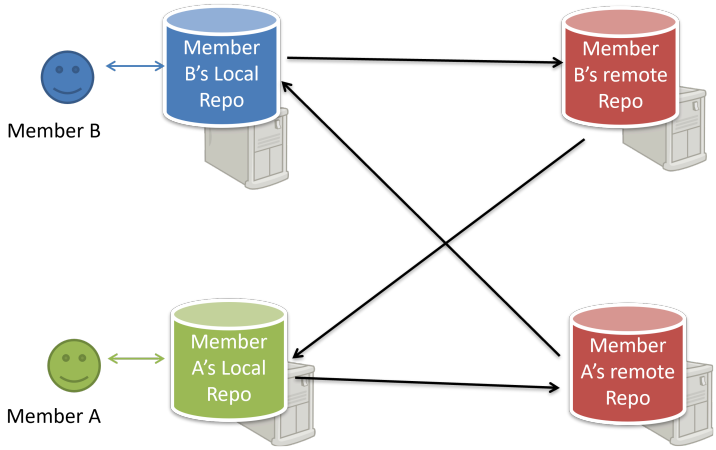
RCS can be done in two ways: the *centralized* way and the *distributed* way.

Centralized RCS (CRCS for short) uses a central remote repo that is shared by the team. Team members interact directly with this central repository. Older RCS tools such as CVS and SVN support only this model.



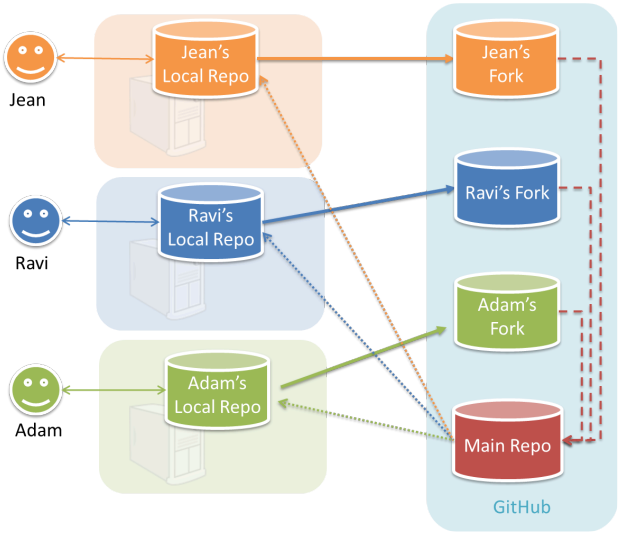
The centralized RCS approach without any local repos (e.g., CVS, SVN)

Distributed RCS (DRCS for short, also known as Decentralized RCS) allows multiple remote/local repos working together. The workflow can vary from team to team. For example, every team member can have his/her own remote repository in addition to their own local repository, as shown in the diagram below. Git and Mercurial are some prominent RCS tools that support the distributed approach.



The decentralized RCS approach

Forking Workflow ⚡⚡⚡



In the *forking workflow*, the 'official' version of the software is kept in a remote repo designated as the 'main repo'. All team members fork the main repo and create pull requests from their fork to the main repo.

To illustrate how the workflow goes, let's assume Jean wants to fix a bug in the code. Here are the steps:

1. Jean creates a separate branch in her local repo and fixes the bug in that branch.
   ❗ Common mistake: Doing the proposed changes in the `master` branch -- if Jean does that, she will not be able to have more than one PR open at any time because any changes to the `master` branch will be reflected in all open PRs.
2. Jean pushes the branch to her fork.
3. Jean creates a pull request from that branch in her fork to the main repo.
4. Other members review Jean's pull request.
5. If reviewers suggested any changes, Jean updates the PR accordingly.
6. When reviewers are satisfied with the PR, one of the members (usually the team lead or a designated 'maintainer' of the main repo) merges the PR, which brings Jean's code to the main repo.
7. Other members, realizing there is new code in the upstream repo, sync their forks with the new upstream repo (i.e., the main repo). This is done by pulling the new code to their own local repo and pushing the updated code to their own fork. If there are unmerged branches in the local repo, they can be updated too e.g., by merging the new `master` branch to each of them.
   ❗ Possible mistake: Creating another 'reverse' PR from the team repo to the team member's fork to sync the member's fork with the merged code. PRs are meant to go from downstream repos to upstream repos, not in the other direction.

One main benefit of this workflow is that it does not require most contributors to have write permissions to the main repository. Only those who are merging PRs need write permissions. The main drawback of this workflow is the extra overhead of sending everything through forks.

You can follow the steps in the simulation of a forking workflow given below to learn how to follow such a workflow.

ℹ This activity is best done as a team.

Step 1. One member: set up the team org and the team repo.

1.1) **Create a GitHub organization** for your team, if you don't have one already. The org name is up to you. We'll refer to this organization as *team org* from now on.

1.2) **Add a team** called `developers` to your team org.

1.3) **Add team members** to the `developers` team.

1.4) **Fork** se-edu/samplerepo-workflow-practice to your team org. We'll refer to this as the *team repo*.

1.5) **Add the forked repo to the `developers` team.** Give write access.

Step 2. Each team member: create PRs via own fork.

2.1) **Fork that repo** from your team org to your own GitHub account.

2.2) **Create a branch** named `add-{your name}-info` (e.g. `add-johnTan-info`) in the local repo.

2.3) **Add a file** `yourName.md` into the `members` directory (e.g., `members/johnTan.md`) containing some info about you into that branch.

2.4) **Push that branch to your fork**.

2.5) **Create a PR** from that branch to the `master` branch of the team repo.

Step 3. For each PR: review, update, and merge.

3.1) **[A team member (not the PR author)] Review the PR** by adding comments (can be just dummy comments).

3.2) **[PR author] Update the PR** by pushing more commits to it, to simulate updating the PR based on review comments.

3.3) **[Another team member] Approve and merge** the PR using the GitHub interface.

3.4) **[All members] Sync your local repo (and your fork)** with upstream repo. In this case, your *upstream repo* is the repo in your team org.

- The basic mechanism for this has two steps (which you can do using Git CLI or any Git GUI):
  (1) First, pull from the upstream repo -- this will update your clone with the latest code from the upstream repo.
  If there are any unmerged branches in your local repo, you can update them too e.g., you can merge the new `master` branch to each of them.
  (2) Then, push the updated branches to your fork. This will also update any PRs from your fork to the upstream repo.
- Some alternatives mechanisms to achieve the same can be found in this GitHub help page.
  If you are new to Git, we recommend that you use the above two-step mechanism instead, so that you get a better view of what's actually happening behind the scene.

Step 4. Create conflicting PRs.

4.1) **[One member]: Update README:** In the `master` branch, remove John Doe and Jane Doe from the `README.md`, commit, and push to the main repo.

4.2) **[Each team member] Create a PR** to add yourself under the `Team Members` section in the `README.md`. Use a new branch for the PR e.g., `add-johnTan-name`.

Step 5. Merge conflicting PRs one at a time. Before merging a PR, you'll have to resolve conflicts.

5.1) [Optional] A member can inform the PR author (by posting a comment) that there is a conflict in the PR.

5.2) **[PR author] Resolve the conflict locally**:

1. Pull the `master` branch from the repo in your team org.
2. Merge the pulled `master` branch to your PR branch.
3. Resolve the merge conflict that crops up during the merge.
4. Push the updated PR branch to your fork.

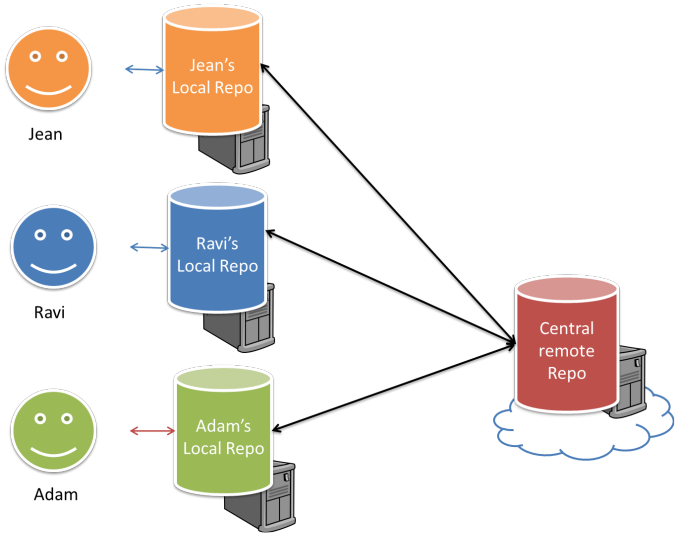
5.3) **[Another member or the PR author]: Merge the de-conflicted PR**: When GitHub does not indicate a conflict anymore, you can go ahead and merge the PR.

Feature Branch flow ϟϟϟ

***Feature branch* workflow is similar to forking workflow except there are no forks.** Everyone is pushing/pulling from the same remote repo. The phrase *feature branch* is used because each new feature (or bug fix, or any other modification) is done in a separate branch and merged to the `master` branch when ready. Pull requests can still be created within the central repository, from the feature branch to the main branch.

As this workflow require all team members to have write access to the repository,

- it is better to *protect* the main branch using some mechanism, to reduce the risk of accidental undesirable changes to it.
- it is not suitable for situations where the code contributors are not 'trusted' enough to be given write permission.



Centralised flow ϟϟϟ

The *centralized workflow* is similar to the feature branch workflow except all changes are done in the `master` branch.

SDLC process models

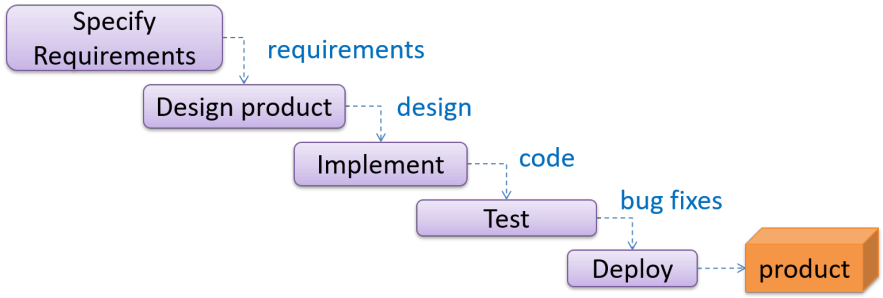
Introduction

What ⚡⚡

Software development goes through different stages such as *requirements, analysis, design, implementation* and *testing*. These stages are collectively known as the *software development lifecycle* (SDLC). There are several approaches, known as *software development lifecycle models* (also called *software process models*), that describe different ways to go through the SDLC. Each process model prescribes a 'roadmap' for the software developers to manage the development effort. The roadmap describes the aims of the development stages, the outcome of each stage, and the workflow i.e., the relationship between stages.

Sequential Models ⚡⚡

The *sequential model*, also called the *waterfall* model, views software development as a linear process, in which the project is seen as progressing through the development stages. The name *waterfall* stems from how the model is drawn to look like a waterfall (see below).



When one stage of the process is completed, it produces some artifacts to be used in the next stage. For example, the requirements stage produces a comprehensive list of requirements, to be used in the design phase.

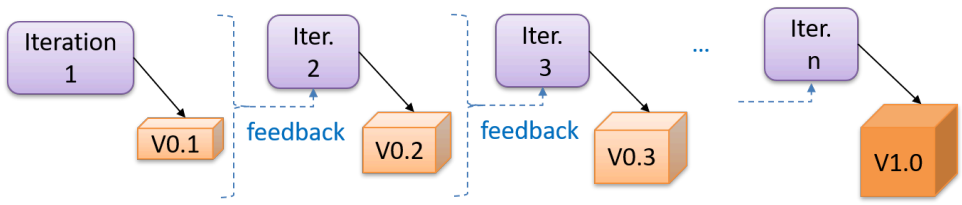
A strict sequential model project moves only in the forward direction i.e., each stage is completed before starting the next. For example, once the requirements stage is over, there is no provision for revising the requirements later.

This model can work well for a project that produces software to solve a well-understood problem, in which case the requirements can remain stable and the effort can be estimated accurately. Furthermore, as each stage has a well-defined outcome, it is easy to track the progress of the project because one can gauge the project progress by monitoring which stage the project is in.

However, real-world projects often tackle problems that are not well-understood at the beginning, making them unsuitable for this model. For example, target users of a software product may not be able to state their requirements accurately at the start of the project, if they have not used a similar product before.

Iterative Models ⚡⚡

The iterative model advocates producing the software by going through several *iterations*. Each of the iterations could potentially go through all the stages of the SDLC, from requirements gathering to deployment.



Each iteration produces a new version of the product, building upon the version produced in the previous iteration. Feedback from each iteration is factored into the subsequent iterations. For example, if an implementation task took longer than expected, the effort estimate for a similar tasks in future iterations can be adjusted accordingly. Similarly, if a feature introduced in the current iteration was not well-received by target users, it can be removed or tweaked in the next iteration.

The iterative model can be done in *breadth-first* or *depth-first* approach.

- **In the breadth-first approach, an iteration evolves all major components and all functionality areas in parallel** i.e., most features and most components will be updated in each iteration, producing a working product at the end of each iteration.

- **In the depth-first approach, an iteration focuses on fleshing out only some components or some functionality area.** Accordingly, early depth-first iterations might not produce a working product.

> 📦 Taking a Minesweeper game as an example,
>
> - breadth-first iterations will deliver a fully playable version early. These early versions may have primitive functionality, for example, a rudimentary text based UI, fixed board size, limited minefield layouts, etc. These functionalities (and corresponding components) will then be improved in later releases.
> - an early depth-first iteration could deliver the full user interface (UI) but with no game logic at all. Alternatively, an early iteration could focus on just the logic for generating initial layouts of the minefield. Neither will be a playable version of the game but both can be used to collect early feedback (about the UI, and the initial minefield layouts, respectively) which can then be used to guide later iterations.

A project can be done as a mixture of breadth-first and depth-first iterations i.e., an iteration can contain some breadth-first work as well as some depth-first work, or, some iterations can be breadth-first while others are depth-first.

Agile Models ϟϟϟ

In 2001, a group of prominent software engineering practitioners met and brainstormed for an alternative to documentation-driven, heavyweight software development processes that were used in most large projects at the time. This resulted in something called the *agile manifesto* (a vision statement of what they were looking to do).

> You are uncovering better ways of developing software by doing it and helping others do it.
>
> Through this work you 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
>
> That is, while there is value in the items on the right, you value the items on the left more.
>
> -- Extract from the Agile Manifesto

Subsequently, some of the signatories of the manifesto went on to create process models that try to follow it. These processes are collectively called agile processes. Some of the key features of agile approaches are:

- Requirements are prioritized based on the needs of the user, are clarified regularly (at times almost on a daily basis) with the entire project team, and are factored into the development schedule as appropriate.
- 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.
- There is a 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.

There are a number of agile processes in the development world today. *eXtreme Programming (XP)* and *Scrum* are two of the well-known ones.

Example process models

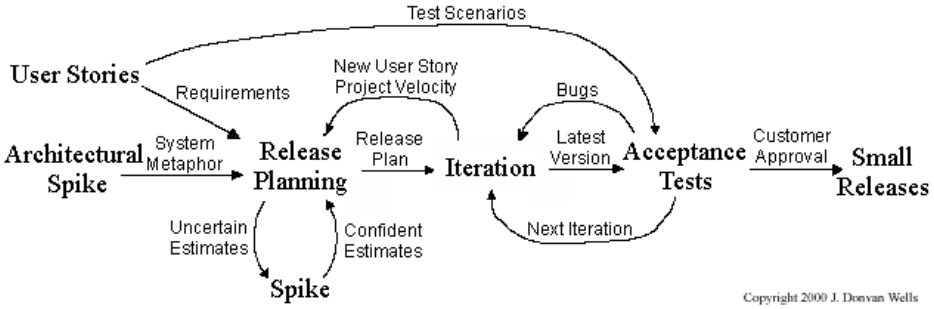
XP ϟϟϟ

The following description was adapted from the XP home page, emphasis added:

> **Extreme Programming (XP) stresses customer satisfaction.** Instead of delivering everything you could possibly want on some date far in the future, this process delivers the software you need as you need it.
>
> **XP aims to empower developers to confidently respond to changing customer requirements**, even late in the lifecycle.
>
> **XP emphasizes teamwork.** Managers, customers, and developers are all equal partners in a collaborative team. XP implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

XP aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. Every small success deepens their respect for the unique contributions of each and every team member. With this foundation, Extreme Programmers are able to courageously respond to changing requirements and technology.

XP has a set of simple rules. XP is a lot like a jig saw puzzle with many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This flow chart shows how Extreme Programming's rules work together.



***Pair programming, CRC cards, project velocity*, and *standup meetings* are some interesting topics related to XP.** Refer to www.extremeprogramming.org to find out more about XP.

Scrum ⚡⚡⚡

This description of Scrum was adapted from Wikipedia [retrieved on 18/10/2011], emphasis added:

Scrum is a process skeleton that contains sets of practices and predefined roles. The main roles in Scrum are:

- **The Scrum Master**, who maintains the processes (typically in lieu of a project manager)
- **The Product Owner**, who represents the stakeholders and the business
- **The Team**, a cross-functional group who do the actual analysis, design, implementation, testing, etc.

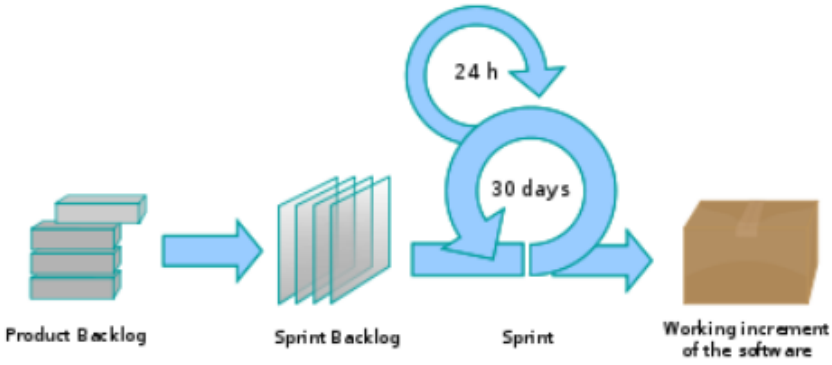
A Scrum project is divided into iterations called Sprints. A sprint is the basic unit of development in Scrum. Sprints tend to last between one week and one month, and are a timeboxed (i.e., restricted to a specific duration) effort of a constant length.

Each sprint is preceded by a planning meeting, where the tasks for the sprint are identified and an estimated commitment for the sprint goal is made, and followed by a review or retrospective meeting, where the progress is reviewed and lessons for the next sprint are identified.

During each sprint, the team creates a potentially deliverable product increment (for example, working and tested software). The set of features that go into a sprint come from the product backlog, which is a prioritized set of high level requirements of work to be done. Which backlog items go into the sprint is determined during the sprint planning meeting. During this meeting, the Product Owner informs the team of the items in the product backlog that he or she wants completed. The team then determines how much of this they can commit to complete during the next sprint, and records this in the sprint backlog. During a sprint, no one is allowed to change the sprint backlog, which means that the requirements are frozen for that sprint. Development is timeboxed such that the sprint must end on time; if requirements are not completed for any reason they are left out and returned to the product backlog. After a sprint is completed, the team demonstrates the use of the software.

Scrum enables the creation of self-organizing teams by encouraging co-location of all team members, and verbal communication between all team members and disciplines in the project.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.



***Daily Scrum* is another key scrum practice**. The description below was adapted from https://www.mountaingoatsoftware.com (emphasis added):

> **In Scrum, on each day of a sprint, the team holds a daily scrum meeting called the "daily scrum."** Meetings are typically held in the same location and at the same time each day. Ideally, a daily scrum meeting is held in the morning, as it helps set the context for the coming day's work. These scrum meetings are strictly time-boxed to 15 minutes. This keeps the discussion brisk but relevant.
>
> ...
>
> **During the daily scrum, each team member answers the following three questions:**
>
> - What did you do yesterday?
> - What will you do today?
> - Are there any impediments in your way?
>
> ...
>
> The daily scrum meeting is not used as a problem-solving or issue resolution meeting. **Issues that are raised are taken offline and usually dealt with by the relevant subgroup immediately after the meeting.**

Project planning

Work Breakdown Structure ⚡⚡⚡

A *Work Breakdown Structure (WBS)* depicts information about tasks and their details in terms of subtasks. When managing projects, it is useful to divide the total work into smaller, well-defined units. Relatively complex tasks can be further split into subtasks. In complex projects, a WBS can also include prerequisite tasks and effort estimates for each task.

📦 The high level tasks for a single iteration of a small project could look like the following:

Task ID	Task	Estimated Effort	Prerequisite Task
A	Analysis	1 man day	-
B	Design	2 man day	A
C	Implementation	4.5 man day	B
D	Testing	1 man day	C
E	Planning for next version	1 man day	D

The effort is traditionally measured in *man hour/day/month* i.e., work that can be done by one person in one hour/day/month. The *Task ID* is a label for easy reference to a task. Simple labeling is suitable for a small project, while a more informative labeling system can be adopted for bigger projects.

📦 An example WBS for a game development project.

Task ID	Task	Estimated Effort	Prerequisite Task
A	High level design	1 man day	-
B	Detail design 1. User Interface 2. Game Logic 3. Persistency Support	2 man day • 0.5 man day • 1 man day • 0.5 man day	A
C	Implementation 1. User Interface 2. Game Logic 3. Persistency Support	4.5 man day • 1.5 man day • 2 man day • 1 man day	• B.1 • B.2 • B.3
D	System Testing	1 man day	C
E	Planning for next version	1 man day	D

All tasks should be well-defined. In particular, it should be clear as to when the task will be considered *done*.

📦 Some examples of ill-defined tasks and their better-defined counterparts:

👎 Bad	👍 Better
more coding	implement component X

Bad	Better
do research on UI testing	find a suitable tool for testing the UI

Milestones

A *milestone* is the end of a stage which indicates significant progress. You should take into account dependencies and priorities when deciding on the features to be delivered at a certain milestone.

Each intermediate product release is a milestone.

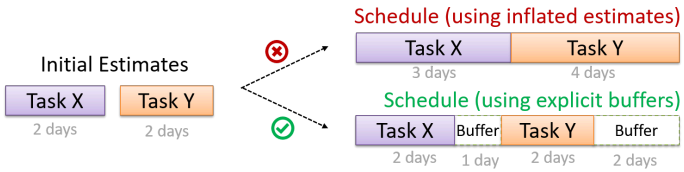
In some projects, it is not practical to have a very detailed plan for the whole project due to the uncertainty and unavailability of required information. In such cases, you can use a high-level plan for the whole project and a detailed plan for the next few milestones.

Milestones for the Minesweeper project, iteration 1

Day	Milestones
Day 1	Architecture skeleton completed
Day 3	'new game' feature implemented
Day 4	'new game' feature tested

Buffers

A *buffer* is time set aside to absorb any unforeseen delays. It is very important to include buffers in a software project schedule because effort/time estimations for software development are notoriously hard. However, **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 feedback to improve future effort estimates.

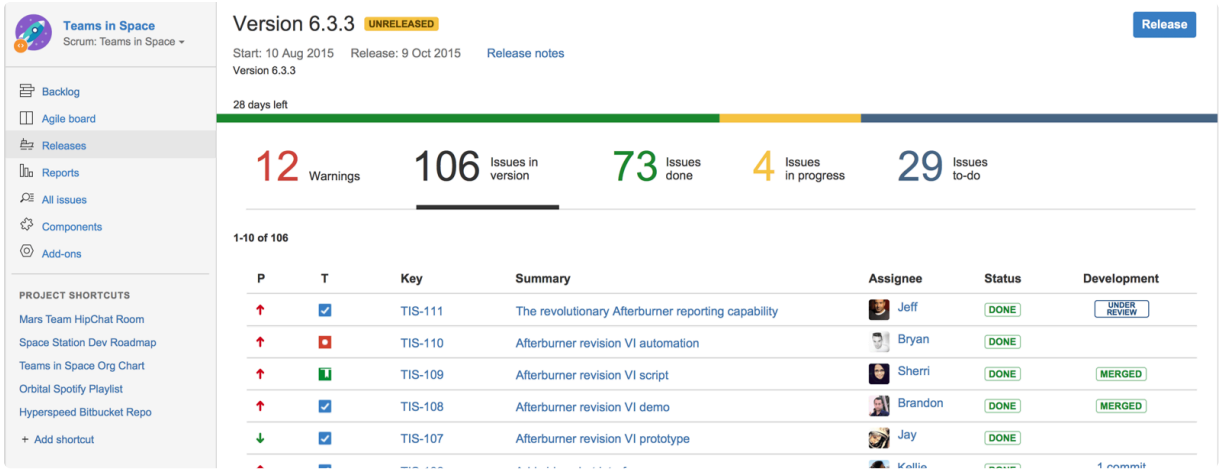


Issue Trackers

Keeping track of project tasks (who is doing what, which tasks are ongoing, which tasks are done etc.) is an essential part of project management. In small projects, it may be possible to keep track of tasks using simple tools such as online spreadsheets or general-purpose/light-weight task tracking tools such as Trello. Bigger projects need more sophisticated task tracking tools.

Issue trackers (sometimes called bug trackers) are commonly used to track task assignment and progress. Most online project management software such as GitHub, SourceForge, and BitBucket come with an integrated issue tracker.

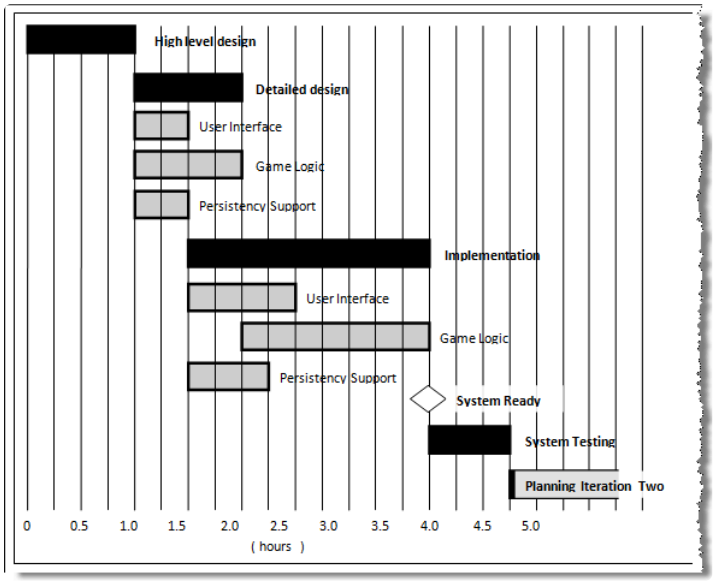
A screenshot from the Jira Issue tracker software (Jira is part of the BitBucket project management tool suite):



GANTT Charts ϟϟϟ

A *Gantt chart* is a 2-D bar-chart, drawn as *time vs tasks* (represented by horizontal bars).

A sample Gantt chart:

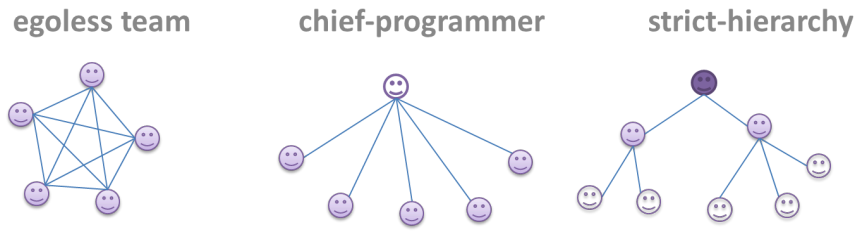


In a Gantt chart, a solid bar represents the main task, which is generally composed of a number of subtasks, shown as grey bars. The diamond shape indicates an important deadline/deliverable/milestone.

Teamwork

Team Structures ⚡⚡⚡

Given below are three commonly used team structures in software development. Irrespective of the team structure, it is a good practice to assign roles and responsibilities to different team members so that someone is clearly in charge of each aspect of the project. In comparison, the 'everybody is responsible for everything' approach can result in more chaos and hence slower progress.



Egoless team

In this structure, **every team member is equal in terms of responsibility and accountability**. When any decision is required, consensus must be reached. This team structure is also known as a *democratic* team structure. This team structure usually finds a good solution to a relatively hard problem as all team members contribute ideas.

However, 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

Frederick Brooks proposed that software engineers learn from the medical surgical team in an operating room. In such a team, there is always a chief surgeon, assisted by experts in other areas. Similarly, in a chief programmer team structure, **there is a single authoritative figure, the chief programmer**. Major decisions, e.g., system architecture, are made solely by him/her and obeyed by all other team members. The chief programmer directs and coordinates the effort of other team members. When necessary, the chief will be assisted by domain specialists e.g., business specialists, database experts, network technology experts, etc. This allows individual group members to concentrate solely on the areas in which they have sound knowledge and expertise.

The success of such a team structure relies heavily on the chief programmer. Not only must he/she be a superb technical hand, he/she also needs good managerial skills. Under a suitably qualified leader, such a team structure is known to produce successful work.

Strict hierarchy team

At the opposite extreme of an egoless team, a strict hierarchy team has **a strictly defined organization among the team members**, reminiscent of the military or a bureaucratic government. Each team member only works on his/her assigned tasks and reports to a single "boss".

In a large, resource-intensive, complex project, this could be a good team structure to reduce communication overhead.

SECTION: PRINCIPLES

Principles

Single Responsibility Principle

> **Single responsibility principle (SRP)**: A class should have one, and only one, reason to change. -- Robert C. Martin

If a class has only one responsibility, it needs to change only when there is a change to that responsibility.

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.

> Gather together the things that change for the same reasons. Separate those things that change for different reasons. _-
>
> *Agile Software Development, Principles, Patterns, and Practices* by Robert C. Martin

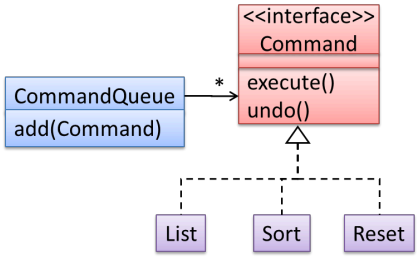
Open-Closed Principle

The Open-Closed Principle aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself.

> **Open-closed principle (OCP)**: A module should be *open* for extension but *closed* for modification. That is, modules should be written so that they can be extended, without requiring them to be modified. -- proposed by Bertrand Meyer

In object-oriented programming, OCP can be achieved in various ways. This often requires separating the *specification (i.e., interface)* of a module from its *implementation*.

In the design given below, the behavior 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 behavior was extended without having to modify its code. Hence, it is open to extensions, but closed to modification.



The behavior of a Java generic class can be altered by passing it a different class as a parameter. In the code below, the `ArrayList` class behaves as a container of `Students` in one instance and as a container of `Admin` objects in the other instance, without having to change its code. That is, the behavior of the `ArrayList` class is extended without modifying its code.

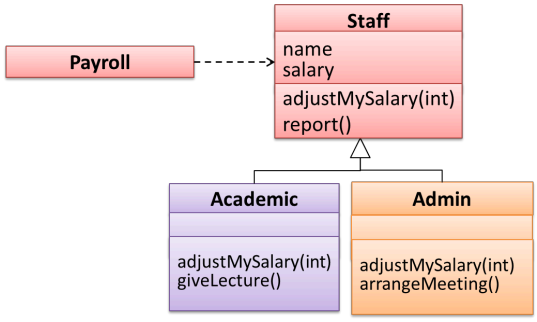
```
ArrayList students = new ArrayList<Student>();
ArrayList admins = new ArrayList<Admin>();
```

Liskov Substitution Principle

Liskov substitution principle (LSP): Derived classes must be substitutable for their base classes. -- proposed by Barbara Liskov

LSP sounds the same as substitutability but it goes beyond substitutability; **LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.** As you know, Java has language support for substitutability. However, if LSP is not followed, substituting a subclass object for a superclass object can break the functionality of the code.

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.

Another example

SOLID Principles ⚡⚡⚡

The five OOP principles given below are known as *SOLID Principles* (an acronym made up of the first letter of each principle):

Single Responsibility Principle (SRP)

Open-Closed Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

Separation of Concerns Principle ⚡⚡

Separation of concerns principle (SoC): To achieve better modularity, separate the code into distinct sections, such that each section addresses a separate *concern*. -- Proposed by Edsger W. Dijkstra

A *concern* in this context is a set of information that affects the code of a computer program.

Examples for *concerns*:

- A specific feature, such as the code related to the `add employee` feature
- A specific aspect, such as the code related to `persistence` or `security`
- A specific entity, such as the code related to the `Employee` entity

Applying SoC reduces functional overlaps among code sections and also limits the ripple effect when changes are introduced to a specific part of the system.

If the code related to *persistence* is separated from the code related to *security*, a change to how the data are persisted will not need changes to how the security is implemented.

This principle can be applied at the class level, as well as at higher levels.

The n-tier architecture utilizes this principle. Each layer in the architecture has a well-defined functionality that has no functional overlap with each other.

This principle should lead to higher cohesion and lower coupling.

Law of Demeter ⚡⚡⚡

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.

Also known as

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

The following code fragment violates LoD due to the following reason: while `b` is a 'friend' of `foo` (because it receives it as a parameter), `g` is a 'friend of a friend' (which should be considered a 'stranger'), and `g.doSomething()` is analogous to 'talking to a stranger'.

```
void foo(Bar b) {
    Goo g = b.getGoo();
    g.doSomething();
}
```

LoD aims to prevent objects from navigating the internal structures of other objects.

An analogy for LoD can be drawn from Facebook. If Facebook followed LoD, you would not be allowed to see posts of friends of friends, unless they are your friends as well. If Jake is your friend and Adam is Jake's friend, you should not be allowed to see Adam's posts unless Adam is a friend of yours as well.