

Table of Contents

SOFTWARE ENGINEERING

Software Engineering	1
----------------------------	---

PROGRAMMING PARADIGMS

Object-Oriented Programming	3
-----------------------------------	---

REQUIREMENTS

Requirements	16
Gathering requirements	18
Specifying requirements	20

DESIGN

Software design	26
Design fundamentals	27
Modeling	30

IMPLEMENTATION

IDEs	36
Code quality	37
Refactoring	43
Documentation	45
Error handling	47
Integration	50
Reuse	51

QUALITY ASSURANCE

Quality assurance	54
-------------------------	----

Testing	56
---------------	----

PROJECT MANAGEMENT

Revision control	64
SDLC process models	67
Project planning	69

PRINCIPLES

Principles	72
------------------	----

TOOLS

UML	73
Git and GitHub	92

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 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) write procedural code. e.g. se-edu/addressbook-level1

☞ JavaScript and Python support functional, procedural, and OOP programming.

Objects

What ★★★★

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

Object Oriented Programming (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.

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).

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
- 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).

Examples:

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. You 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
<code>Person</code>	objects representing Adam, Beth, Charlie, Daisy
<code>AgeList</code>	an object to represent the age list
<code>Calculator</code>	an object to do the calculations
<code>Main</code>	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 module or drop the module 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.

Navigability ★★★★

When two classes are linked by an association, it does not necessarily mean the two objects taking part in an instance of the association *knows about* (i.e., has a reference to) each other. **The concept of which object in the association knows about the other object is called **navigability**.**

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`. 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 i.e., there are two separate associations here, which rules out it being 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();  
    ...  
}
```

Bidirectional associations require matching variables in both classes.

💡 In the code below, the `Foo` class has a variable to hold a `Bar` object and vice versa i.e., each object can have an association with an object of the other type.

```
class Foo {  
    Bar bar;  
    //...  
}  
  
class Bar {  
    Foo foo;  
    //...  
}
```

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 only concerned about *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. When the *whole* is destroyed, *parts* are destroyed too i.e., the *part* should not exist without being attached to a *whole*.

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

Composition also implies that there cannot be cyclical links.

💡 The 'sub-folder' association between `Folder` objects is a composition type association. That means if the `Folder` object `foo` is a sub-folder of `Folder` object `bar`, `bar` cannot be a sub-folder of `foo`.

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.

💡 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.

```

class Email {
    private Subject subject;
    ...
}

```

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;  
    }  
}
```

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 *sub-type* of the base class (and the base class is a *super-type* of the derived class), resulting in an *is a* relationship.

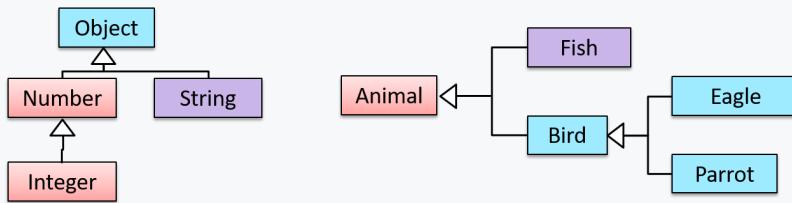
⌚ Inheritance does not necessarily mean a sub-type relationship exists. However, the two often go hand-in-hand. For simplicity, at this point let us assume inheritance implies a sub-type relationship.

⌚ To continue the previous example,

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

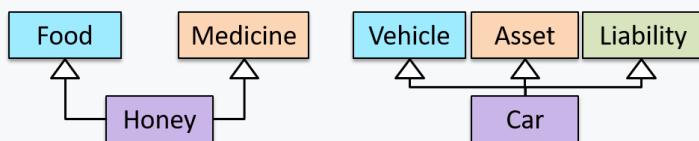
Inheritance relationships through a chain of classes can result in inheritance **hierarchies** (aka inheritance **trees**).

💡 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 sub-class changes the behavior inherited from the parent class by re-implementing the method. Overridden methods have the same name, same type signature, and same 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 <u>overloading</u> (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
<code>int add(int X, int Y)</code>	<code>(int, int)</code>
<code>void add(int A, int B)</code>	<code>(int, int)</code>
<code>void m(int X, double Y)</code>	<code>(int, double)</code>
<code>void m(double X, int Y)</code>	<code>(double, int)</code>

💡 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.

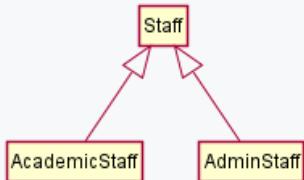
💡 **Abstract method:** An *abstract method* is a method signature without a method implementation.

💡 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
```

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".

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 **brown-field project** i.e., develop a product to replace/update an existing software product
- a **green-field project** i.e., develop a totally new system with no precedent

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

Requirements come from **stakeholders**.

- 💡 **Stakeholder:** A party that is 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.

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:

Name	<input type="text"/>
Modifiers:	<input checked="" type="radio"/> public <input type="radio"/> default <input type="radio"/> private <input checked="" type="radio"/> protected <input type="checkbox"/> abstract <input type="checkbox"/> final <input type="checkbox"/> static
Superclass:	<input type="text"/> java.lang.Object <input type="button" value="Browse..."/>

[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 code base, 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 on index cards or sticky notes, and arrange them on walls or tables, to facilitate planning and discussion. 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	I want to	so that
impatient user	to be able 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.

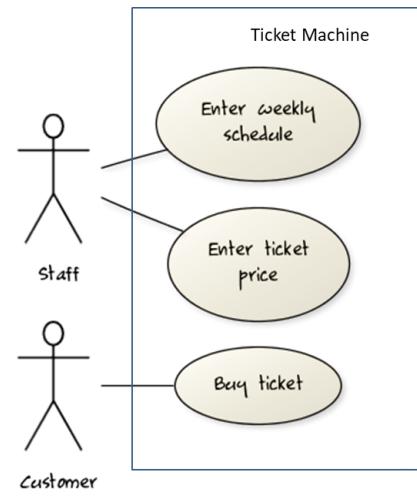
- *a. At any time, User chooses to cancel the transfer.
 - *a1. OBS requests to confirm the cancellation.
 - *a2. User confirms the cancellation.Use case ends.



➤ 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.

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.

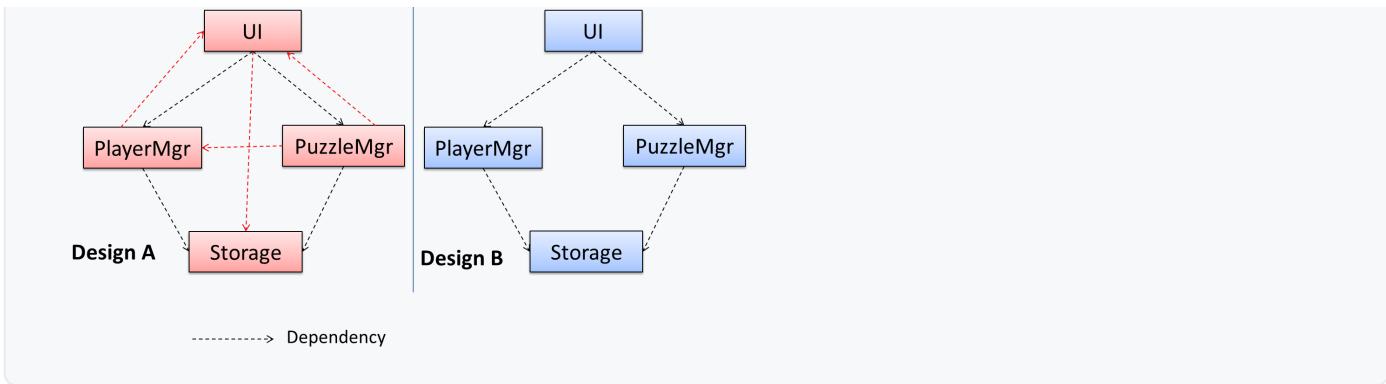
Coupling

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.

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.



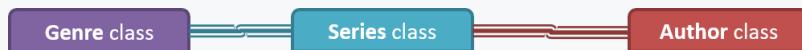
Note that these **object structures within the same software can change over time**.

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



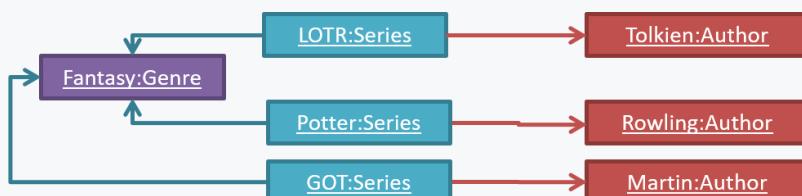
However, object structures do not change at random; they change based on a set of rules, as was decided by the designer of that software. Those **rules that object structures need to follow can be illustrated as a class structure** i.e. a structure that exists among the relevant classes.

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 are used to model object structures and UML Class Diagrams are used to model class structures of an OO solution.

Here is an object diagram for the above example:



💡 And here is the class diagram for it:



Class Diagrams (Basics) ★★★

ⓘ Contents of the panels given below belong to a different chapter; they have been embedded here for convenience and are collapsed by default to avoid content duplication in the printed version.

- [UML](#) → Class Diagrams → Introduction → What

Classes form the basis of class diagrams.

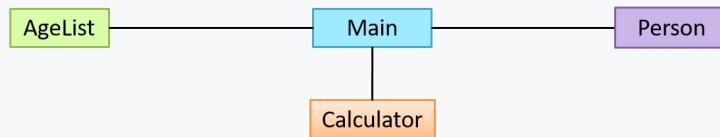
- [UML](#) → Class Diagrams → Classes → What
- [UML](#) → Class Diagrams → Class-Level Members → What

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

- [OOP](#) → Associations → What
- [UML](#) → Class Diagrams → Associations → What

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

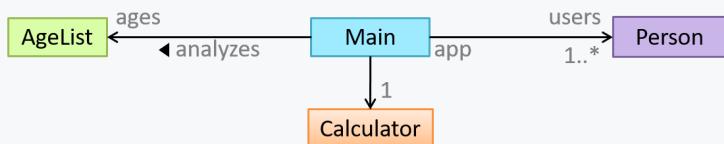
💡 An example class diagram showing associations between classes.



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
- [UML](#) → Class Diagrams → Associations → Roles
- [OOP](#) → Associations → Multiplicity
- [UML](#) → Class Diagrams → Associations → Multiplicity
- [OOP](#) → Associations → Navigability
- [UML](#) → Class Diagrams → Associations → Navigability

💡 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:

This may be redundant.
To be verified later.

This diagram is only a
work in progress.

```
graph LR; Admin[Admin] *-- staff --> Professor[Professor]; Professor -- supervisor --> Student[Student]; Professor -- "0.5 students" --> Student;
```

The screenshot shows a UML class diagram with three classes: Admin (green), Professor (blue), and Student (purple). It features several associations: Admin has a * to * association labeled 'staff' with Professor. Professor has a 1 to 1 association labeled 'supervisor' with Student, and a 0.5 to * association labeled 'students' with Student. Two red-bordered notes are present: one connected to the Admin class and another floating above the Professor class.

Class Diagrams - Intermediate ★★★★

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

Modeling inheritance

- 🔍 OOP → Inheritance → What
- 🔍 UML → Class Diagrams → Inheritance → What

Modeling composition

- 🔍 OOP → Associations → Composition
- 🔍 UML → Class Diagrams → Composition → What

Modeling aggregation

- 🔍 OOP → Associations → Aggregation

-  UML → Class Diagrams → Aggregation → What

Modeling dependencies

-  OOP → Associations → Dependencies
-  UML → Class Diagrams → Dependencies → What

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

Modeling enumerations

-  OOP → Classes → Enumerations
-  UML → Class Diagrams → Enumerations → What

Modeling abstract classes

-  OOP → Inheritance → Abstract Classes
-  UML → Class Diagrams → Abstract Classes → What

Modeling interfaces

-  OOP → Inheritance → Interfaces
-  UML → Class Diagrams → Interfaces → What

Object Diagrams

 UML → Object Diagrams → Introduction

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
-  UML → Object Diagrams → Associations

Modeling behaviors

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
-  UML  Sequence Diagrams → Basic Notation
-  UML  Sequence Diagrams → Loops
-  UML  Sequence Diagrams → Object Creation
-  UML  Sequence Diagrams → Minimal Notation

Sequence Diagrams - Intermediate ★★★☆

-  UML  Sequence Diagrams → Object Deletion
-  UML  Sequence Diagrams → Self-Invocation
-  UML  Sequence Diagrams → Alternative Paths
-  UML  Sequence Diagrams → Optional Paths
-  UML  Sequence Diagrams → Calls to Static Methods

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, and modeling support.

Examples of popular IDEs:

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

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-ball', 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: Maximise 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 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.

☞ The two code samples given below achieve the same functionality, but one is easier to read.

👎 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;  
}
```

Basic

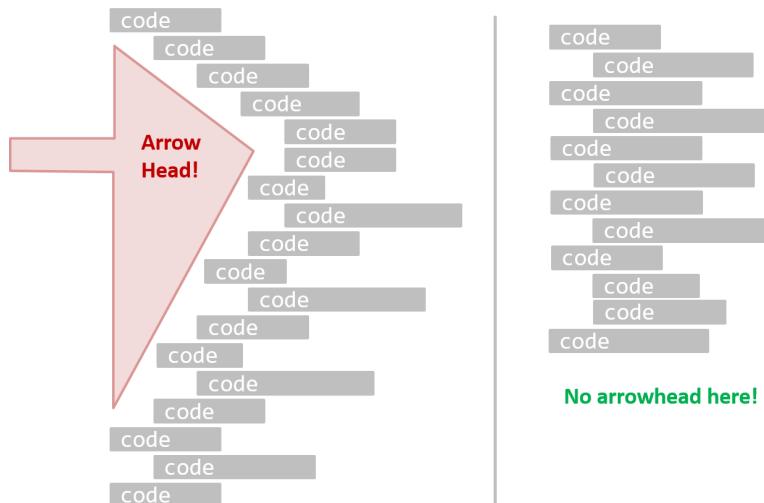
Avoid Long Methods ★★★☆

Be wary when a method is longer than the computer screen, and take corrective action when it goes beyond 30 LOC (lines of code). 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

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).

📦 Example:

👎 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 ★★★☆

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.

📦 Example:

👎 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!
```

In general, try to avoid any magic literals.

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 group related statements together.

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.

Guideline: Follow a standard

Introduction ★★☆☆

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

The aim of a coding standard is to make the entire code base 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.

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.

Examples:

Name for a	 Bad	 Good
Class	<code>CheckLimit</code>	<code>LimitChecker</code>

Name for a	👎 Bad	👍 Good
Method	<code>result()</code>	<code>calculate()</code>

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

💡 Examples:

👍 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.

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 ★★☆☆

If the code is self-explanatory, refrain from repeating the description in a comment just for the sake of 'good documentation'.

👎 Bad

```
//increment x
x++;

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

Write to the Reader ★★☆☆

Do not write comments as if they are private notes to yourself. Instead, write them well enough to be understood by another programmer. 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.

💡 Examples:

👎 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() {
    ....
}
```

Refactoring

What ★★★

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

- **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.

💡 Improving code structure 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 ★★★☆

Given below are some more commonly used refactorings. A more comprehensive list is available at [refactoring-catalog](#).

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](#)

Documentation

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 (adapted from [Oracle's Java documentation](#))

```
/**  
 * 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 ★★★★☆

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();
```

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:** They are similar in purpose but JUnit assertions are more powerful and customized for testing. In addition, JUnit assertions are not disabled by default. We recommend you 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 considered 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.

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.

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 multi-step 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.

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.



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**.



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.



Eclipse is a fully functional Java IDE out-of-the-box.

A framework facilitates the adaptation and customization of some desired functionality.



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.



- 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:

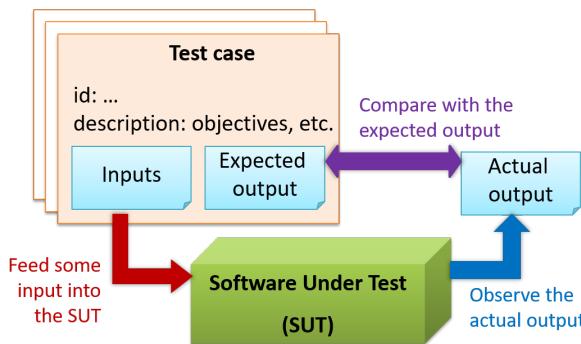
- It only proves the compliance with the specification, but not the actual utility of the software.
- It requires highly specialized notations and knowledge which makes it an expensive technique to administer. Therefore, **formal verifications are more commonly used in safety-critical software such as 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.

- Other details a test case can contain [+ extra](#)

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), unless the error is 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.

- A deeper look at the definition of testing [+ extra](#)

Testing types

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  
    }  
}
```

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.

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

Developer testing

What

Developer testing is the testing done by the developers themselves as opposed to professional testers or end-users.

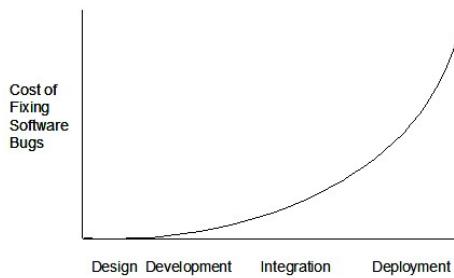
Why

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 a large 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 of partially developed software is usually, and by necessity, done by the developers themselves i.e. developer testing.

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.

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. Note that to detect regressions, you need to retest 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**.

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.

- 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`.

➤ Example `input.txt`

2. Store the output you expect from the SUT in another text file `expected.txt`.

➤ Example `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 code changes to `AddressBook`.

```
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 command window to run the app, not a PowerShell window.

➤ More on the `<` operator and the `>` operator + extra

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.

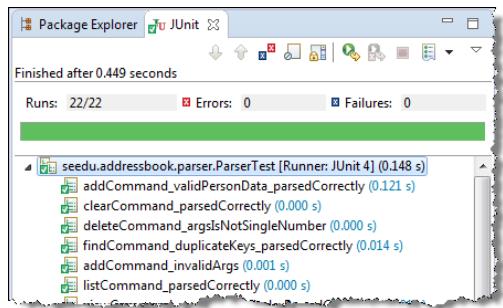
```
@Test
public void testTotalSalary() {
    Payroll p = new Payroll();

    // test case 1
    p.setEmployees(new String[]{"E001", "E002"});
    assertEquals(6400, p.totalSalary());

    // test case 2
    p.setEmployees(new String[]{"E001"});
    assertEquals(2300, p.totalSalary());

    // more tests...
}
```

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.



SECTION: PROJECT MANAGEMENT

Revision control

What ★★★★



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.

Revision control software 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.

Revision control software 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 really 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.

-- [adapted from [bryan-mercurial-guide](#)]



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



Revision: A *revision* (some seem to use it interchangeably with *version* while others seem to distinguish the two -- here, let us treat them as the same, for simplicity) is a state of a piece of information at a specific time that is a result of some changes to it e.g., if you modify the code and save the file, you have a new revision (or a version) of that file.

Revision control software are also known as *Version Control Software (VCS)*, and by a few other names.

Repositories ★★★★



Repository (*repo* for short): The database of the history of a directory being tracked by an RCS software (e.g. Git).



The **repository** is the database where the meta-data about the revision history are stored. Suppose you want to apply revision control on files in a directory called `ProjectFoo`. In that case, you need to set up a *repo* (short for repository) in the `ProjectFoo` directory, which is referred to as the *working directory* of the repo. For example, Git uses a hidden folder named `.git` inside the working directory.

You can have multiple repos in your computer, each repo revision-controlling files of a different working directory, for examples, files of different projects.

Saving History ★★★★

Tracking and ignoring

In a repo, you can specify which files to track and which files to ignore. Some files such as temporary log files created during the build/test process should not be revision-controlled.

Staging and committing



Committing saves a snapshot of the current state of the tracked files in the revision control history. Such a snapshot is also called a *commit* (i.e. the noun).

When ready to commit, you first **stage** the specific changes you want to commit. This intermediate step allows you to commit only some changes while saving other changes for a later commit.

Using History ★★★

RCS tools store the history of the working directory as a series of commits. This means you should commit after each change that you want the RCS to 'remember'.

Each commit in a repo is a recorded point in the history of the project that is uniquely identified by an auto-generated **hash** e.g. `a16043703f28e5b3dab95915f5c5e5bf4fdc5fc1`.

You can **tag** a specific commit with a more easily identifiable name e.g. `v1.0.2`.

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

To restore the state of the working directory at a point in the past, you can **checkout** the commit in concern. i.e., you can traverse the history of the working directory simply by checking out the commits you are interested in.

Remote Repositories ★★★★

Remote repositories are repos that are hosted on remote computers and allow remote access. They are especially useful for sharing the revision history of a codebase among team members of a multi-person project. They can also serve as a remote backup of your codebase.

It is possible to set up your own remote repo on a server, but the easier option is to use a remote repo hosting service such as GitHub or BitBucket.

You can **clone** a repo to create a copy of that repo in another location on your computer. The copy will even have the revision history of the original repo i.e., identical to the original repo. For example, you can clone a remote repo onto your computer to create a local copy of the remote repo.

When you clone from a repo, the original repo is commonly referred to as the **upstream** repo. A repo can have multiple upstream repos. For example, let's say a repo `repo1` was cloned as `repo2` which was then cloned as `repo3`. In this case, `repo1` and `repo2` are upstream repos of `repo3`.

You can **pull** from one repo to another, to receive new commits in the second repo, if the repos have a shared history. Let's say some new commits were added to the **upstream** repo after you cloned it and you would like to copy over those new commits to your own clone i.e., **sync** your clone with the upstream repo. In that case, you pull from the upstream repo to your clone.

You can **push** new commits in one repo to another repo which will copy the new commits onto the destination repo. Note that pushing to a repo requires you to have write-access to it. Furthermore, you can push between repos only if those repos have a shared history among them (i.e., one was created by copying the other at some point in the past).

Cloning, pushing, and pulling can be done between two local repos too, although it is more common for them to involve a remote repo.

A repo can work with any number of other repositories as long as they have a shared history e.g., `repo1` can pull from (or push to) `repo2` and `repo3` if they have a shared history between them.

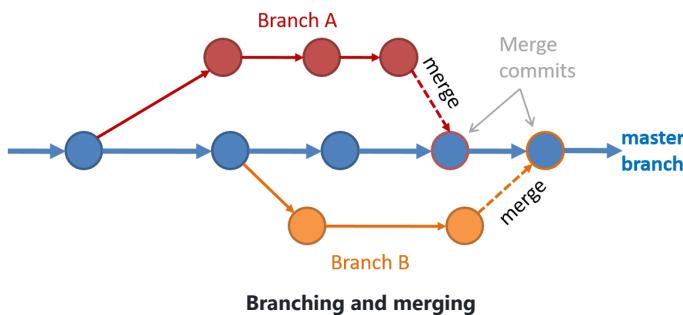
A **fork** is a remote copy of a remote repo. As you know, cloning creates a local copy of a repo. In contrast, forking creates a remote copy of a Git repo hosted on GitHub. This is particularly useful if you want to play around with a GitHub repo but you don't have write permissions to it; you can simply fork the repo and do whatever you want with the fork as you are the owner of the fork.

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". 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.

Branching ★★★☆

Branching is the process of evolving multiple versions of the software in parallel. For example, one team member can create a new branch and add an experimental feature to it while the rest of the team keeps working on another branch. Branches can be given names e.g. `master`, `release`, `dev`.

A branch can be **merged** into another branch. Merging usually results in a new commit that represents the changes done in the branch being merged.



Merge conflicts happen when you try to merge two branches that had changed the same part of the code and the RCS cannot decide which changes to keep. In those cases, you have to 'resolve' the conflicts manually.

SDLC process models

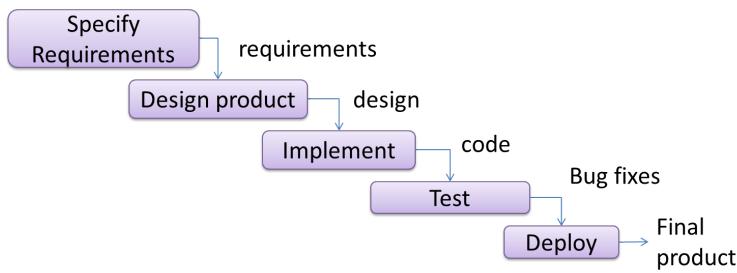
Introduction

What ★★★★

Software development goes through different stages such as **requirements**, **analysis**, **design**, **implementation** and **testing**. These stages are collectively known as the **software development life cycle (SDLC)**. There are several approaches, known as *software development life cycle 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 stage(s), the artifacts or outcome of each stage, as well as the workflow i.e. the relationship between stages.

Sequential Models ★★★★

The **sequential model**, also called the **waterfall model**, models software development as a **linear process**, in which the project is seen as progressing steadily in one direction through the development stages. The name *waterfall* stems from how the model is drawn to look like a waterfall (see below).



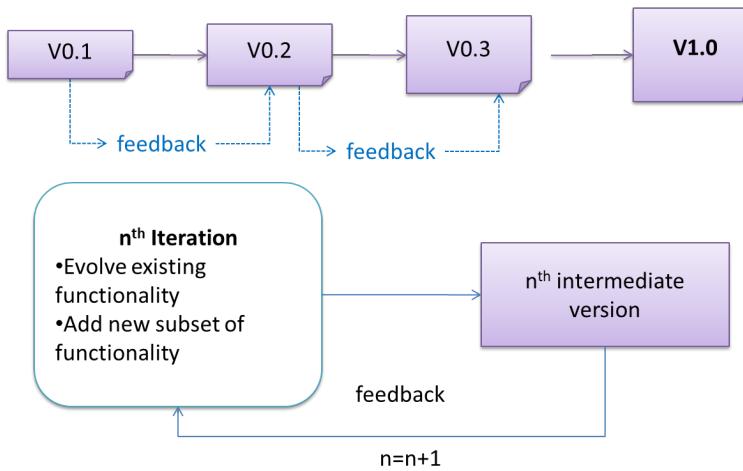
When one stage of the process is completed, it should produce some artifacts to be used in the next stage. For example, upon completion of the requirements stage, a comprehensive list of requirements is produced that will see no further modifications. A strict application of the sequential model would require each stage to be completed before starting the next.

This could be a useful model when the problem statement is well-understood and stable. In such cases, using the sequential model should result in a timely and systematic development effort, provided that all goes well. As each stage has a well-defined outcome, the progress of the project can be tracked with relative ease.

The major problem with this model is that the requirements of a real-world project are rarely well-understood at the beginning and keep changing over time. One reason for this is that users are generally not aware of how a software application can be used without prior experience in using a similar application.

Iterative Models ★★★★

The **iterative model** (sometimes called **iterative** and **incremental**) advocates having several **iterations** of **SDLC**. Each of the iterations could potentially go through all the development stages, from requirements gathering to testing & deployment. Roughly, it appears to be similar to several cycles of the sequential model.



In this model, each of the iterations produces a new version of the product. Feedback on the new version can then be fed to the next iteration. Taking the Minesweeper game as an example, the iterative model will deliver a fully playable version from the early iterations. However, the first iteration will have primitive functionality, for example, a clumsy text based UI, fixed board size, limited randomization, etc. These functionalities will then be improved in later releases.

The iterative model can take a **breadth-first** or a **depth-first** approach to iteration planning.

- **breadth-first:** an iteration evolves all major components in parallel e.g., add a new feature fully, or enhance an existing feature.
- **depth-first:** an iteration focuses on fleshing out only some components e.g., update the backend to support a new feature that will be added in a future iteration.

Most projects use 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.

Agile Models ★★★

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.

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	-
	Detail design	2 man day	
B	1. User Interface 2. Game Logic 3. Persistence Support	<ul style="list-style-type: none">• 0.5 man day• 1 man day• 0.5 man day	A
	Implementation	4.5 man day	
C	1. User Interface 2. Game Logic 3. Persistence Support	<ul style="list-style-type: none">• 1.5 man day• 2 man day• 1 man day	<ul style="list-style-type: none">• 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

Bad	Better
more coding	implement component X
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.

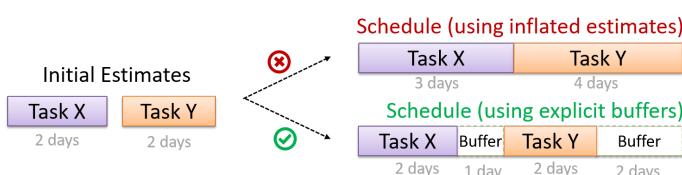
In some projects, it is not practical to have a very detailed plan for the whole project due to the uncertainty and unavailability of required information. In such cases, you can use a high-level plan for the whole project and a detailed plan for the next few milestones.

- 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):

 Teams in Space
Scrum: Teams in Space ▾

[Release](#)

[Backlog](#)

[Agile board](#)

[Releases](#) **Releases**

[Reports](#)

[All issues](#)

[Components](#)

[Add-ons](#)

PROJECT SHORTCUTS

Mars Team HipChat Room

Space Station Dev Roadmap

Teams in Space Org Chart

Orbital Spotify Playlist

Hyperspeed Bitbucket Repo

+ Add shortcut

Version 6.3.3 UNRELEASED

Start: 10 Aug 2015 Release: 9 Oct 2015 [Release notes](#)

Version 6.3.3

28 days left

12 Warnings **106 Issues in version** **73 Issues done** **4 Issues in progress** **29 Issues to-do**

1-10 of 106

P	T	Key	Summary	Assignee	Status	Development
↑	✓	TIS-111	The revolutionary Afterburner reporting capability	 Jeff	DONE	UNDER REVIEW
↑	✗	TIS-110	Afterburner revision VI automation	 Bryan	DONE	
↑	✓	TIS-109	Afterburner revision VI script	 Sherri	DONE	MERGED
↑	✓	TIS-108	Afterburner revision VI demo	 Brandon	DONE	MERGED
↓	✓	TIS-107	Afterburner revision VI prototype	 Jay	DONE	
▲	✗	-----	-----	 Malia	REVIEW	1 commit

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

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.

SECTION: TOOLS

UML

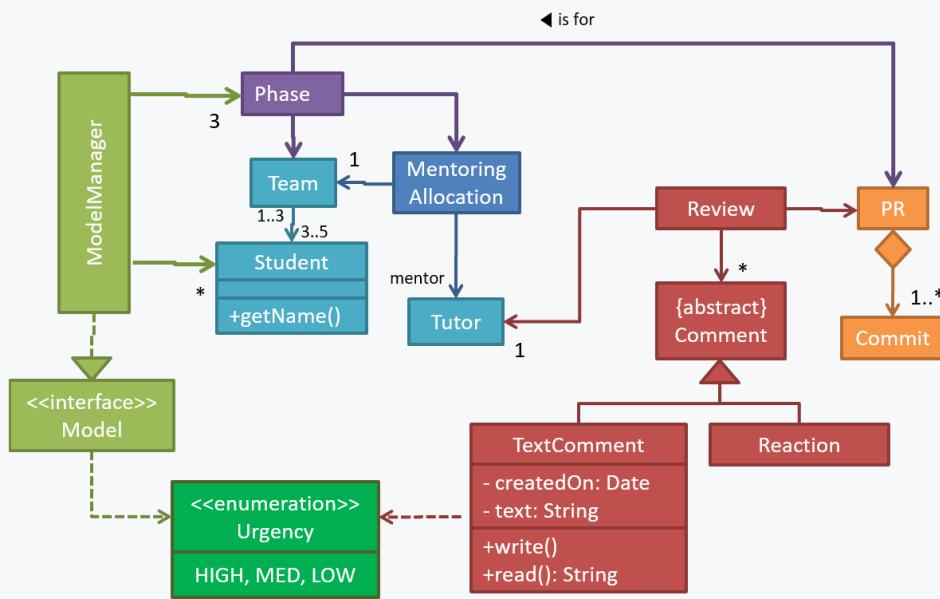
Class diagrams

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:



Classes

What ★★★★

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

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

attributes

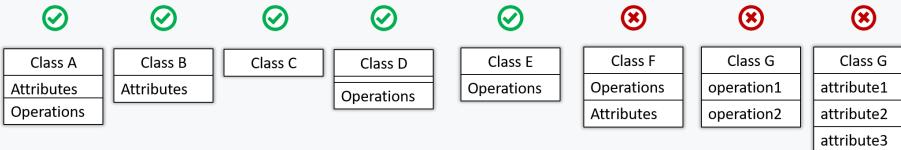
methods

⌚ A **Table** class shown in UML notation:

Table
number: Integer chairs: Chair[] = null
getNumber() : Integer setNumber(n: Integer)

➤ 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. '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

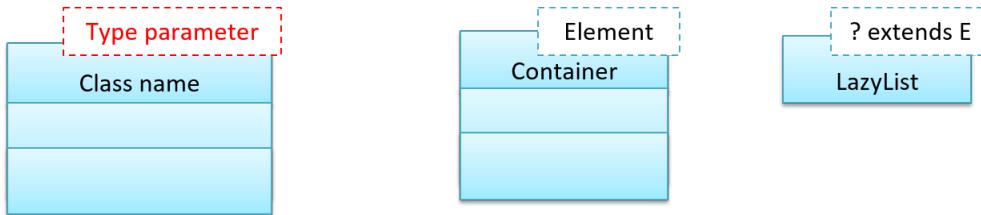
➤ How visibilities map to programming language features

➤ **Table** class with visibilities shown:

Table
- number: Integer - chairs: Chair[] = null
+ getNumber() : Integer + setNumber(n: Integer)

➤ The equivalent code

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



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:



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`.



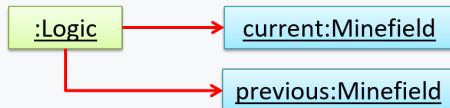
```

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

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

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.



Roles ★★★☆

Association Role labels 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)



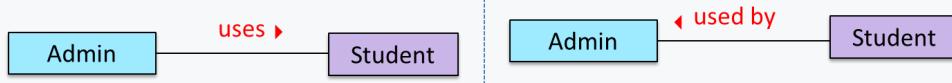
```
class Admin {  
    List<Student> charges;  
}
```

Labels ★★☆☆

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

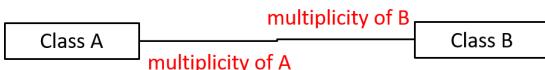


💡 In this example, the same association is described using two different labels.



- 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.

Multiplicity ★★★☆



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

Commonly used multiplicities:

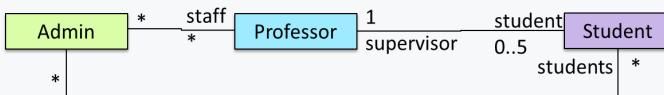
- **0..1**: *optional*, can be linked to 0 or 1 objects.
- **1**: *compulsory*, must be linked to one object at all times.
- *****: can be linked to 0 or more objects.
- **n..m**: the number of linked objects must be within **n** to **m** inclusive.

💡 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,

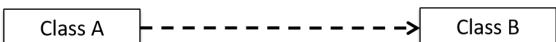
- Each student must be supervised by exactly one professor. i.e. There cannot be a student who doesn't have a supervisor or has multiple supervisors.
- A professor cannot supervise more than 5 students but can have no students to supervise.
- An admin can handle any number of professors and any number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



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`.

Associations as attributes

What ★★★★

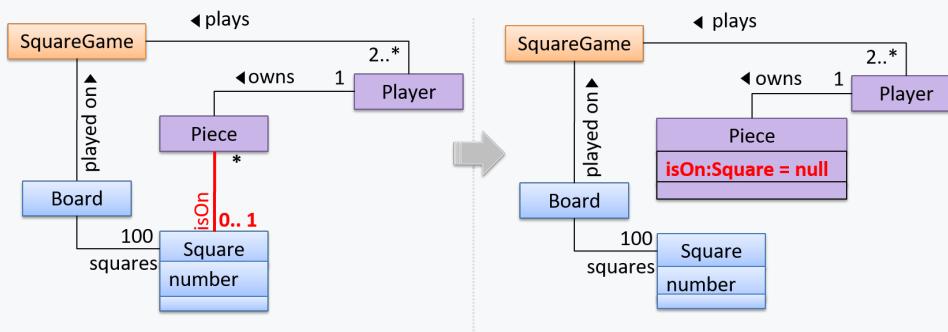
An association can be shown as an attribute instead of a line.

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

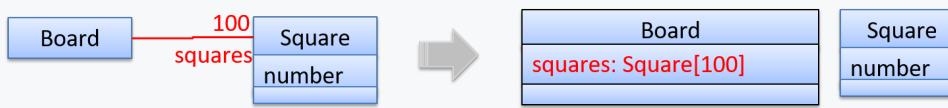
`name: type [multiplicity] = default value`

💡 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:

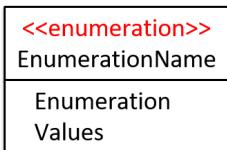


❗ Show each association as **either an attribute or a line but not both**. A line is preferred as it is easier to spot.

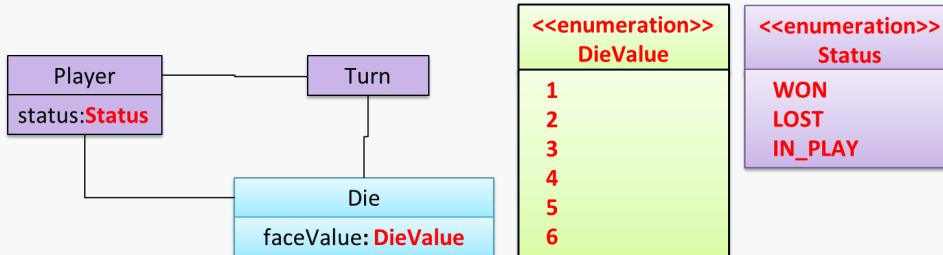
Enumerations

What ★★★★

Notation:



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

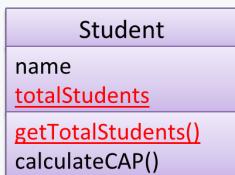


Class-level members

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.



Composition

What ★★★★

UML uses a solid diamond symbol to denote composition.

Notation:



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



Aggregation

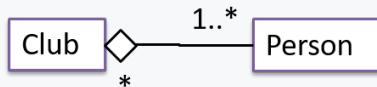
What ★★★★

UML uses a hollow diamond to indicate an aggregation.

Notation:



Example:



Aggregation vs Composition

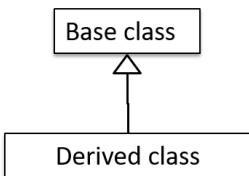
💡 The distinction between composition (◆) and aggregation (◊) is rather blurred. Martin Fowler's famous book *UML Distilled* advocates omitting the aggregation symbol altogether because using it adds more confusion than clarity.

Class inheritance

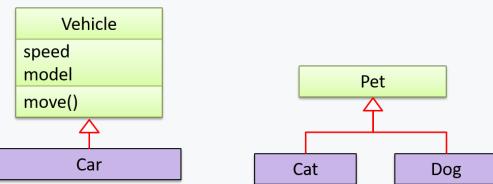
What ★★★★

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.

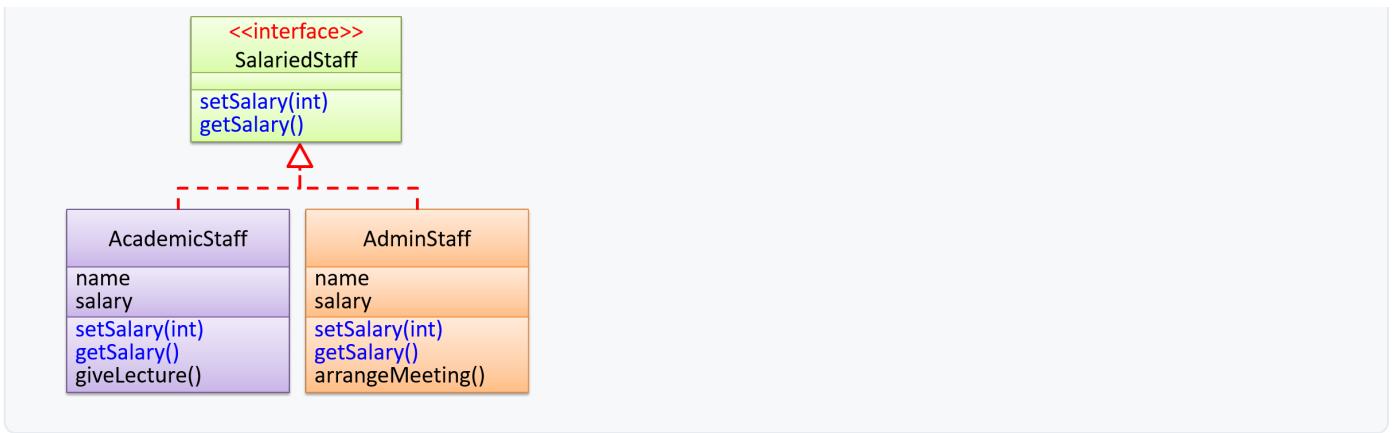


Interfaces

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.

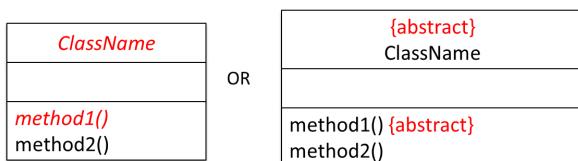
💡 The `AcademicStaff` and the `AdminStaff` classes *implement* the `SalariedStaff` interface.



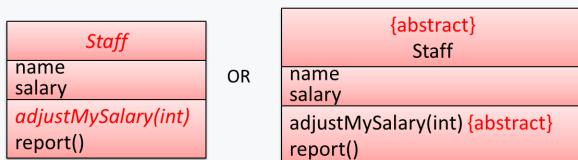
Abstract classes

What ★★★☆

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



Example:



Sequence diagrams

Introduction ★★★☆

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

💡 Consider the code below.

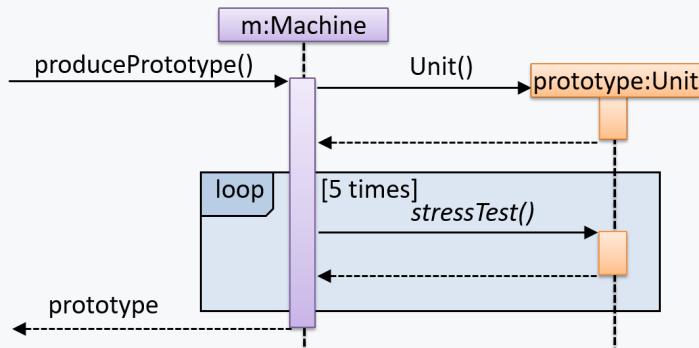
```

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

class Unit {
    public void stressTest() {
    }
}

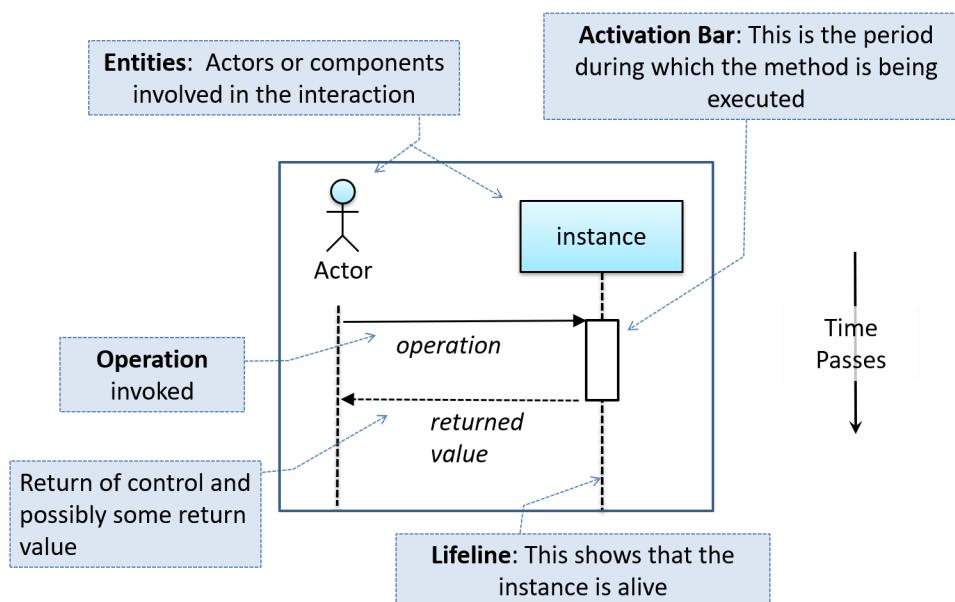
```

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

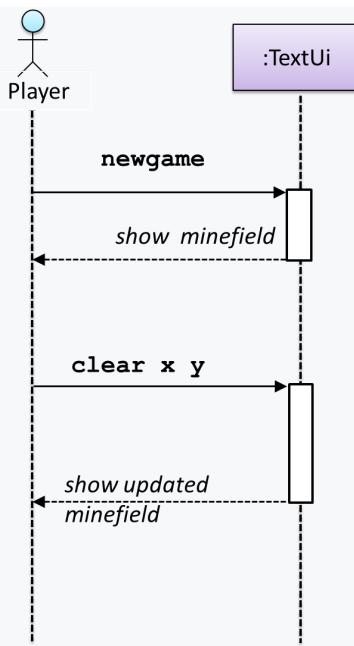


Basic ★★★★

Notation:



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



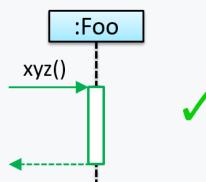
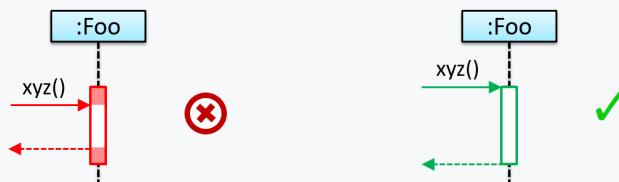
The player runs the **newgame** action on the **:TextUi** object which results in the **:TextUi** showing the minefield to the player. Then, the player runs the **clear x y** command; in response, the **:TextUi** object shows the updated minefield.

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**.

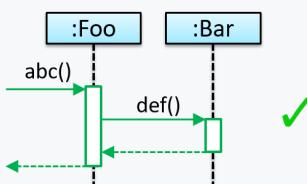
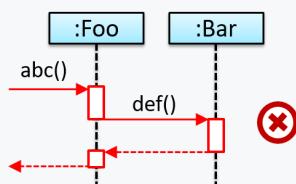
Arrows representing method calls should be solid arrows while those representing method returns should be dashed arrows.

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

✗ [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.

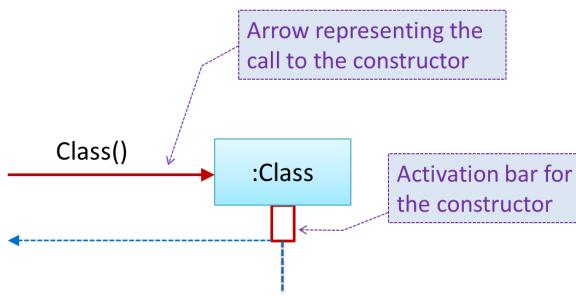


✗ [Common notation error] Broken activation bar: The activation bar should remain unbroken from the point the method is called until the method returns. In the two sequence diagrams below, the one on the left commits this error because the activation bar for the method **Foo#abc()** is not contiguous, but appears as two pieces instead.



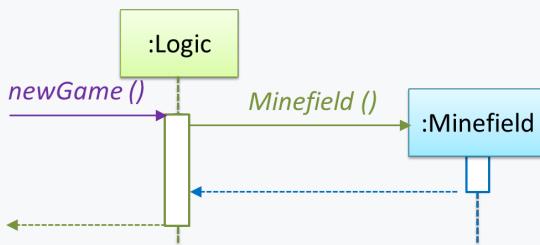
Object Creation ★★★

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.

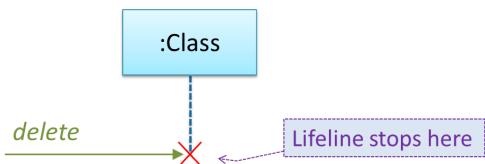


Object Deletion ★★★☆

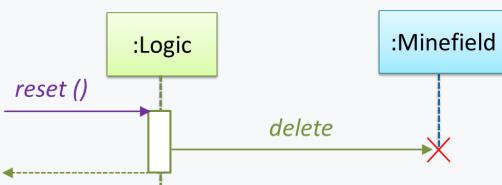
UML uses an at the end of the lifeline of an object to show its deletion.

Although object deletion is not that important in languages such as Java that support automatic memory management, you can still show object deletion in UML diagrams to indicate the point at which the object ceases to be used.

Notation:

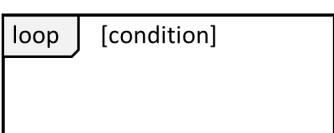


Note how the below diagram shows the deletion of the `Minefield` object.

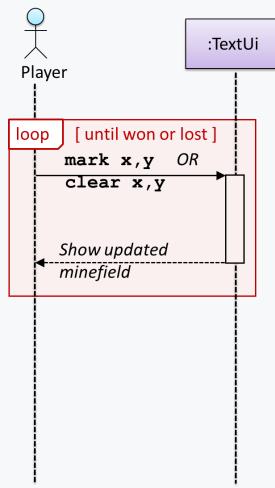


Loops ★★★★

Notation:



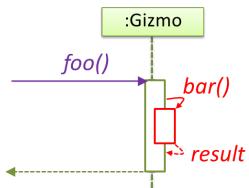
 The `Player` calls the `mark x,y` command or `clear x,y` command repeatedly until the game is won or lost.



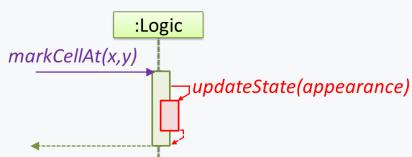
Self Invocation ★★★★

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

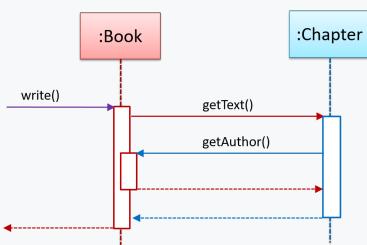
Notation:



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



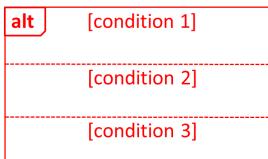
 In this variation, the `Book#write()` method is calling the `Chapter#getText()` method which in turn does a *call back* by calling the `getAuthor()` method of the calling object.



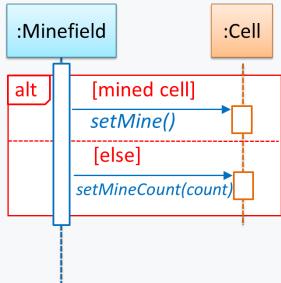
Alternative Paths ★★★★

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.

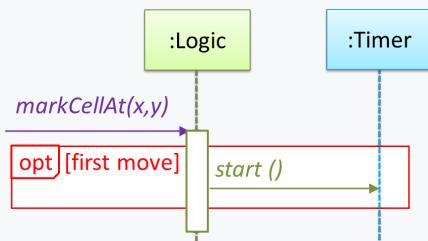
Optional Paths ★★★☆

UML uses `opt` frames to indicate optional paths.

Notation:



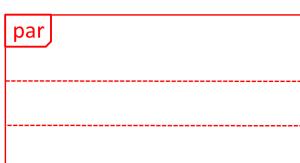
`Logic#markCellAt(...)` calls `Timer#start()` only if it is the first move of the player.



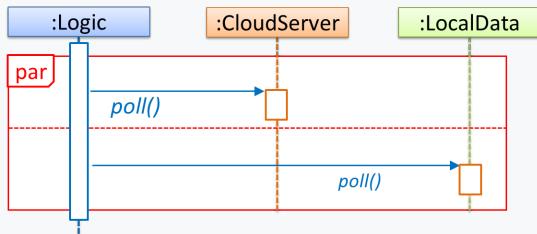
Parallel Paths ★★★☆

UML uses `par` frames to indicate parallel paths.

Notation:



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

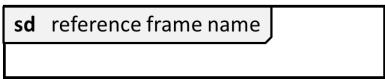
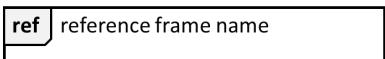


If you show parallel paths in a sequence diagram, the corresponding Java implementation is likely to be *multi-threaded* because a normal Java program cannot do multiple things at the same time.

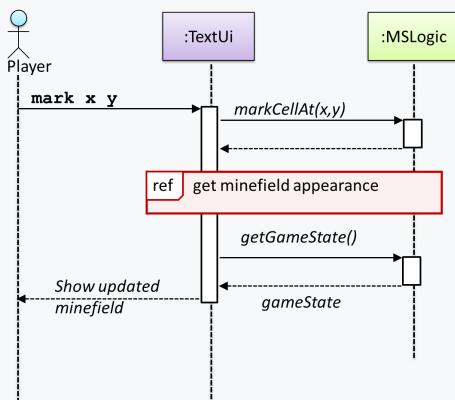
Reference Frames ★★★☆

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.

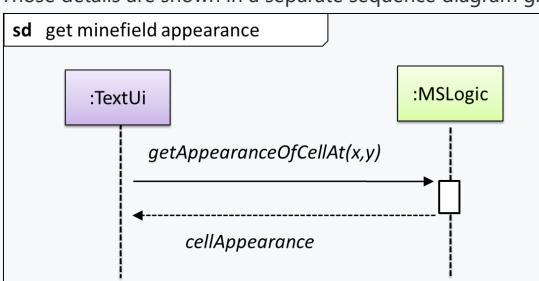
Notation:



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



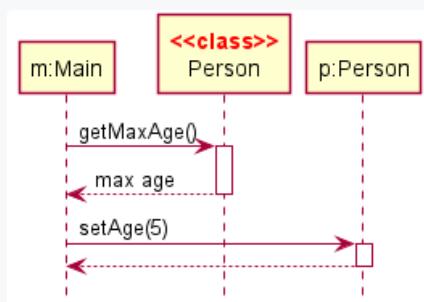
Those details are shown in a separate sequence diagram given below.



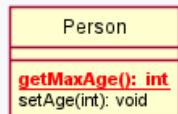
Calls to Static Methods ★★★☆

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`.



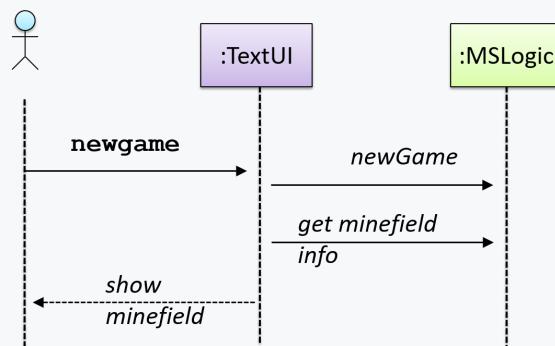
Here is the `Person` class, for reference:



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

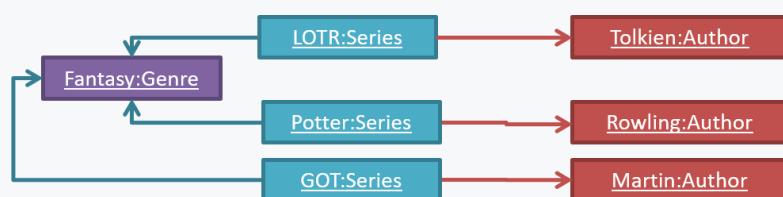


Object diagrams

Introduction ★★★★

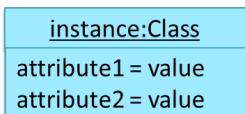
An object diagram shows an object structure at a given point of time.

💡 An example object diagram:



Objects ★★★★

Notation:



Notes:

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

Some example objects:



What ★★★☆

A solid line indicates an association between two objects.



An example object diagram showing two associations:

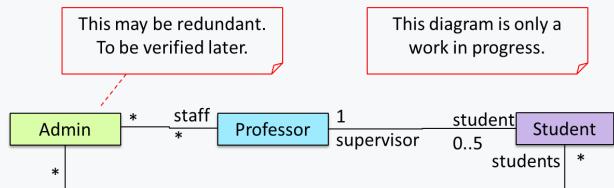


Notes

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:



Miscellaneous

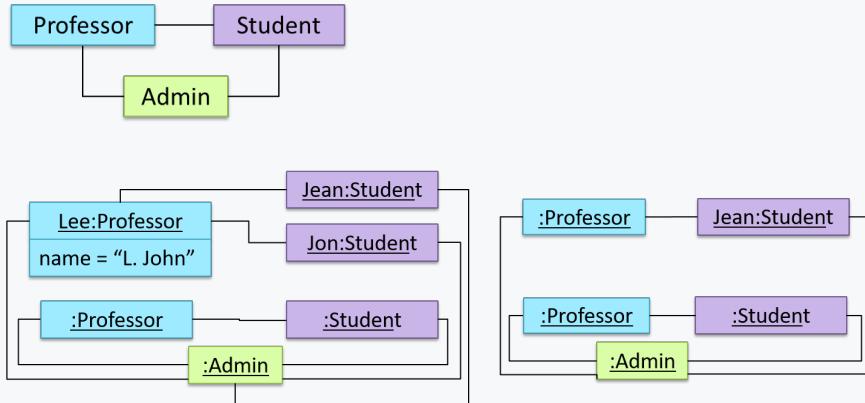
Object vs Class Diagrams ★★☆☆

Compared to the notation for class diagrams, object diagrams differ in the following ways:

- Show objects instead of classes:
 - Instance name may be shown
 - There is a `:` before the class name
 - Instance and class names are underlined
- Methods are omitted
- Multiplicities are omitted. Reason: an association line in an object diagram represents a connection to exactly one object (i.e., the multiplicity is always 1).

Furthermore, **multiple object diagrams can correspond to a single class diagram.**

💡 Both object diagrams are derived from the same class diagram shown earlier. In other words, each of these object diagrams shows 'an instance of' the same class diagram.



When the class diagram has an inheritance relationship, **the object diagram should show either an object of the parent class or the child class, but not both.**

💡 Suppose `Employee` is a child class of the `Person` class. The class diagram will be as follows:

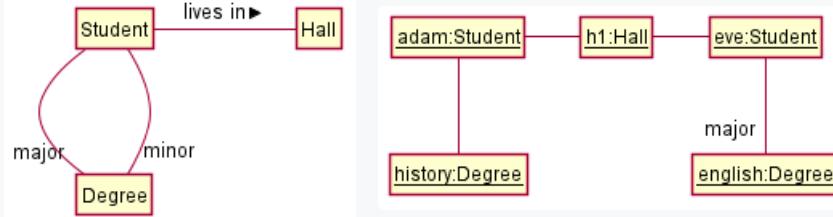


Now, how do you show an `Employee` object named `jake`?

- ✗ This is not correct, as there should be only one object.
- ✓ This is OK.
- ✓ This is OK, as `jake` is a `Person` too. That is, we can show the parent class instead of the child class if the child class doesn't matter to the purpose of the diagram (i.e., the reader of this diagram will not need to know that `jake` is in fact an `Employee`).

Association labels/roles can be omitted unless they add value (e.g., showing them is useful if there are multiple associations between the two classes in concern -- otherwise you wouldn't know which association the object diagram is showing)

💡 Consider this class diagram and the object diagram:



We can clearly see that both Adam and Eve lives in hall h1 (i.e., OK to omit the association label `lives in`) but we can't see if History is Adam's major or his minor (i.e., the diagram should have included either an association label or a role there). In contrast, we can see Eve is an English major.

Git and GitHub

init: Getting started ★★★★

Let's take your first few steps in your Git (with GitHub) journey.

1. The first step is to install SourceTree, which is Git + a GUI for Git. If you prefer to use Git via the command line (i.e., without a GUI), you can [install Git](#) instead.

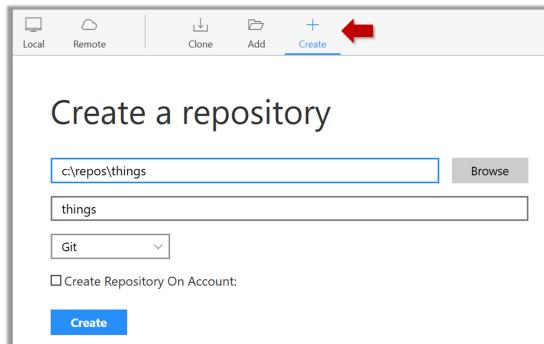
2. Next, initialize a repository. Let us assume you want to version control content in a specific directory. In that case, you need to initialize a Git repository in that directory. Here are the steps:

Create a directory for the repo (e.g., a directory named **things**).



Windows: Click **File** → **Clone/New...**. Click on **Create** button.
Mac: **New...** → **Create New Repository**.

Enter the location of the directory (Windows version shown below) and click **Create**.



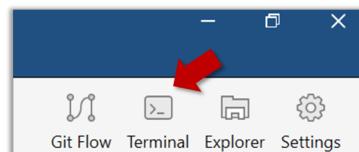
Go to the **things** folder and observe how a hidden folder **.git** has been created.

Windows: you might have to [configure Windows Explorer to show hidden files](#).



Open a Git Bash Terminal.

If you installed SourceTree, you can click the **Terminal** button to open a GitBash terminal.



Navigate to the **things** directory.

Use the command **git init** which should initialize the repo.

```
$ git init  
Initialized empty Git repository in c:/repos/things/.git/
```

You can use the command `ls -a` to view all files, which should show the `.git` directory that was created by the previous command.

```
$ ls -a  
. .. .git
```

You can also use the `git status` command to check the status of the newly-created repo. It should respond with something like the following:

```
git status
```

↓

```
# On branch master  
#  
# Initial commit  
#  
nothing to commit (create/copy files and use "git add" to track)
```

? As you see above, this textbook explains how to use Git via SourceTree (a GUI client) as well as via the Git CLI. If you are new to Git, **we recommend you learn both the GUI method and the CLI method** -- The GUI method will help you visualize the result better while the CLI method is more universal (i.e., you will not be tied to any GUI) and more flexible/powerful.

! If you are new to Git, **we caution you against using Git or GitHub features that come with the IDE** as it is better to learn Git independent of any other tool. Similarly, using clients provided by GitHub (e.g., *GitHub Desktop* GUI client) will make it harder for you to separate Git features from GitHub features.

`commit`: Saving changes to history ★★★★

After initializing a repository, Git can help you with revision controlling files inside the **working directory**. However, it is not automatic. It is up to you to tell Git which of your changes (aka *revisions*) should be *committed* to its memory for later use. Saving changes into Git's memory in that way is often called *committing* and a change saved to the revision history is called a *commit*.

Working directory: the root directory revision-controlled by Git (e.g., the directory in which the repo was initialized).

Commit (noun): a change (aka a *revision*) saved in the Git revision history.
(verb): the act of creating a commit i.e., saving a change in the working directory into the Git revision history.

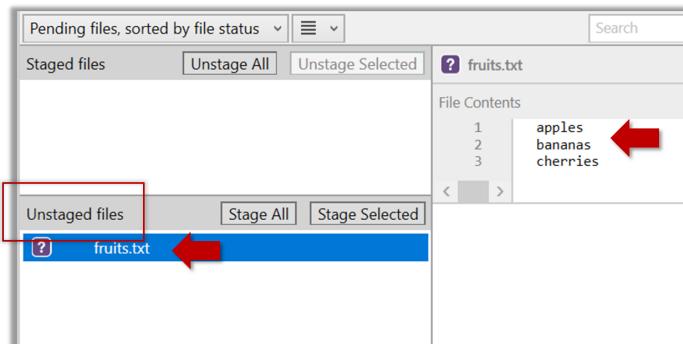
Here are the steps you can follow to learn how to work with Git commits:

1. Do some changes to the content inside the **working directory** e.g., create a file named `fruits.txt` in the `things` directory and add some dummy text to it.

2. Observe how the file is detected by Git.

SourceTree

The file is shown as 'unstaged'.



CLI

You can use the `git status` command to check the status of the working directory.

```
git status
```

↓

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   a.txt
nothing added to commit but untracked files present (use "git add" to track)
```

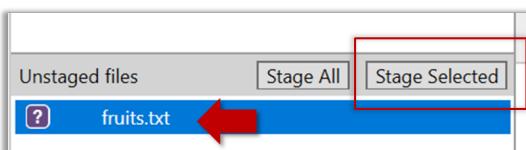
3. Stage the changes to commit: Although Git has detected the file in the working directory, it will not do anything with the file unless you tell it to. Suppose you want to commit the current changes to the file. First, you should *stage* the file.



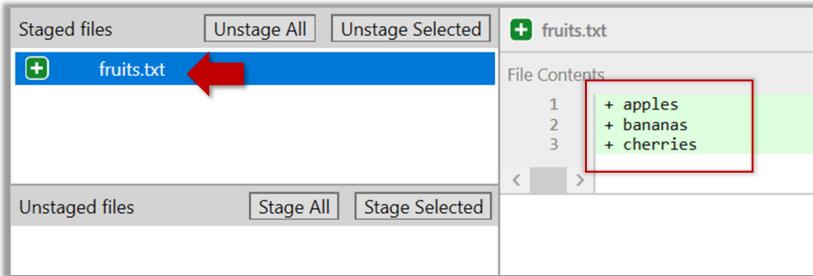
Stage (verb): Instructing Git to prepare a file for committing.

SourceTree

Select the `fruits.txt` and click on the `Stage Selected` button.



`fruits.txt` should appear in the `Staged files` panel now.



CLI

You can use the `stage` or the `add` command (they are synonyms, `add` is the more popular choice) to stage files.

```
git add fruits.txt
git status
```

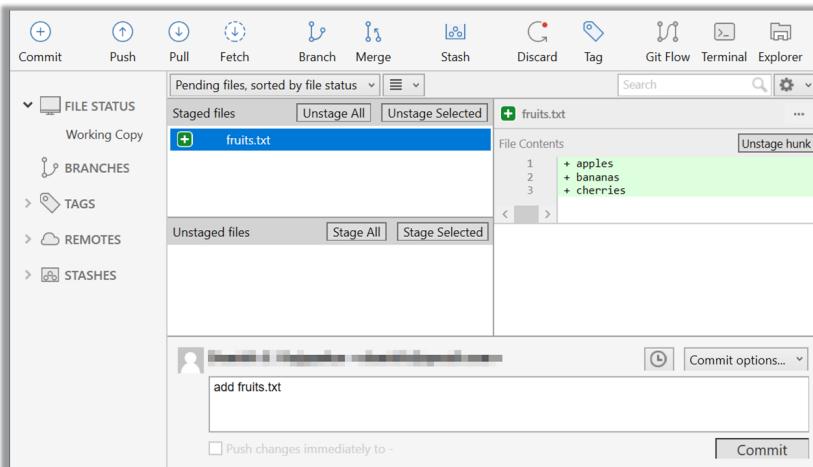
↓

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   fruits.txt
#
```

4. Commit the staged version of `fruits.txt`.

SourceTree

Click the `Commit` button, enter a commit message e.g. `add fruits.txt` into the text box, and click `Commit`.



Use the `commit` command to commit. The `-m` switch is used to specify the commit message.

```
git commit -m "Add fruits.txt"
```

You can use the `log` command to see the commit history.

```
git log
```



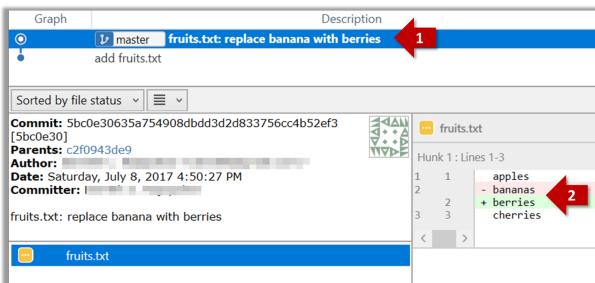
```
commit 8fd30a6910efb28bb258cd01be93e481caeab846
Author: ... <...@...>
Date:   Wed Jul 5 16:06:28 2017 +0800
```

```
Add fruits.txt
```

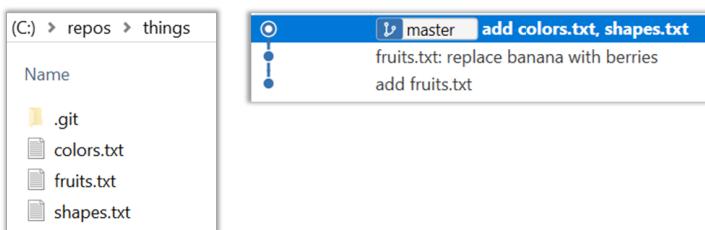
Note the existence of something called the `master` branch. Git allows you to have multiple branches (i.e. it is a way to evolve the content in parallel) and Git auto-creates a branch named `master` on which the commits go on by default.

5. Do a few more commits.

1. Make some changes to `fruits.txt` (e.g. add some text and delete some text). Stage the changes, and commit the changes using the same steps you followed before. You should end up with something like this.



2. Next, add two more files `colors.txt` and `shapes.txt` to the same working directory. Add a third commit to record the current state of the working directory.



6. **See the revision graph:** Note how commits form a path-like structure aka the *revision tree/graph*. In the revision graph, each commit is shown as linked to its 'parent' commit (i.e., the commit before it).

SourceTree

To see the revision graph, click on the `History` item on the menu on the right edge of SourceTree.

CLI

The `gitk` command opens a rudimentary graphical view of the revision graph.

Omitting files from revision control ★★★

Often, there are files inside the Git working folder that you don't want to revision-control e.g., temporary log files. Follow the steps below to learn how to configure Git to ignore such files.

1. Add a file into your repo's working folder that you supposedly don't want to revision-control e.g., a file named `temp.txt`. Observe how Git has detected the new file.

2. Tell Git to ignore that file:

SourceTree

The file should be currently listed under `Unstaged files`. Right-click it and choose `Ignore...`. Choose `Ignore exact filename(s)` and click `OK`.

Observe that a file named `.gitignore` has been created in the working directory root and has the following line in it.

`temp.txt`

CLI

Create a file named `.gitignore` in the working directory root and add the following line in it.

`temp.txt`

The `.gitignore` file

The `.gitignore` file tells Git which files to ignore when tracking revision history. That 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, follow the same steps you followed above when you set Git to ignore the `temp.txt` file.
- It supports file patterns e.g., adding `temp/*.*tmp` to the `.gitignore` file prevents Git from tracking any `.tmp` files in the `temp` directory.

More information about the `.gitignore` file: git-scm.com/docs/gitignore



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
- **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).

tag: Naming commits ★★☆

Each Git commit is uniquely identified by a hash e.g., `d670460b4b4aece5915caf5c68d12f560a9fe3e4`. As you can imagine, using such an identifier is not very convenient for our day-to-day use. As a solution, Git allows adding a more human-readable **tag** to a commit e.g., `v1.0-beta`.

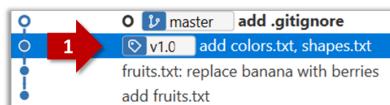
Here's how you can tag a commit in a local repo (e.g. in the `samplerrepo-things` repo):

SourceTree

Right-click on the commit (in the graphical revision graph) you want to tag and choose `Tag...`.

Specify the tag name e.g. `v1.0` and click `Add Tag`.

The added tag will appear in the revision graph view.



CLI

To add a tag to the current commit as `v1.0`,

```
git tag -a v1.0
```

To view tags

```
git tag
```

To learn how to add a tag to a past commit, go to the '[Git Basics – Tagging](#)' page of the [git-scm book](#) and refer the 'Tagging Later' section.

! Remember to push tags to the repo. A normal push does not include tags.

```
# push a specific tag  
git push origin v1.0b  
  
# push all tags  
git push origin --tags
```

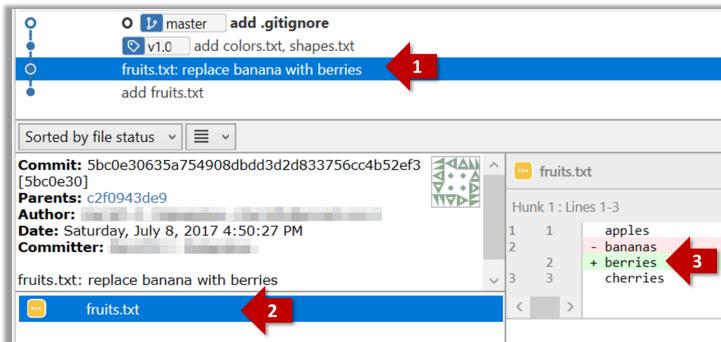
After adding a tag to a commit, you can use the tag to refer to that commit, as an alternative to using the hash.

diff : Comparing revisions ★★★

Git can show you what changed in each commit.

SourceTree

To see which files changed in a commit, click on the commit. To see what changed in a specific file in that commit, click on the file name.



CLI

```
git show < part-of-commit-hash >
```

Example:

```
git show 251b4cf
```

↓

```
commit 5bc0e30635a754908dbdd3d2d833756cc4b52ef3
Author: ... < ... >
Date:   Sat Jul 8 16:50:27 2017 +0800

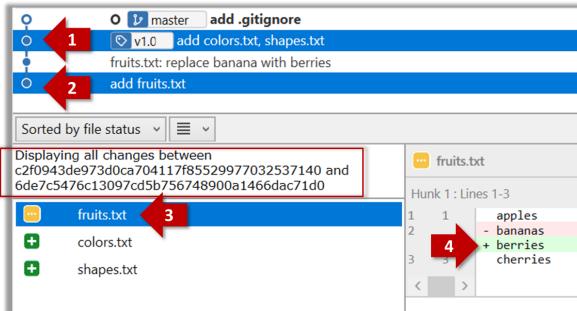
fruits.txt: replace banana with berries

diff --git a/fruits.txt b/fruits.txt
index 15b57f7..17f4528 100644
--- a/fruits.txt
+++ b/fruits.txt
@@ -1,3 +1,3 @@
 apples
-bananas
+berries
 cherries
```

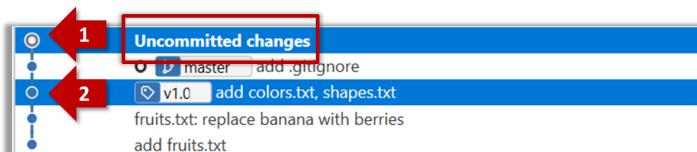
Git can also show you the difference between two points in the history of the repo.

SourceTree

Select the two points you want to compare using `Ctrl` + `Click`. The differences between the two selected versions will show up in the bottom half of SourceTree, as shown in the screenshot below.



The same method can be used to compare the current state of the working directory (which might have uncommitted changes) to a point in the history.



CLI

The `diff` command can be used to view the differences between two points of the history.

- `git diff`: shows the changes (uncommitted) since the last commit.
- `git diff 0023cd..fc6199`: shows the changes between the points indicated by commit hashes.
💡 Note that when using a commit hash in a Git command, you can use only the first few characters (e.g., first 7-10 chars) as that's usually enough for Git to locate the commit.
- `git diff v1.0..HEAD`: shows changes that happened from the commit tagged as `v1.0` to the most recent commit.

Resources

- [Git-Tower Tutorial: Inspecting Changes with Diffs](#)
- [How to view the next page of a `git` command result](#)

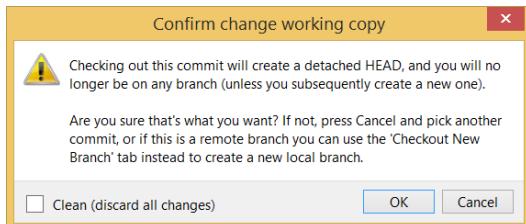
checkout : Retrieving a specific revision ★★★

Git can load a specific version of the history to the working directory. Note that if you have uncommitted changes in the working directory, you need to stash them first to prevent them from being overwritten.

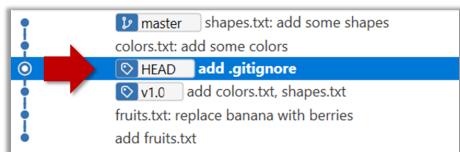
SourceTree

Double-click the commit you want to load to the working directory, or right-click on that commit and choose `Checkout...`.

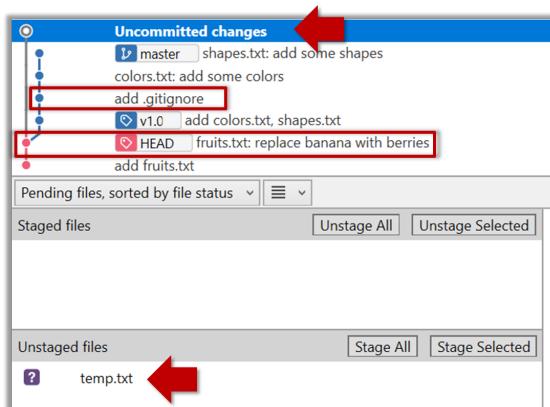
Click `OK` to the warning about 'detached HEAD' (similar to below).



The specified version is now loaded to the working folder, as indicated by the `HEAD` label. `HEAD` is a reference to the currently checked out commit.



If you checkout a commit that comes before the commit in which you added the `.gitignore` file, Git will now show ignored files as 'unstaged modifications' because at that stage Git hasn't been told to ignore those files.



To go back to the latest commit, double-click it.

CLI

Use the `checkout <commit-identifier>` command to change the working directory to the state it was in at a specific past commit.

- `git checkout v1.0`: loads the state as at commit tagged `v1.0`
- `git checkout 0023cdd`: loads the state as at commit with the hash `0023cdd`
- `git checkout HEAD~2`: loads the state that is 2 commits behind the most recent commit

For now, you can ignore the warning about 'detached HEAD'.

`clone` : Copying a repo ★★★★

Given below is an example scenario you can try yourself to learn Git cloning.

Suppose you want to clone the sample repo [samplerepo-things](#) to your computer.

! Note that the URL of the GitHub project is different from the URL you need to clone a repo in that GitHub project. e.g.

GitHub project URL: <https://github.com/se-edu/samplerepo-things>

Git repo URL: <https://github.com/se-edu/samplerepo-things.git> (note the `.git` at the end)

SourceTree

`File` → `Clone / New...` and provide the URL of the repo and the destination directory.

CLI

You can use the `clone` command to clone a repo.

Follow the instructions given [here](#).

`pull`, `fetch`: Downloading data from other repos ★★★

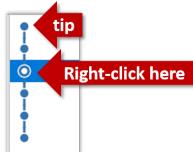
Here's a scenario you can try in order to learn how to `pull` commits from another repo to yours.

1. **Clone a repo** (e.g., the repo used in [Git & GitHub → Clone]) to be used for this activity.

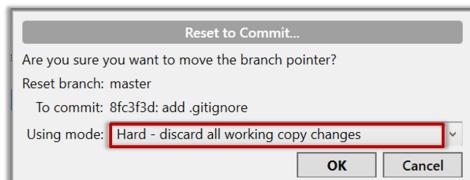
2. **Delete the last few commits to simulate cloning the repo a few commits ago.**

SourceTree

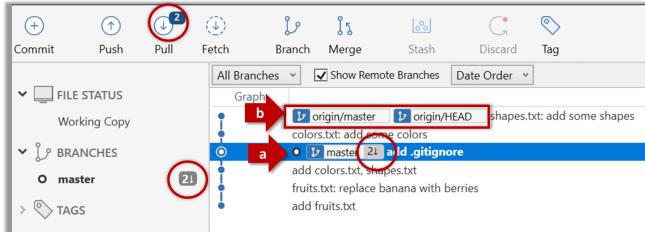
Right-click the target commit (i.e. the commit that is 2 commits behind the tip) and choose `Reset current branch to this commit`.



Choose the `Hard - ...` option and click `OK`.



This is what you will see.



Note the following (cross refer the screenshot above):

Arrow marked as **a**: The local repo is now at this commit, marked by the **master** label.

Arrow marked as **b**: The **origin/master** label shows what is the latest commit in the **master** branch in the remote repo.

CLI

Use the **reset** command to delete commits at the tip of the revision history.

```
git reset --hard HEAD~2
```

Now, your local repo state is exactly how it would be if you had cloned the repo 2 commits ago, as if somebody has added two more commits to the remote repo since you cloned it.

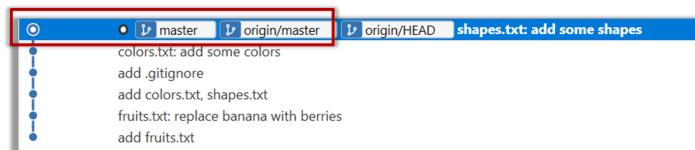
3. Pull from the other repo: To get those missing commits to your local repo (i.e. to sync your local repo with upstream repo) you can do a pull.

SourceTree

Click the **Pull** button in the main menu, choose **origin** and **master** in the next dialog, and click **OK**.



Now you should see something like this where **master** and **origin/master** are both pointing the same commit.



CLI

```
git pull origin
```

i You can also do a `fetch` instead of a `pull` in which case the new commits will be downloaded to your repo but the working directory will remain at the current commit. To move the current state to the latest commit that was downloaded, you need to do a `merge`. A `pull` is a shortcut that does both those steps in one go.

Working with multiple remotes

When you clone a repo, Git automatically adds a **remote repo named `origin`** to your repo configuration. As you know, you can pull commits from that repo. As you know, a Git repo can work with remote repos other than the one it was cloned from.

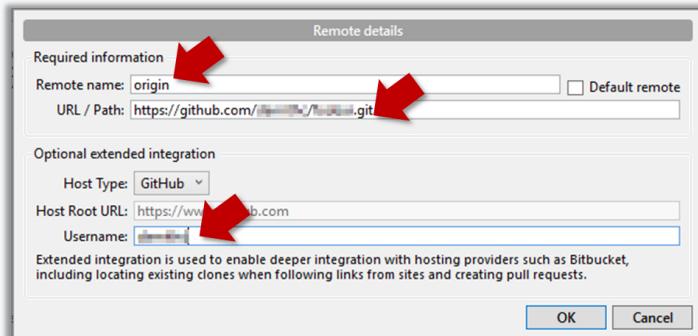
To communicate with another remote repo, you can first add it as a **remote of your repo**. Here is an example scenario you can follow to learn how to pull from another repo:

SourceTree

1. Open the local repo in SourceTree. Suggested: Use your local clone of the `samplerrepo-things` repo.

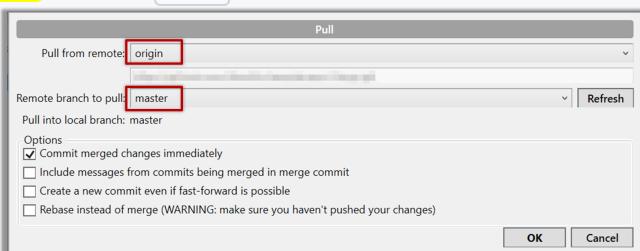
2. Choose `Repository` → `Repository Settings` menu option.

3. Add a new `remote` to the repo with the following values.



- o `Remote name`: the name you want to assign to the remote repo e.g., `upstream1`
- o `URL/path`: the URL of your repo (ending in `.git`) that. Suggested: <https://github.com/se-edu/samplerrepo-things-2.git> (`samplerrepo-things-2` is another repo that has a shared history with `samplerrepo-things`)
- o `Username`: your GitHub username

4. Now, you can pull from the added repo as you did before **but choose the remote name of the repo you want to pull from** (instead of `origin`):



? If the `Remote branch to pull` dropdown is empty, click the `Refresh` button on its right.

5. If the pull from the `samplerrepo-things-2` was successful, you should have received one more commit into your local repo.

CLI

1. Navigate to the folder containing the local repo.

2. Set the new remote repo as a *remote* of the local repo.

command: `git remote add {remote_name} {remote_repo_url}`

e.g., `git remote add upstream1 https://github.com/johndoe/foobar.git`

3. Now you can pull from the new remote.

e.g., `git pull upstream1 master`

Fork: Creating a remote copy ★★★★

Given below is a scenario you can try in order to learn how to fork a repo:

0. Create a GitHub account if you don't have one yet.

1. Go to the GitHub repo you want to fork e.g., [samplerepo-things](#)

2. Click on the  button on the top-right corner. In the next step, choose to fork to your own account or to another GitHub organization that you are an admin of.

ⓘ 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.

push: Uploading data to other repos ★★★★

Given below is a scenario you can try in order to learn how to push commits to a remote repo hosted on GitHub:

1. Fork an existing GitHub repo (e.g., [samplerepo-things](#)) to your GitHub account.

2. Clone the fork (not the original) to your computer.

3. Commit some changes in your local repo.

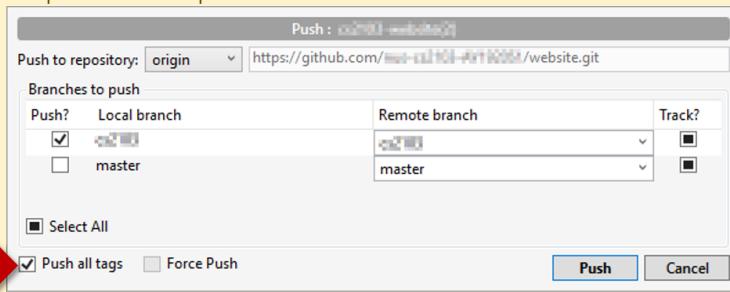
4. Push the new commits to your fork on GitHub

SourceTree

Click the **Push** button on the main menu, ensure the settings are as follows in the next dialog, and click the **Push** button on the dialog.



! Tags are not included in a normal push. Remember to tick `Push all tags` when pushing to the remote repo if you want them to be pushed to the repo.



CLI

Use the command `git push origin master`. Enter your Github username and password when prompted.

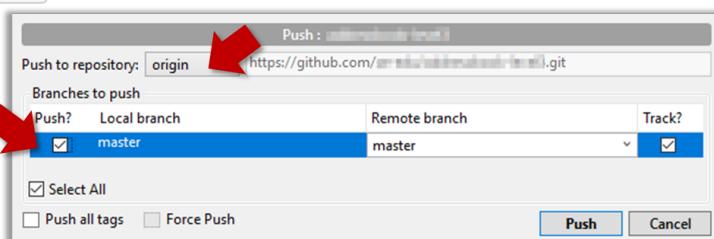
! Tags are not included in a normal push. To push a tag, use this command: `git push origin <tag_name>` e.g. `git push origin v1.0`

You can push to repos other than the one you cloned from, as long as the target repo and your repo have a shared history.

1. Add the GitHub repo URL as a remote, if you haven't done so already.
2. Push to the target repo.

SourceTree

Push your repo to the new remote the usual way, but select the name of target remote instead of `origin` and remember to select the `Track` checkbox.



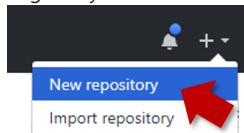
CLI

Push to the new remote the usual way e.g., `git push upstream1 master` (assuming you gave the name `upstream1` to the remote).

You can even push an entire local repository to GitHub, to form an entirely new remote repository. For example, you created a local repo and worked with it for a while but now you want to upload it onto GitHub (as a backup or to share it with others). The steps are given below.

1. Create an **empty** remote repo on GitHub.

1. Login to your GitHub account and choose to create a new Repo.



2. In the next screen, provide a name for your repo but keep the **Initialize this repo ...** tick box unchecked.

The screenshot shows the 'Create repository' form on GitHub. A red arrow points to the 'Repository name' field where 'foobar' is entered. Another red arrow points to the 'Initialize this repository with a README' checkbox, which is unchecked.

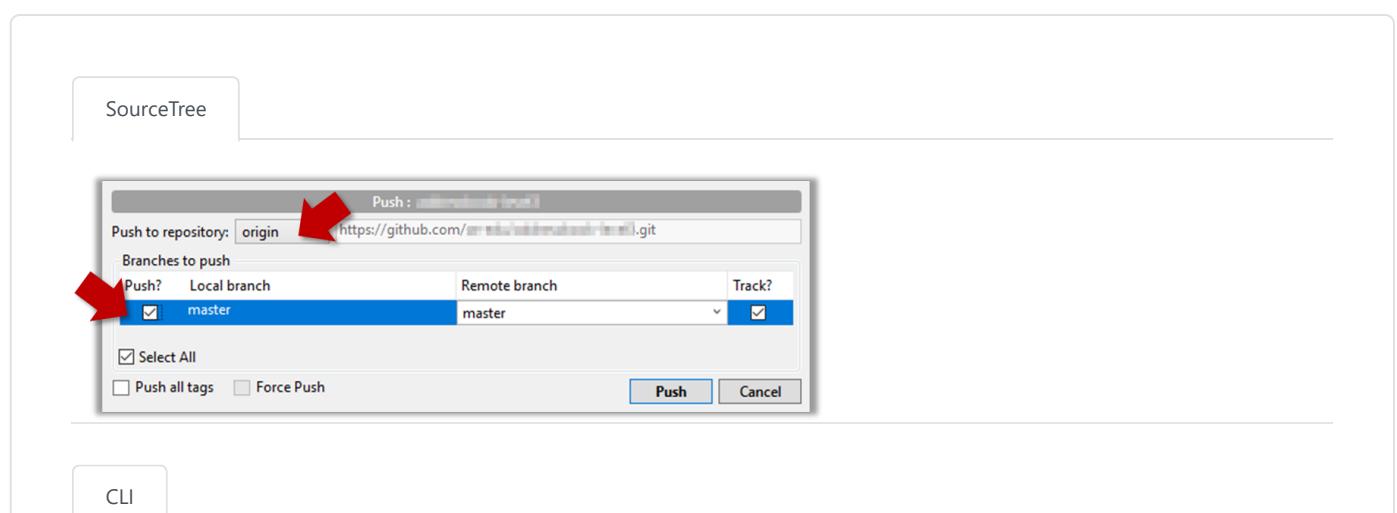
Owner: [dropdown] / Repository name: foobar ✓
Great repository names are short and memorable. Need inspiration? How about studious-carnival.
Description (optional):
Public: Anyone can see this repository. You choose who can commit.
Private: You choose who can see and commit to this repository.
Initialize this repository with a README: This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.
Add .gitignore: None | Add a license: None | ⓘ
Create repository

3. Note the URL of the repo. It will be of the form https://github.com/{your_user_name}/{repo_name}.git. e.g., <https://github.com/johndoe/foobar.git> (note the `.git` at the end)



2. Add the **GitHub repo URL** as a remote of the local repo. You can give it the name `origin` (or any other name).

3. Push the repo to the remote.



Push each branch to the new remote the usual way but use the `-u` flag to inform Git that you wish to track the branch.

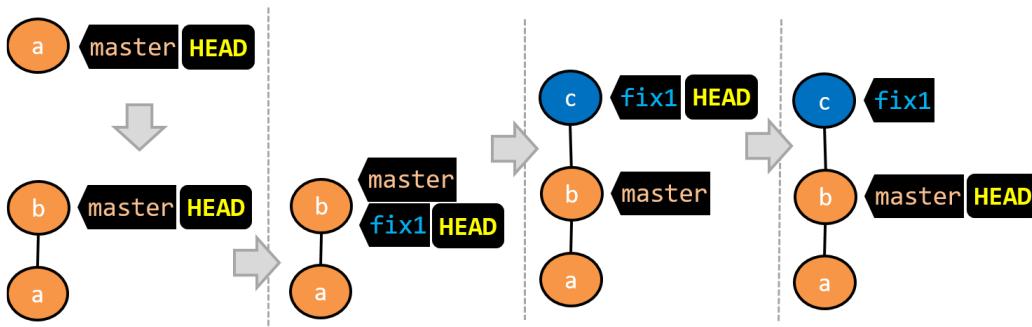
e.g., `git push -u origin master`

branch : Doing multiple parallel changes ★★★

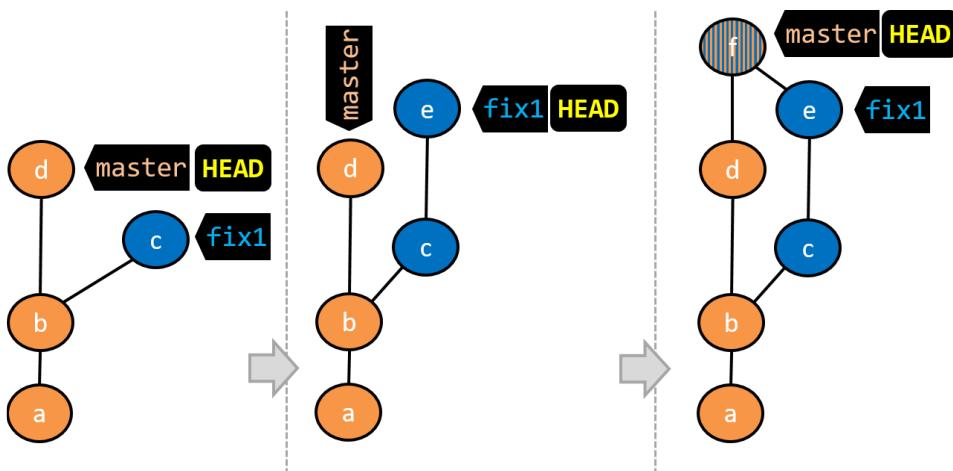
Git supports branching, which allows you to do multiple parallel changes to the content of a repository.

A Git branch is simply a **named label pointing to a commit**. The `HEAD` label indicates which branch you are on. Git creates a branch named `master` by default. When you add a commit, it goes into the branch you are currently on, and the branch label (together with the `HEAD` label) moves to the new commit.

Given below is an illustration of how branch labels move as branches evolve.



1. There is only one branch (i.e., `master`) and there is only one commit on it.
2. A new commit has been added. The `master` and the `HEAD` labels 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` label is attached to the `fix1` branch).
4. A new commit (`c`) has been added. The current branch label `fix1` moves to the new commit, together with the `HEAD` label.
5. The repo has switched back to the `master` branch.

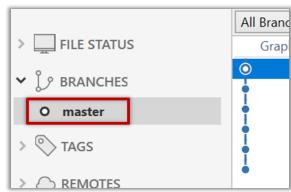


6. A new commit (`d`) has been added. The `master` label has moved to that commit.
7. The repo has switched back to the `fix1` branch and added a new commit (`e`) to it.
8. The repo has switched to the `master` branch and the `fix1` branch has been merged into the `master` branch, creating a *merge commit* (`f`). The repo is currently on the `master` branch.

Follow the steps below to learn how to work with branches. You can use any repo you have on your computer (e.g. a clone of the [samplerepo-things](#)) for this.

0. Observe that you are normally in the branch called `master`.

SourceTree



CLI

```
git status
```

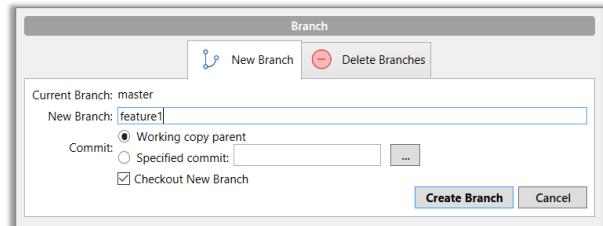
↓

```
on branch master
```

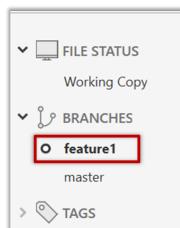
1. Start a branch named `feature1` and switch to the new branch.

SourceTree

Click on the `Branch` button on the main menu. In the next dialog, enter the branch name and click `Create Branch`.



Note how the `feature1` is indicated as the current branch.



CLI

You can use the `branch` command to create a new branch and the `checkout` command to switch to a specific branch.

```
git branch feature1  
git checkout feature1
```

One-step shortcut to create a branch and switch to it at the same time:

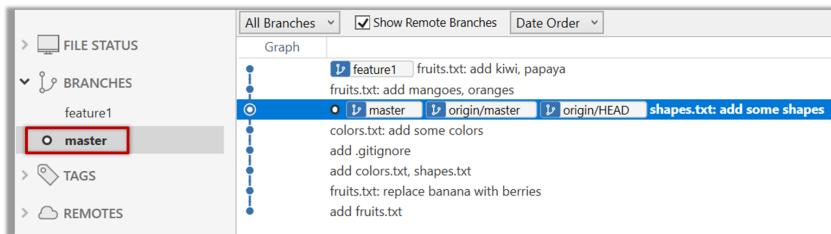
```
git checkout -b feature1
```

2. Create some commits in the new branch. Just commit as per normal. Commits you add while on a certain branch will become part of that branch. Note how the **master** label and the **HEAD** label moves to the new commit (The **HEAD** label of the local repo is represented as **●** in SourceTree).

3. Switch to the **master branch.** Note how the changes you did in the **feature1** branch are no longer in the working directory.

SourceTree

Double-click the **master** branch.

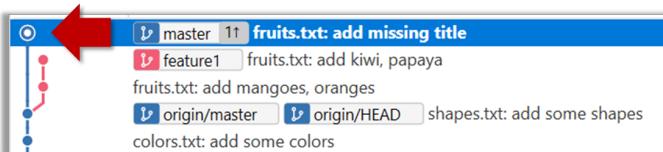


CLI

```
git checkout master
```

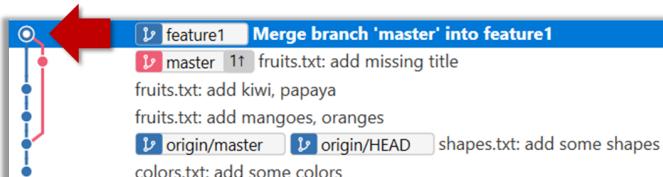
4. Add a commit to the **master branch.** Let's imagine it's a bug fix.

To keep things simple for the time being, this commit should **not involve the same content that you changed in the **feature1** branch**. To be on the safe side, this commit can change an entirely different file.



5. Switch back to the **feature1 branch** (similar to step 3).

6. Merge the **master branch to the **feature1** branch**, giving an end-result like the following. Also note how Git has created a *merge commit*.



SourceTree

Right-click on the `master` branch and choose `merge master into the current branch`. Click `OK` in the next dialog.

CLI

```
git merge master
```

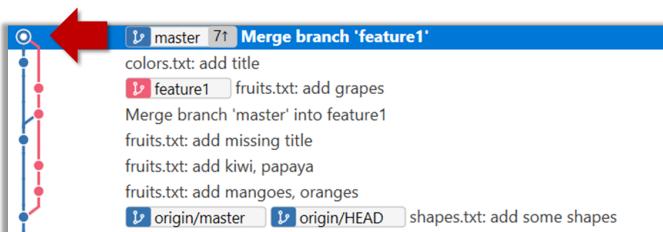
The objective of that merge was to sync the `feature1` branch with the `master` branch. Observe how the changes you did in the `master` branch (i.e. the imaginary bug fix) is now available even when you are in the `feature1` branch.

i Instead of merging `master` to `feature1`, an alternative is to *rebase* the `feature1` branch. However, rebasing is an advanced feature that requires modifying past commits. If you modify past commits that have been pushed to a remote repository, you'll have to *force-push* the modified commit to the remote repo in order to update the commits in it.

7. Add another commit to the `feature1` branch.

8. Switch to the `master` branch and add one more commit.

9. Merge `feature1` to the `master` branch, giving an end-result like this:



SourceTree

Right-click on the `feature1` branch and choose `Merge...`.

CLI

```
git merge feature1
```

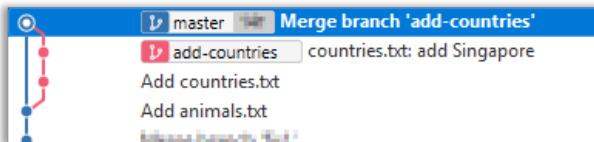
10. Create a new branch called `add-countries`, switch to it, and add some commits to it (similar to steps 1-2 above). You should have something like this now:



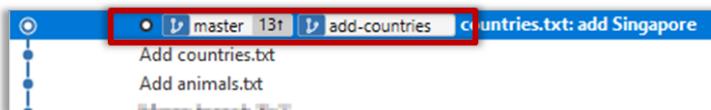
X **Avoid this common rookie mistake!**

Always remember to switch back to the `master` branch before creating a new branch. If not, your new branch will be created on top of the current branch.

11. Go back to the `master` branch and merge the `add-countries` branch onto the `master` branch (similar to steps 8-9 above). While you might expect to see something like the following,



... you are likely to see something like this instead:

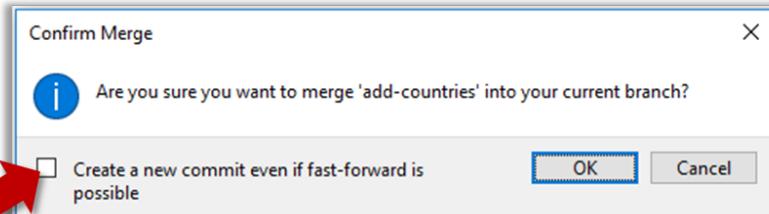


That is because **Git does a *fast forward merge* if possible**. Seeing that the `master` branch has not changed since you started the `add-countries` branch, Git has decided it is simpler to just put the commits of the `add-countries` branch in front of the `master` branch, without going into the trouble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

It is possible to force Git to create a merge commit even if fast forwarding is possible.

SourceTree

Tick the box shown below when you merge a branch:



CLI

Use the `--no-ff` switch (short for *no fast forward*):

```
git merge --no-ff add-countries
```

Pushing a branch to a remote repo

Here's how to push a branch to a remote repo:

SourceTree

Here's how to push a branch named `add-intro` to your own fork of a repo named `samplerrepo-pr-practice`:

The screenshot shows the SourceTree interface with a 'Push' dialog open. The dialog has the following details:

- Push to repository: origin <https://github.com/john/samplerrepo-pr-practice.git>
- Branches to push:

Push?	Local branch	Remote branch	Track?
<input checked="" type="checkbox"/>	add-intro	add-intro	<input checked="" type="checkbox"/>
<input type="checkbox"/>	master	master	<input type="checkbox"/>
- Buttons: Push, Cancel

Red numbered arrows indicate the steps: 1 points to the Push button, 2 points to the URL input field with a placeholder 'Your GitHub username', 3 points to the 'add-intro' checkbox in the branches list, and 4 points to the 'Push' button.

CLI

Normally: `git push {remote repository} {branch}`. Examples:

- `git push origin master` pushes the `master` branch to the repo named `origin` (i.e., the repo you cloned from)
- `git push upstream-repo add-intro` pushes the `add-intro` branch to the repo named `upstream-repo`

If pushing a branch you created locally to the remote for the *first time*, add the `-u` flag to get the local branch to track the new upstream branch:
e.g., `git push -u origin add-intro`

See git-scm.com/docs/git-push for details of the `push` command.

Dealing with merge conflicts ★★★★

Merge conflicts happen when you try to combine two incompatible versions (e.g., merging a branch to another but each branch changed the same part of the code in a different way).

Here are the steps to simulate a merge conflict and use it to learn how to resolve merge conflicts.

0. Create an empty repo or clone an existing repo, to be used for this activity.

1. Start a branch named `fix1` in the repo. Create a commit that adds a line with some text to one of the files.

2. Switch back to `master` branch. Create a commit with a conflicting change i.e. it adds a line with some different text in the exact location the previous line was added.

```
Commit: 5b0e246d0c981d6b1a9d2332467bd1155c5b83be [5b0e246]
colors.txt
Hunk 1 : Lines 1-6
1 1 | COLORS
2 2 | -----
3 3 | blue
4 4 + green
5 5 | red
```

```
Commit: 5d45a8324b4d0050b2bb87704eb1d1b3ae46c01a [5d45a83]
colors.txt
Hunk 1 : Lines 1-6
1 1 | COLORS
2 2 | -----
3 3 | blue
4 4 + black
5 5 | red
```

3. Try to merge the `fix1` branch onto the `master` branch. Git will pause mid-way during the merge and report a merge conflict. If you open the conflicted file, you will see something like this:

```
COLORS
-----
blue
<<<< HEAD
black
=====
green
>>>> fix1
red
white
```

4. Observe how the conflicted part is marked between a line starting with `<<<<` and a line starting with `>>>>`, separated by another line starting with `=====`.

Highlighted below is the conflicting part that is coming from the `master` branch:

```
3 blue
4 <<<< HEAD
5 black
6 =====
7 green
8 >>>> fix1
9 red
```

This is the conflicting part that is coming from the `fix1` branch:

```
3 blue
4 <<<< HEAD
5 black
6 =====
7 green
8 >>>> fix1
9 red
```

5. Resolve the conflict by editing the file. Let us assume you want to keep both lines in the merged version. You can modify the file to be like this:

```
COLORS
-----
```

```
blue  
black  
green  
red  
white
```

6. Stage the changes, and commit.

Creating PRs ★★★☆

Suppose you want to propose some changes to a GitHub repo (e.g., [samplerepo-pr-practice](#)) as a [pull request \(PR\)](#). Here is a scenario you can try in order to learn how to create PRs:

1. **Fork** the repo onto your GitHub account.

2. **Clone** it onto your computer.

3. **Commit** your changes e.g., add a new file with some contents and commit it.

- **Option A - Commit changes to the `master` branch**
- **Option B - Commit to a new branch** e.g., create a branch named `add-intro` (remember to switch to the `master` branch before creating a new branch) and add your commit to it.

4. **Push** the branch you updated (i.e., `master` branch or the new branch) to your fork, as explained [here](#).

5. Initiate the PR creation:

1. Go to your fork.

2. Click on the `Pull requests` tab followed by the `New pull request` button. This will bring you to the 'Comparing changes' page.

3. Set the appropriate target repo and the branch that should receive your PR, using the `base repository` and `base` dropdowns. e.g.,

base repository: `se-edu/samplerepo-pr-practice` ▾ base: `master` ▾

i Normally, the default value shown in the dropdown is what you want but in case your fork has multiple upstream repos, the default may not be what you want.

4. Indicate which repo:branch contains your proposed code, using the `head repository` and `compare` dropdowns. e.g.,

head repository: `myrepo/samplerepo-pr-practice` ▾ compare: `master` ▾

6. Verify the proposed code: Verify that the diff view in the page shows the exact change you intend to propose. If it doesn't, [update the branch](#) as necessary.

7. Submit the PR:

1. Click the `Create pull request` button.

2. Fill in the PR name and description e.g.,

Name: `Add an introduction to the README.md`

Description:

Add some paragraph to the `README.md` to explain ...
Also add a heading ...

3. If you want to indicate that the PR you are about to create is 'still work in progress, not yet ready', click on the dropdown arrow in the `Create pull request` button and choose `Create draft pull request` option.

4. Click the `Create pull request` button to create the PR.

5. Go to the receiving repo to verify that your PR appears there in the [Pull requests](#) tab.

The next step of the PR life cycle 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 [master](#) branch, 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.

To create another PR while the current PR is still under review, create a new branch (remember to [switch back to the \[master\]\(#\) branch first](#)), add your new proposed change in that branch, and create a new PR following the steps given above.

It is possible to create PRs within the same repo 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.

 Resources

- [GitHub's own documentation on creating a PR](#)



abstract classes 13, 34, 81
abstraction 5, 27, 30
acceptance testing 59-60
aggregation 11, 33-34, 79-80
agile models 68
alternative paths 35, 85
apis 51, 60
assertions 48-49
associations 7-9, 32-34, 75, 78, 89-90
associations as attributes 78
availability 70
avoid complicated expressions 38-39
avoid deep nesting 38
avoid long methods 37
avoid magic numbers 39
benefit 10, 20-22, 43, 51, 59
brainstorming 18, 22-23
branching 66, 108
buffers 70
build automation 36, 50
business analyst 26, 30
class diagrams 30-34, 73, 75, 79, 89
class diagrams (basics) 32
class inheritance 13, 80
class level members 6
class-level members 7, 32, 79
client 10, 23, 51, 93
code quality -1, 37, 40-41
code reviews 54
cohesion 28, 72
complexity 27
compliance 16, 55
composition 10-11, 33, 79-80
configuration 104
constraint 16
context 5, 9-10, 20-21, 41, 59, 72
coupling 27-28, 72
creating prs 115
customer 20, 22, 30, 60, 68
debugging 2, 36, 52
defect 36, 54, 56
dependencies 9, 33-34, 51, 70, 77-78
dependency 9, 50-52, 78
design fundamentals -1, 27
developer testing 58-59
documentation -1, 16, 19, 30, 41, 45, 52, 68, 116
dogfooding 58
domain 16, 18, 30, 40
effects 60
efficiency 16, 37
encapsulation of objects 5
entity 30, 72
enumerations 7, 34, 40, 78-79
error handling -1, 47
estimate 21-22, 69-70
evaluation 11-12, 56
exception handling 47
exceptions 47, 49, 55
extreme programming 68
failure 48-49, 56, 58
feature lists 20
focus groups 18
formal verification 55
frameworks 51-53
frameworks vs libraries 52
functional requirement 16-17, 22, 25, 58
functionality 17, 21, 24, 28, 30, 37, 52, 54, 68, 72
gathering requirements -1, 18, 23
git and github 0, 92
glossary 1, 25
goal 1
high-level 21, 30, 70
impact 48
implementing aggregation 11
implementing associations 7
implementing composition 10
implementing multiplicity 8
inheritance 11-14, 33-34, 78, 80, 90
inspection 54
inspector 54
integration -1, 27, 50, 57
integration testing 57
interfaces 13, 34, 80
interoperability 16, 58
interview 18
interviews 18
issue trackers 70
iterative models 67
javadoc 45-46
labels 32, 76, 90, 108
libraries 50-53, 62
logging 49
loops 35, 84
maintainability 16, 28
maintenance 1, 27
measure 2, 27-28, 69
milestone 70
milestones 70
minimal notation 35, 88
miscellaneous 15, 89
modeling -1, 10, 30-31, 33-34, 36
modeling abstract classes 34
modeling aggregation 33
modeling behaviors 34
modeling composition 33
modeling dependencies 34
modeling enumerations 34
modeling inheritance 33
modeling interfaces 34
modeling structures 31
moderator 54
module 7, 27-28, 53
multiplicity 8, 32, 77-78, 90
navigability 7, 32, 75
non-functional requirement 16, 22, 58
non-functional requirements 16, 22, 58
notes 16, 21, 33, 41, 83, 89
object creation 35, 83
object deletion 35, 84
object diagram 31, 34, 75-76, 83, 88-90
object diagrams 31, 34, 75, 83, 88-90
object-oriented programming -1, 3-5
objects as abstractions 5
observation 18, 59
oo structures 31
operability 16, 58
optional paths 35, 86
output 1, 27, 56, 61-62
overloading 12, 15
overriding 12, 15
pair programming 54
parallel paths 86-87
path 35, 37, 85-87, 96, 104
performance 16, 43, 48, 51, 55, 58
persona 22, 98
platforms 51, 53-54, 58
polymorphism 14-15
portability 16, 58
principles 0-1, 72
priority 7, 20-21
process model 0, 67-68
process requirement 16
product requirement 18
product surveys 18
project planning 0, 69
pros and cons 1
prose 20
prototype 18, 81-82
prototyping 18
quality -1, 16, 37, 40-41, 43, 54, 58
quality assurance -1, 54
refactoring -1, 43-44
reference frames 87
regression testing 60-61
release 50, 58, 66, 68, 70
reliability 16
remote repositories 65
repositories 64-66
requirements specification 22, 60
reuse -1, 27, 51-52
review 18, 54, 56, 116
reviewer 54, 116
revision control 0, 50, 64-65, 93, 97-98
risk 22, 36, 51
roles 4, 24, 32, 54, 76, 90
safety 48, 55
saving history 65
scalability 16, 58
scope 16, 22, 64
scribe 20, 24, 40, 67, 73, 76
scrum 68
sdlc process models 0, 67
security 16-17, 37, 51, 53, 58, 72
self invocation 85
sequence diagram 34-35, 81-83, 87-88
sequence diagrams 34-35, 81, 83, 87
sequential models 67
single responsibility principle 72
software design -1, 26, 30
software engineering -1, 1, 13, 20, 26, 51, 68
software quality 54
specification 13, 15, 18, 22, 55-58, 60
specifying requirements -1, 18, 20
stability 16
staging and committing 65
stakeholder 16-18, 22, 25, 30
standard 1, 40-41, 46, 54
static analysis 54-55
structure code logically 40
substitutability 14
supplementary requirements 25
system testing 57-58, 60, 69
test automation 61-62
test automation tools 62
testability 16
testing types 56
tracking and ignoring 65
understandability 28, 37
unit testing 56-57
usability 28, 58
usage 18, 22
use case 24-25, 30, 60
use case diagram 24, 30
use cases 24-25
use standard words 41
user stories 20-24
user surveys 18
using history 65
validation 40, 54
validation vs verification 54
verification 54-55
version control 64, 92, 97-98
work breakdown structure 69