Software Engineering

Introduction

Pros and Cons

★★★☆ **Can explain pros and cons of software engineering**

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, we want others to use our 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 we 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....

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

Buildings	Software
Visible, tangible	Invisible, intangible
Wears out over time	Does not wear out
Change is limited by physical restrictions (e.g. difficult to remove a floor from a high rise building)	Change is not limited by such restrictions. Just change the code and recompile.
Creating an exact copy of a building is impossible. Creating a near copy is almost as costly as creating the original.	Any number of exact copies can be made with near zero cost.
Difficult to move.	Easily delivered from one place to another.
Many low-skilled workers following tried-and-tested procedures.	No low-skilled workers involved. Workers have more freedom to follow their own procedures.
Easier to assure quality (just follow accepted procedure).	Not easy to assure quality.
Majority of the work force has to be on location.	Can be built by people who are not even in the same country.

Buildings	Software
Raw materials are costly, costly equipment required.	Almost free raw materials and relatively cheap equipment.
Once construction is started, it is hard to do drastic changes to the design.	Building process is very flexible. Drastic design changes can be done, although costly
A lot of manual and menial labor involved.	Most work involves highly-skilled labor.
Generally robust. E.g. removing a single brick is unlikely to destroy a building.	More fragile than buildings. A single misplaced semicolon can render the whole system useless.

write your answer here...

Comment on this statement: Building software is cheaper and easier than building bridges (all we need is a PC!).

Depends on the size of the software. Manpower required for software is very costly. On the other hand, we can create a very valuable software (e.g. an iPhone application that can make million dollars in a month) with a just a PC and a few days of work!

write your answer here...

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

Arguments to support this statement:

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

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

write your answer here...

List some (at least three each) pros and cons of Software Engineering compared to other traditional Engineering careers.

write your answer here...

- [ ] a. Need for perfection when developing software
- [ ] b. Requiring some amount of tedious, painstaking labor
- [ ] c. Ease of copying and transporting software makes it difficult to keep track of versions
- [ ] d. High dependence on others
- [ ] e. Seemingly never ending effort required for testing and debugging software
- [ ] f. Fast moving industry making our work obsolete quickly

(c)

This site is not ready yet! The updated version will be available soon.

Week 1 [Jan 13] - Topics

- Topics allocated to the week will appear in this tab.
- If the lecture is in the 2nd half of the week (i.e., Wednesday 12 noon or later), the lecture in week `N` will cover topics allocated to the week `N+1` e.g., **Lecture 1 will cover Week 2 topics**, and so on.

[W1.1] OOP: Classes & Objects

W1.1a : ★

Paradigms → OOP → Introduction → What

Can describe OOP at a higher level

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 we 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#, Haskel, 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.

A) Choose the correct statements

- a. OO is a programming paradigm
- b. OO guides us in how to structure the solution
- c. OO is mainly an abstraction mechanism
- d. OO is a programming language
- e. OO is modeled after how the objects in real world work

B) Choose the correct statements

- a. Java and C++ are OO languages
- b. C language follows the *Functional Programming* paradigm
- c. Java can be used to write procedural code
- d. Prolog follows the Logic Programming paradigm

A) (a)(b)(c)(e)

Explanation: While many languages support the OO paradigm, OO is not a language itself.

B) Choose the correct statement

(a)~~(b)~~(c)(d)

Explanation: C follows the procedural paradigm. Yes, we can write procedural code using OO languages e.g., AddressBook-level1.

OO is a higher level mechanism than the procedural paradigm.

- True
- False

True.

Explanation: Procedural languages work at simple data structures (e.g., integers, arrays) and functions level. Because an object is an abstraction over data+related functions, OO works at a higher level.

W1.1b : ★★

Paradigms → OOP → Objects → What

Can describe how OOP relates to the real world

> Every object has both state (data) and behavior (operations on data). In that, they're not much different from ordinary physical objects. It's easy to see how a mechanical device, such as a pocket watch or a piano, embodies both state and behavior. But almost anything that's designed to do a job does, too. Even simple things with no moving parts such as an ordinary bottle combine state (how full the bottle is, whether or not it's open, how warm its contents are) with behavior (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand high or low temperatures).
>
> It's this resemblance to real things that gives objects much of their power and appeal. They can not only model components of real systems, but equally as well fulfill assigned roles as components in software systems.
>
> -- Object-Oriented Programming with Objective-C, Apple

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 a 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, we 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 message 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

Consider the following real-world scenario.

> Tom read a Software Engineering textbook (he has been assigned to read the book) and highlighted some of the text in it.

Explain the following statements about OOP using the above scenario as an example.

1. Object Oriented Programming (OOP) views the world as a network of interacting objects.
2. Every object has both state (data) and behavior (operations on data).
3. Every object has an interface and an implementation.
4. Objects interact by sending messages.
5. OOP does not demand that the virtual world object network follow the real world exactly.

> [1] Object Oriented Programming (OOP) views the world as a network of interacting objects.

Interacting objects in the scenario: `Tom` , `SE Textbook` (`Book` for short), `Text` , (possibly) `Highlighter`

(💡) objects usually match nouns in the description

> [2]Every object has both state (data) and behavior (operations on data).

Object	Examples of state	Examples of behavior
`Tom`	memory of the text read	read
`Book`	title	show text
`Text`	font size	get highlighted

> [3] Every object has an interface and an implementation.

- Interface of an object consists of how other objects interact with it i.e., what other objects can do to that object
- Implementation consist of internals of the object that facilitate the interactions but not visible to other objects.

Object	Examples of interface	Examples of implementation
`Tom`	receive reading assignment	understand/memorize the text read, remember the reading assignment
`Book`	show text, turn page	how pages are bound to the spine
`Text`	read	how characters/words are connected together or fixed to the book

> [4] Objects interact by sending messages.

Examples:

- `Tom` sends message `turn page` to the `Book`
- `Tom` sends message `show text` to the `Book`. When the `Book` shows the `Text`, `Tom` sends the message `read` to the `Text` which returns the text content to `Tom`.
- `Tom` sends message `highlight` to the `Highlighter` while specifying which `Text` to highlight. Then the `Highlighter` sends the message `highlight` to the specified `Text`.

> [5] OOP does not demand that the virtual world object network follow the real world exactly.

Examples:

- A virtual world simulation of the above scenario can omit the `Highlighter` object. Instead, we can teach `Text` to highlight themselves when requested.

write your answer here...

W1.1c : ★★

Paradigms → OOP → Objects → Objects as Abstractions

Can explain the abstraction aspect of OOP

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.

> We 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.

W1.1d : ★★

Paradigms → OOP → Objects → Encapsulation of Objects

Can explain the encapsulation aspect of OOP

> 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.

Choose the correct statements

- a. An object is an encapsulation because it packages data and behavior into one bundle.
- b. An object is an encapsulation because it lets us think in terms of higher level concepts such as Students rather than student-related functions and data separately.

Don't confuse encapsulation with abstraction.

Choose the correct statement

(a)

Explanation: The second statement should be: An object is an **abstraction** ~~encapsulation~~ because it lets ...

W1.1e

Paradigms → OOP → Classes → What

Can explain the relationship between classes and objects

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 we 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 that same instructions can be used to *instantiate* objects of that kind. We call such instructions a *Class*.

Classes and objects in an example scenario

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

Instructions for creating objects `Adam`, `Beth`, `Charlie`, and `Daisy` will be very similar because they are all of the same kind: they all represent ‘persons’ with the same interface, the same kind of data (i.e. `name`, `dateOfBrith`, etc.), and the same kind of behavior (i.e. `getAge(Date)`, `getName()`, etc.). Therefore, we 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, we need classes `AgeList`, `Calculator`, and `Main` classes to instantiate one each of `AgeList`, `Calculator`, and `Main` objects.

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

Consider the following scenario. If you were to simulate this in an OOP program, what are the classes and the objects you would use?

> A customer (name: John) gave a cheque to the Cashier (name: Peter) to pay for the LoTR and GoT books he bought.

Class	Objects
`Customer`	`john`
`Book`	`LoTR` , `GoT`
`Cheque`	`checqueJohnGave`
`Cashier`	`peter`

Assume you are writing a CLI program called **`CityConnect`** for storing and querying distances between cities. The behavior is as follows:

```
Welcome to CityConnect!

Enter command: addroute Clementi BuonaVista 12
Route from Clementi to BuonaVista with distance 12km added

Enter command: getdistance Clementi BuonaVista
Distance from Clementi to BuonaVista is 12

Enter command: getdistance Clementi JurongWest
No route exists from Clementi to JurongWest!

Enter command: addroute Clementi JurongWest 24
Route from Clementi to JurongWest with distance 24km added

Enter command: getdistance Clementi JurongWest
Distance from Clementi to JurongWest is 24

Enter command: exit

```

What classes would you have in your code if you write your program based on the OOP paradigm?

One class you can have is `Route`

write your answer here...

W1.1f :★

C++ to Java → Classes → Defining Classes

Can define Java classes

As you know,

- Defining a class introduces a new object type.
- Every object belongs to some object type; that is, it is an instance of some class.
- A class definition is like a template for objects: it specifies what attributes the objects have and what methods can operate on them.

- The `new` operator instantiates objects, that is, it creates new instances of a class.
- The methods that operate on an object type are defined in the class for that object.

Here's a class called `Time`, intended to represent a moment in time. It has three attributes and no methods.

```
1 public class Time {
2     private int hour;
3     private int minute;
4     private int second;
5 }
```

You can give a class any name you like. **The Java convention is to use PascalCase format for class names**.

The code is usually placed in a file whose name matches the class e.g., the `Time` class should be in a file named `Time.java`.

When a class is `public` (e.g., the `Time` class in the above example) it can be used in other classes. But the instance variables that are `private` (e.g., the `hour`, `minute` and `second` attributes of the `Time` class) can only be accessed from inside the `Time` class.

Constructors

The syntax for constructors is similar to that of other methods, except:

- The name of the constructor is the same as the name of the class.
- The keyword `static` is omitted.
- Does not return anything. A constructor returns the created object by default.

When you invoke `new`, Java creates the object and calls your constructor to initialize the instance variables. When the constructor is done, it returns a reference to the new object.

Here is an example constructor for the `Time` class:

```
1 public Time() {
2     hour = 0;
3     minute = 0;
4     second = 0;
5 }
```

This constructor does not take any arguments. Each line initializes an instance variable to `0` (which in this example means midnight). Now you can create `Time` objects.

```
Time time = new Time();
```

Like other methods, constructors can be overloaded.

You can add another constructor to the `Time` class to allow creating `Time` objects that are initialized to a specific time:

```
1 public Time(int h, int m, int s) {
2     hour = h;
3     minute = m;
4     second = s;
5 }
```

Here's how you can invoke the new constructor: `Time justBeforeMidnight = new Time(11, 59, 59);`

`this` keyword

The `this` keyword is a reference variable in Java that refers to the current object. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods. But you do not declare `this`, and you can't make an assignment to it.

📦 In the following version of the constructor, the names and types of the parameters are the same as the instance variables (parameters don't have to use the same names, but that's a common style). As a result, the parameters ***shadow*** (or hide) the instance variables, so the keyword `this` is necessary to tell them apart.

```
public Time(int hour, int minute, int second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
```

`this` can be used to refer to a constructor of a class within the same class too.

📦 In this example the constructor `Time()` uses the `this` keyword to call its own overloaded constructor `Time(int, int, int)`

```
public Time() {
    this(0, 0, 0); // call the overloaded constructor
}

public Time(int hour, int minute, int second) {
    // ...
}

```

Instance methods

You can add methods to a class which can then be used from the objects of that class. These *instance* methods do not have the `static` keyword in the method signature. Instance methods can access attributes of the class.

📦 Here's how you can add a method to the `Time` class to get the number of seconds passed till midnight.

```
public int secondsSinceMidnight() {
    return hour*60*60 + minute*60 + second;
}
```

Here's how you can use that method.

```
Time t = new Time(0, 2, 5);
System.out.println(t.secondsSinceMidnight() + " seconds since midnight!");
```

[Key Exercise] define a `Circle` class

Define a `Circle` class so that the code given below produces the given output. The nature of the class is a follows:

- Attributes(all `private`):
  - **`int x`, `int y`** : represents the location of the circle
  - **`double radius`** : the radius of the circle
- Constructors:
  - **`Circle()`** : initializes `x`, `y`, `radius` to 0
  - **`Circle(int x, int y, double radius)`** : initializes the attributes to the given values
- Methods:
  - **`getArea()`** : `int`
    Returns the area of the circle as an `int` value (not `double`). Calculated as `Pi * (radius)`^{2}
    💡 You can convert a `double` to an `int` using `(int)` e.g., `x = (int)2.25` gives `x` the value `2`.
    💡 You can use `Math.PI` to get the value of Pi

You can use `Math.pow()` to raise a number to a specific power e.g., `Math.pow(3, 2)` calculates 3^2

```
public class Main {
    public static void main(String[] args) {
        Circle c = new Circle();

        System.out.println(c.getArea());
        c = new Circle(1, 2, 5);
        System.out.println(c.getArea());

    }
}
```

↓

```
0
78
```

- Put the `Circle` class in a file called `Circle.java`

Partial solution:

```
public class Circle {
    private int x;
    // ...

    public Circle(){
        this(0, 0, 0);
    }

    public Circle(int x, int y, double radius){
        this.x = x;
        // ...
    }

    public int getArea(){
        double area = Math.PI * Math.pow(radius, 2);
        return (int)area;
    }

}
```

W1.1g

C++ to Java → Classes → Getters and setters

Can define getters and setters

As the instance variables of `Time` are private, you can access them from within the `Time` class only. To compensate, **you can provide methods to access attributes**:

```java
public int getHour() {
    return hour;
}

public int getMinute() {
    return minute;
}

public int getSecond() {
    return second;
}
```

Methods like these are formally called “accessors”, but more commonly referred to as *getters*. By convention, the method that gets a variable named `something` is called `getSomething` .

Similarly, **you can provide *setter* methods to modify attributes** of a `Time` object:

```java
public void setHour(int hour) {
    this.hour = hour;
}

public void setMinute(int minute) {
    this.minute = minute;
}

public void setSecond(int second) {
    this.second = second;
}
```

[Key Exercise] add getters/setters to the `Circle` class

Consider the `Circle` class below:

```java
public class Circle {
    private int x;
    private int y;
    private double radius;

    public Circle(){
        this(0, 0, 0);
    }

    public Circle(int x, int y, double radius){
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public int getArea(){
        double area = Math.PI * Math.pow(radius, 2);
        return (int)area;
    }

}
```

Update it as follows so that code given below produces the given output.

- Add getter/setter methods for all three attributes
- Update the setters and constructors such that if the radius supplied is negative, the code automatically set the radius to 0 instead.

```java
public class Main {
    public static void main(String[] args) {
        Circle c = new Circle(1,2, 5);

        c.setX(4);
        c.setY(5);
        c.setRadius(6);
        System.out.println("x      : " + c.getX());
        System.out.println("y      : " + c.getY());
        System.out.println("radius : " + c.getRadius());
        System.out.println("area   : " + c.getArea());

        c.setRadius(-5);
        System.out.println("radius : " + c.getRadius());
        c = new Circle(1, 1, -4);
        System.out.println("radius : " + c.getRadius());

    }
}
```

↓

```
x      : 4
y      : 5
radius : 6.0
area   : 113
radius : 0.0
radius : 0.0
```

Partial solution:

```java
public Circle(int x, int y, double radius){
    setX(x);
    setY(y);
    setRadius(radius);
}
```

W1.1h ★★

Paradigms → OOP → Classes → Class-Level Members

Can explain class-level members

While all objects of a class has 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 message 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**.

Which of these are suitable as class-level variables?

- ☐ a. system: multi-player Pac Man game, Class: `Player` , variable: `totalScore`
- ☐ b. system: eLearning system, class: `Course` , variable: `totalStudents`
- ☐ c. system: ToDo manager, class: `Task` , variable: `totalPendingTasks`
- ☐ d. system: any, class: `ArrayList` , variable: `total` (i.e., total items in a given `ArrayList` object)

(c)

Explanation: `totalPendingTasks` should not be managed by individual `Task` objects and therefore suitable to be maintained as a class-level variable. The other variables should be managed at instance level as their value varies from instance to instance. e.g., `totalStudents` for one `Course` object will differ from `totalStudents` of another.

W1.1i : ★★

C++ to Java → Classes → Class-Level Members

🏆 **Can use class-level members**

The content below is an extract from -- Java Tutorial, with slight adaptations.

> When a number of objects are created from the same class blueprint, they each have their own distinct copies of instance variables. In the case of a `Bicycle` class, the instance variables are gear, and speed. Each Bicycle object has its own values for these variables, stored in different memory locations.
>
> **Sometimes, you want to have variables that are common to all objects. This is accomplished with the `static` modifier.** Fields that have the `static` modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

📦 Suppose you want to create a number of Bicycle objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many `Bicycle` objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, `numberOfBicycles` , as follows:

```java
public class Bicycle {

    private int gear;
    private int speed;

    // an instance variable for the object ID
    private int id;

    // a class variable for the number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;
        ...
}
```

Class variables **are referenced by the class name itself**, as in `Bicycle.numberOfBicycles` This makes it clear that they are class variables.

The Java programming language supports static methods as well as static variables. Static methods, which have the `static` modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in `ClassName.methodName(args)`

The `static` modifier, in combination with the `final` modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change. For example, the following variable declaration defines a constant named `PI`, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter): `static final double PI = 3.141592653589793;`

Here is an example with class-level variables and class-level methods:

```java
public class Bicycle {

    private int gear;
    private int speed;

    private int id;

    private static int numberOfBicycles = 0;


    public Bicycle(int startSpeed, int startGear) {
        gear = startGear;
        speed = startSpeed;

        numberOfBicycles++;
        id = numberOfBicycles;
    }

    public int getID() {
        return id;
    }

    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }

    public int getGear(){
        return gear;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    // ...

}
```

Explanation of `System.out.println(...)` :

- `out` is a class-level public attribute of the `System` class.
- `println` is a instance level method of the `out` object.

[Key Exercise] add getMaxRadius to the `Circle` class

Consider the `Circle` class below:

```
1  public class Circle {
2      private int x;
3      private int y;
4      private double radius;
5
6      public Circle(){
7          this(0, 0, 0);
8      }
9
10     public Circle(int x, int y, double radius){
11         setX(x);
12         setY(y);
13         setRadius(radius);
14     }
15
16     public int getX() {
17         return x;
18     }
19
20     public void setX(int x) {
21         this.x = x;
22     }
23
24     public int getY() {
25         return y;
26     }
27
28     public void setY(int y) {
29         this.y = y;
30     }
31
32     public double getRadius() {
33         return radius;
34     }
35
36     public void setRadius(double radius) {
37         this.radius = Math.max(radius, 0);
38     }
39
40     public int getArea(){
41         double area = Math.PI * Math.pow(radius, 2);
42         return (int)area;
43     }
44 }
```

Update it as follows so that code given below produces the given output.

- Add a class-level `getMaxRadius` method that returns the maximum radius that has been used in all `Circle` objects created thus far.

```java
public class Main {
    public static void main(String[] args) {
        Circle c = new Circle();
                    System.out.println("max radius used so far : " +
Circle.getMaxRadius());
        c = new Circle(0, 0, 10);
                    System.out.println("max radius used so far : " +
Circle.getMaxRadius());
        c = new Circle(0, 0, -15);
                    System.out.println("max radius used so far : " +
Circle.getMaxRadius());
        c.setRadius(12);
                    System.out.println("max radius used so far : " +
Circle.getMaxRadius());
    }
}
```

```
max radius used so far : 0.0
max radius used so far : 10.0
max radius used so far : 10.0
max radius used so far : 12.0
```

You can use a `static` variable `maxRadius` to track the maximum value used for the `radius` attribute so far.

Partial solution:

```java
public void setRadius(double radius) {
    this.radius = Math.max(radius, 0);
    if (maxRadius < this.radius){
        // ...
```

W1.1j

Paradigms → OOP → Classes → Enumerations

Can explain the meaning of 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 values such as `9` being assigned to it. But if you define an enumeration type called `Priority` that has three values `HIGH`, `MEDIUM`, `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`

W1.1k

C++ to Java → Miscellaneous Topics → Enumerations

Can use Java enumerations

You can define an enum type by using the `enum` keyword. Because they are constants, the names of an enum type's fields are in uppercase letters e.g., `FLAG_SUCCESS` by convention.

Defining an enumeration to represent days of a week (code to be put in the `Day.java` file):

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Some examples of using the `Day` enumeration defined above:

```
Day today = Day.MONDAY;
Day[] holidays = new Day[]{Day.SATURDAY, Day.SUNDAY};

switch (today) {
case SATURDAY:
case SUNDAY:
    System.out.println("It's the weekend");
    break;
default:
    System.out.println("It's a week day");
}
```

Note that while enumerations are usually a simple set of fixed values, **Java enumerations can have behaviors too**, as explained in this tutorial from -- Java Tutorial

[Key Exercise] show priority color

Define an enumeration named `Priority`. Add the missing `describe` method to the code below so that it produces the output given.

```
public class Main {

    // Add your method here

    public static void main(String[] args) {
        describe("Red", Priority.HIGH);
        describe("Orange", Priority.MEDIUM);
        describe("Blue", Priority.MEDIUM);
        describe("Green", Priority.LOW);
    }
}
```

↓

```
Red indicates high priority
Orange indicates medium priority
Blue indicates medium priority
Green indicates low priority
```

Use a `switch` statement to select between possible values for `Priority`.

```
    public static void describe(String color, Priority p) {
        switch (p) {
            case LOW:
                System.out.println(color + " indicates low priority");
                break;
            // ...
        }
    }
```

Code for the enumeration is given below:

```
public enum Priority {
    HIGH, MEDIUM, LOW
}
```

[W1.2] OOP Inheritance

W1.2a ★

Paradigms → OOP → Inheritance → What

Can explain the meaning of inheritance

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, *Super* class
- Other names for Derived class: *Child* class, *Sub* class, *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` behaves 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.

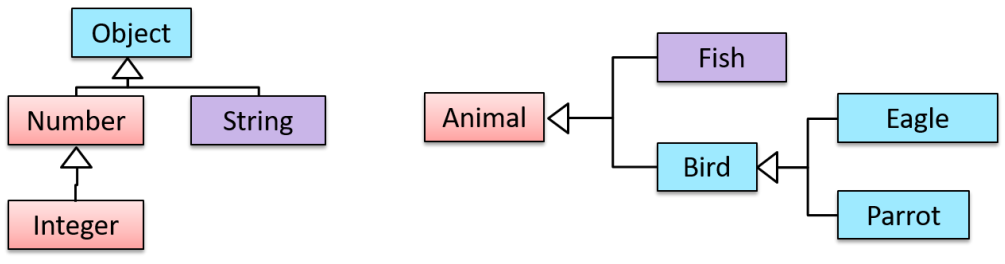
Inheritance does not necessarily mean a sub-type relationship exists. However, the two often go hand-in-hand. For simplicity, at this point let us assume inheritance implies a sub-type relationship.

To continue the previous example,

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

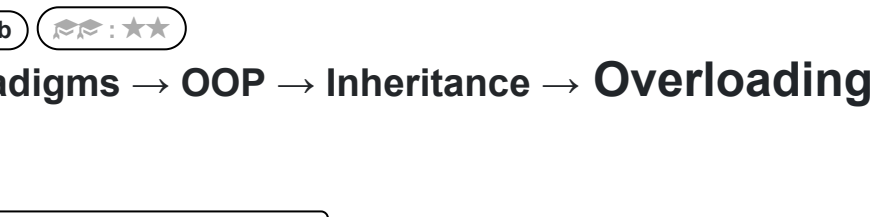
Inheritance relationships through a chain of classes can result in inheritance *hierarchies* (aka inheritance *trees*).

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 an `Vehicle`, an `Asset` and a `Liability`.



Which of these are correct?

- a. Superclass is more general than the subclass.
- b. Child class is more specialized than the parent class.
- c. A class can inherit behavior from its ancestor classes (ancestor classes = classes above it in the inheritance hierarchy).
- d. Code reuse can be one benefit of inheritance.
- e. A change to the superclass will not affect its subclasses.

(a) (b) (c) (d)

Explanation: (e) is incorrect. Because subclasses inherit behavior from the superclass, any changes to the superclass could affect subclasses.

W1.2b : ★★

Paradigms → OOP → Inheritance → Overloading

Can explain method overloading

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

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

Example:

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

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

- `calculate(String): void`

- `calculate(int): void`

W1.2c : ★★

Paradigms → OOP → Inheritance → Overriding

OOP → Inheritance → What

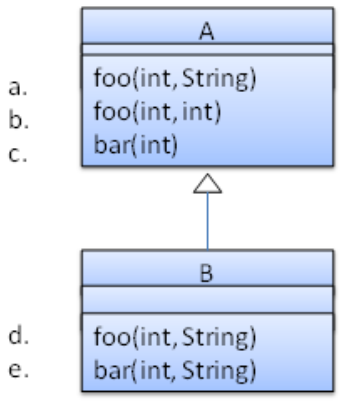
Can explain method overriding

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 *overloading* (rather than overriding).

Which of these methods override another method? `A` is the parent class. `B` inherits `A`.



a.
b.
c.
d.
e.

- [ ] a
- [ ] b
- [ ] c
- [ ] d
- [ ] e

d

Explanation: Method overriding requires a method in a *child class* to use the same method name and same parameter sequence used by one of its ancestors

W1.2d : ★★

C++ to Java → Inheritance → Inheritance (Basics)

Can use basic inheritance

Given below is an extract from the -- Java Tutorial, with slight adaptations.

A class that is derived from another class is called a *subclass* (also a *derived* class, *extended* class, or *child* class). The class from which the subclass is derived is called a *superclass* (also a *base* class or a *parent* class).

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Every class has one and only one direct superclass (*single inheritance*), except the `Object` class, which has no superclass, . In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`. Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be *descended* from all the classes in the *inheritance chain* stretching back to `Object`. Java does not support *multiple inheritance* among classes.

The `java.lang.Object` class defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a single hierarchy of classes.

The keyword `extends` indicates one class inheriting from another.

Here is the sample code for a possible implementation of a `Bicycle` class and a `MountainBike` class that is a subclass of the `Bicycle`:

```
1  public class Bicycle {
2
3      public int gear;
4      public int speed;
5
6      public Bicycle(int startSpeed, int startGear) {
7          gear = startGear;
8          speed = startSpeed;
9      }
10
11     public void setGear(int newValue) {
12         gear = newValue;
13     }
14
15     public void applyBrake(int decrement) {
16         speed -= decrement;
17     }
18
19     public void speedUp(int increment) {
20         speed += increment;
21     }
22
23 }
```

```
1  public class MountainBike extends Bicycle {
2
3      // the MountainBike subclass adds one field
4      public int seatHeight;
5
6      // the MountainBike subclass has one constructor
7      public MountainBike(int startHeight, int startSpeed, int startGear)
   {
8          super(startSpeed, startGear);
9          seatHeight = startHeight;
10     }
11
12     // the MountainBike subclass adds one method
13     public void setHeight(int newValue) {
14         seatHeight = newValue;
15     }
16 }
```

A subclass inherits all the fields and methods of the superclass. In the example above, `MountainBike` inherits all the fields and methods of `Bicycle` and adds the field `seatHeight` and a method to set it.

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super` . You can also use `super` to refer to a hidden field (although hiding fields is discouraged).

Consider this class, `Superclass` and a subclass, called `Subclass` , that overrides `printMethod()` :

```
1  public class Superclass {
2
3      public void printMethod() {
4          System.out.println("Printed in Superclass.");
5      }
6  }
```

```
1  public class Subclass extends Superclass {
2
3      // overrides printMethod in Superclass
4      public void printMethod() {
5          super.printMethod();
6          System.out.println("Printed in Subclass");
7      }
8      public static void main(String[] args) {
9          Subclass s = new Subclass();
10         s.printMethod();
11     }
12 }
```

↓

```
1  Printed in Superclass.
2  Printed in Subclass
```

Within `Subclass` , the simple name `printMethod()` refers to the one declared in `Subclass` , which overrides the one in `Superclass` . So, to refer to `printMethod()` inherited from `Superclass` , `Subclass` must use a qualified name, using `super` as shown.

Subclass Constructors

A subclass constructor can invoke the superclass constructor. Invocation of a superclass constructor must be the first line in the subclass constructor. The syntax for calling a superclass constructor is `super()` (which invokes the no-argument constructor of the superclass) or `super(parameters)` (to invoke the superclass constructor with a matching parameter list).

The following example illustrates how to use the `super` keyword to invoke a superclass's constructor. Recall from the `Bicycle` example that `MountainBike` is a subclass of `Bicycle` . Here is the `MountainBike` (subclass) constructor that calls the superclass constructor and then adds some initialization code of its own (i.e., `seatHeight = startHeight;`):

```
1  public MountainBike(int startHeight, int startSpeed, int startGear) {
2      super(startSpeed, startGear);
3      seatHeight = startHeight;
4  }
```

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. `Object` does have such a constructor, so if `Object` is the only superclass, there is no problem.

Access Modifiers (simplified)

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. Given below is a simplified version of Java access modifiers, assuming you have not yet started placing your classes in different packages i.e., all classes are placed in the root level. A full explanation of access modifiers is given in a later topic.

There are two levels of access control:

1. **At the class level**:
    - `public` : the class is visible to all other classes
    - **no modifier**: same as `public`
2. **At the member level**:
    - `public` : the member is visible to all other classes
    - **no modifier**: same as `public`
    - `protected` : the member is visible to sub classes only
    - `private` : the member is not visible to other classes (but can be accessed in its own class)

[Key Exercise] inherit the `Task` class

Background: Suppose we are creating a software to manage various tasks a person has to do. Two types of such tasks are,

- *Todos*: i.e., things that needs to be done some day e.g., 'Read the book Lord of the Rings'
- *Deadlines*: i.e., things to be done by a specific date/time e.g., 'Read the text book by Nov 25th'

The `Task` class is given below:

```
1  public class Task {
2      protected String description;
3
4      public Task(String description) {
5          this.description = description;
6      }
7
8      public String getDescription() {
9          return description;
10     }
11 }
```

1. Write a `Todo` class that inherits from the `Task` class.
    - It should have an additional `boolean` field `isDone` to indicate whether the todo is done or not done.
    - It should have a `isDone()` method to access the `isDone` field and a `setDone(boolean)` method to set the `isDone` field.
2. Write a `Deadline` class that inherits from the `Todo` class that you implemented in the previous step. It should have,
    - an additional `String` field `by` to store the details of when the task to be done e.g., `Jan 25th 5pm`
    - a `getBy()` method to access the value of the `by` field, and a corresponding `setBy(String)` method.
    - a constructor of the form `Deadline(String description, String by)`

The expected behavior of the two classes is as follows:

```java
public class Main {
    public static void main(String[] args) {
        // create a todo task and print details
        Todo t = new Todo("Read a good book");
        System.out.println(t.getDescription());
        System.out.println(t.isDone());

        // change todo fields and print again
        t.setDone(true);
        System.out.println(t.isDone());

        // create a deadline task and print details
        Deadline d = new Deadline("Read textbook", "Nov 16");
        System.out.println(d.getDescription());
        System.out.println(d.isDone());
        System.out.println(d.getBy());

        // change deadline details and print again
        d.setDone(true);
        d.setBy("Postponed to Nov 18th");
        System.out.println(d.isDone());
        System.out.println(d.getBy());
    }
}
```

```
Read a good book
false
true
Read textbook
false
Nov 16
true
Postponed to Nov 18th
```

`Todo` class is given below. You can follow a similar approach for the `Deadline` class.

```java
public class Todo extends Task {
    protected boolean isDone;

    public Todo(String description) {
        super(description);
        isDone = false;
    }
}
```

[W1.3] OOP: Polymorphism

Polymorphism

W1.3a ★★★

Paradigms → OOP → Polymorphism → What

Can explain OOP polymorphism

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' 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".

W1.3b ★★★

C++ to Java → Inheritance → Polymorphism

Can use polymorphism in Java

Java is a *strongly-typed* language which means the code works with only the object types that it targets.

The following code `PetShelter` keeps a list of `Cat` objects and make them `speak` . The code will not work with any other type, for example, `Dog` objects.

```
1  public class PetShelter {
2      private static Cat[] cats = new Cat[]{
3              new Cat("Mittens"),
4              new Cat("Snowball")};
5
6      public static void main(String[] args) {
7          for (Cat c: cats){
8              System.out.println(c.speak());
9          }
10     }
11 }
```

```
Mittens: Meow
Snowball: Meow
```

▶ The `Cat` class

This **strong-typing can lead to unnecessary verbosity** caused by repetitive similar code that do similar things with different object types.

📦 If the `PetShelter` is to keep both cats and dogs, you'll need two arrays and two loops:

```
public class PetShelter {
    private static Cat[] cats = new Cat[]{
            new Cat("Mittens"),
            new Cat("Snowball")};
    private static Dog[] dogs = new Dog[]{
            new Dog("Spot")};

    public static void main(String[] args) {
        for (Cat c: cats){
            System.out.println(c.speak());
        }
        for(Dog d: dogs){
            System.out.println(d.speak());
        }
    }
}
```

```
Mittens: Meow
Snowball: Meow
Spot: Woof
```

▶ The `Dog` class

A better way is to **take advantage of polymorphism to write code that targets a superclass so that it works with any subclass objects.**

📦 The `PetShelter2` uses one data structure to keep both types of animals and one loop to make them speak. The code targets the `Animal` superclass (assuming `Cat` and `Dog` inherits from the `Animal` class) instead of repeating the code for each animal type.

```
public class PetShelter2 {
    private static Animal[] animals = new Animal[]{
            new Cat("Mittens"),
            new Cat("Snowball"),
            new Dog("Spot")};

    public static void main(String[] args) {
        for (Animal a: animals){
            System.out.println(a.speak());
        }
    }
}
```

```
Mittens: Meow
Snowball: Meow
Spot: Woof
```

The `Animal`, `Cat`, and `Dog` classes

Explanation: Because Java supports polymorphism, you can store both `Cat` and `Dog` objects in an array of `Animal` objects. Similarly, you can call the `speak` method on any `Animal` object (as done in the loop) and yet get different behavior from `Cat` objects and `Dog` objects.

Suggestion: try to add an `Animal` object (e.g., `new Animal("Unnamed")`) to the `animals` array and see what happens.

Polymorphic code is better in several ways:

- It is **shorter**.
- It is **simpler**.
- It is more **flexible** (in the above example, the `main` method will work even if we add more animal types).

[Key Exercise] print shape area

The `Main` class below keeps a list of `Circle` and `Rectangle` objects and prints the area (as an `int` value) of each shape when requested.

Add the missing variables/methods to the code below so that it produces the output given.

```
public class Main {
    //TODO add your methods here

    public static void main(String[] args) {
        addShape(new Circle(5));
        addShape(new Rectangle(3, 4));
        addShape(new Circle(10));
        printAreas();
        addShape(new Rectangle(4, 4));
        printAreas();
    }
}
```

↓

```
78
12
314
78
12
314
16
```

`Circle` class and `Rectangle` class is given below but you'll need to add a parent class `Shape`:

```
public class Circle {

    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public int area() {
        return (int)(Math.PI * radius * radius);
    }
}
```

```java
public class Rectangle {
    private int height;
    private int width;

    public Rectangle(int height, int width){
        this.height = height;
        this.width = width;
    }

    public int area() {
        return height * width;
    }
}
```

Abstract Classes

W1.3c : ★★★

Paradigms → OOP → Inheritance → Abstract Classes and Methods

Can implement 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 exist 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.

W1.3d : ★★★

C++ to Java → Inheritance → Abstract Classes and Methods

Can use abstract classes and methods

In Java, **an *abstract method* is declared with the keyword `abstract` and given without an implementation**. If a class includes abstract methods, then the class itself must be declared abstract.

The `speak` method in this `Animal` class is `abstract`. Note how the method signature ends

with a semicolon and there is no method body. This makes sense as the implementation of the `speak` method depends on the type of the animal and it is meaningless to provide a common implementation for all animal types.

```
public abstract class Animal {

    protected String name;

    public Animal(String name){
        this.name = name;
    }
    public abstract String speak();
}
```

As one method of the class is `abstract`, the class itself is `abstract`.

An *abstract class* is declared with the keyword `abstract`. Abstract classes can be used as reference type but cannot be instantiated.

This `Account` class has been declared as abstract although it does not have any abstract methods. Attempting to instantiate `Account` objects will result in a compile error.

```
public abstract class Account {

    int number;

    void close(){
        //...
    }
}
```

`Account a;` → ✔ OK to use as a type
`a = new Account();` → ✖ Compile error!

In Java, even **a class that does not have any abstract methods *can* be declared as an abstract class.**

When an abstract class is subclassed, the subclass should provides implementations for all of the abstract methods in its superclass or else the subclass must also be declared abstract.

The `Feline` class below inherits from the abstract class `Animal` but it does not provide an implementation for the abstract method `speak`. As a result, the `Feline` class needs to be abstract too.

```
public abstract class Feline extends Animal {
    public Feline(String name) {
        super(name);
    }

}
```

The `DomesticCat` class inherits the abstract `Feline` class and provides the implementation for the abstract method `speak`. As a result, it need not be (but *can* be) declared as abstract.

```
public class DomesticCat extends Feline {
    public DomesticCat(String name) {
        super(name);
    }

    @Override
    public String speak() {
        return "Meow";
    }
}
```

- `Animal a = new Feline("Mittens");`
  ✖ Compile error! `Feline` is abstract.

- `Animal a = new DomesticCat("Mittens");`
  ✔ OK. `DomesticCat` can be instantiated and assigned to a variable of `Animal` type (the assignment is allowed by polymorphism).

- Oracle's Java Tutorials: Abstract Methods and Classes

[Key Exercise] print area with abstract `Shape`

The `Main` class below keeps a list of `Circle` and `Rectangle` objects and prints the area (as an `int` value) of each shape when requested.

```
public class Main {
    private static Shape[] shapes = new Shape[100];
    private static int shapeCount = 0;

    public static void addShape(Shape s){
        shapes[shapeCount] = s;
        shapeCount++;
    }

    public static void printAreas(){
        for (int i = 0; i < shapeCount; i++){
            shapes[i].print();
        }
    }

    public static void main(String[] args) {
        addShape(new Circle(5));
        addShape(new Rectangle(3, 4));
        addShape(new Circle(10));
        addShape(new Rectangle(4, 4));
        printAreas();
    }
}
```

↓

```
Circle of area 78
Rectangle of area 12
Circle of area 314
Rectangle of area 16
```

`Circle` class and `Rectangle` class is given below:

```
public class Circle extends Shape {

    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public int area() {
        return (int)(Math.PI * radius * radius);
    }

    @Override
    public void print() {
        System.out.println("Circle of area " + area());
    }
}
```

```java
public class Rectangle extends Shape {
    private int height;
    private int width;

    public Rectangle(int height, int width){
        this.height = height;
        this.width = width;
    }

    @Override
    public int area() {
        return height * width;
    }

    @Override
    public void print() {
        System.out.println("Rectangle of area " + area());
    }
}
```

Add the missing `Shape` class as an abstract class with two abstract methods .

▶ Partial solution

Choose the correct statements about Java abstract classes and concrete classes.

- a. A concrete class can contain an abstract method.
- b. An abstract class can contain concrete methods.
- c. An abstract class need not contain any concrete methods.
- d. An abstract class cannot be instantiated.

(b)(c)(d)

Explanation: A concrete class cannot contain even a single abstract method.

Interfaces

W1.3e

Paradigms → OOP → Inheritance → Interfaces

Can explain 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()`.

W1.3f :★★★

C++ to Java → Inheritance → Interfaces

Can use interfaces in Java

The text given in this section borrows some explanations and code examples from the -- Java Tutorial.

In Java, **an *interface* is a reference type, similar to a class, mainly containing method signatures.** Defining an interface is similar to creating a new class except it uses the keyword `interface` in place of `class`.

Here is an interface named `DrivableVehicle` that defines methods needed to drive a vehicle.

```
1  public interface DrivableVehicle {
2      void turn(Direction direction);
3      void changeLanes(Direction direction);
4      void signalTurn(Direction direction, boolean signalOn);
5      // more method signatures
6  }
```

Note that the method signatures have no braces (`{ }`) and are terminated with a semicolon.

Interfaces cannot be instantiated—they can only be *implemented* by classes. When an instantiable class implements an interface, indicated by the keyword `implements`, it provides a method body for each of the methods declared in the interface.

Here is how a class `CarModelX` can implement the `DrivableVehicle` interface.

```
1  public class CarModelX implements DrivableVehicle {
2
3      @Override
4      public void turn(Direction direction) {
5         // implementation
6      }
7
8      // implementation of other methods
9  }
```

An interface can be used as a type e.g., `DrivableVechile dv = new CarModelX();`.

Interfaces can inherit from other interfaces using the `extends` keyword, similar to a class inheriting another.

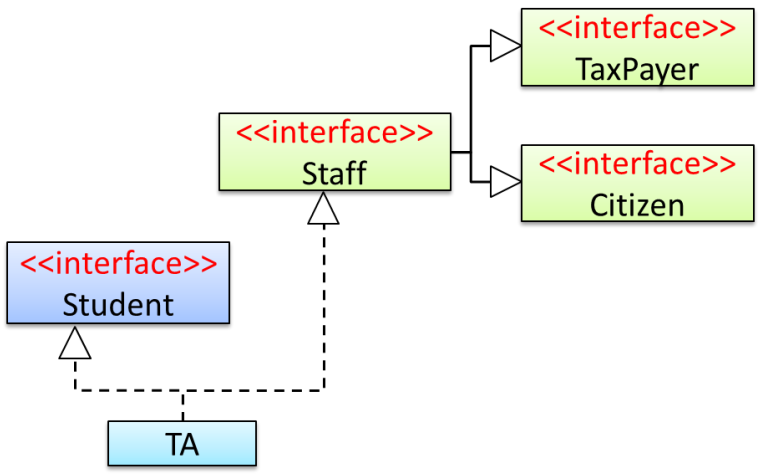
Here is an interface named `SelfDrivableVehicle` that inherits the `DrivableVehicle` interface.

```
public interface SelfDrivableVehicle extends DrivableVehicle {
   void goToAutoPilotMode();
}
```

Note that the method signatures have no braces and are terminated with a semicolon.

Furthermore, **Java allows multiple inheritance among interfaces.** A Java interface can inherit multiple other interfaces. **A Java class can *implement* multiple interfaces** (and inherit from one class).

The design below is allowed by Java. In case you are not familiar with UML notation used: solid lines indicate normal inheritance; dashed lines indicate interface inheritance; the triangle points to the parent.



1. `Staff` interface inherits (note the solid lines) the interfaces `TaxPayer` and `Citizen`.
2. `TA` class implements both `Student` interface and the `Staff` interface.
3. Because of point 1 above, `TA` class has to implement all methods in the interfaces `TaxPayer` and `Citizen`.
4. Because of points 1,2,3, a `TA` *is a* `Staff`, *is a* `TaxPayer` and *is a* `Citizen`.

Interfaces can also contain constants and static methods.

This example adds a constant `MAX_SPEED` and a static method `isSpeedAllowed` to the interface `DrivableVehicle`.

```
public interface DrivableVehicle {

    int MAX_SPEED = 150;

    static boolean isSpeedAllowed(int speed){
        return speed <= MAX_SPEED;
    }

    void turn(Direction direction);
    void changeLanes(Direction direction);
    void signalTurn(Direction direction, boolean signalOn);
    // more method signatures
}
```

Interfaces can contain *default* method implementations and *nested types*. They are not covered here.

[Key Exercise] print `Printable` items

The `Main` class below passes a list of `Printable` objects (i.e., objects that implement the `Printable` interface) for another method to be printed.

```java
public class Main {

    public static void printObjects(Printable[] items) {
        for (Printable p : items) {
            p.print();
        }
    }

    public static void main(String[] args) {
        Printable[] printableItems = new Printable[]{
                new Circle(5),
                new Rectangle(3, 4),
                new Person("James Cook")};

        printObjects(printableItems);
    }
}
```

```
Circle of area 78
Rectangle of area 12
Person of name James Cook
```

Classes `Shape`, `Circle`, and `Rectangle` are given below:

```java
public abstract class Shape {

    public abstract int area();
}
```

```java
public class Circle extends Shape implements Printable {

    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public int area() {
        return (int)(Math.PI * radius * radius);
    }

    @Override
    public void print() {
        System.out.println("Circle of area " + area());
    }
}
```

```java
public class Rectangle extends Shape implements Printable {
    private int height;
    private int width;

    public Rectangle(int height, int width){
        this.height = height;
        this.width = width;
    }
```

How Polymorphism Works

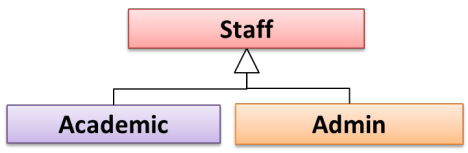
W1.3g : ★★★

Paradigms → OOP → Inheritance → Substitutability

Paradigms → Object Oriented Programming → Inheritance → What

Can explain 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 `Academic` is an instance of a `Staff`, but a `Staff` is not necessarily an instance of an `Academic`. 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.

```java
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`.

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

W1.3h : ★★★

Paradigms → OOP → Inheritance → Dynamic and Static Binding

Can explain dynamic and static binding

Dynamic Binding (aka late binding) : a mechanism where method calls in code are resolved at runtime, rather than at compile time.

Overridden methods are resolved using dynamic binding, and therefore resolves to the implementation in the actual type of the object.

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

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

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

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

In contrast, overloaded methods are resolved using static binding.

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

```
class Account {

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

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

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

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

W1.3i : ★★★

Paradigms → OOP → Polymorphism → How

Paradigms → Object Oriented Programming → Inheritance → Substitutability

Paradigms → Object Oriented Programming → Inheritance → Dynamic and Static Binding

Can explain how substitutability operation overriding, and dynamic binding relates to polymorphism

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

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

- **Dynamic binding**: Calls to overridden methods are bound to the implementation of the actual object's class dynamically during the runtime. That is how the polymorphic code can call the method of the parent class and yet execute the implementation of the child class.

Which one of these is least related to how OO programs achieve polymorphism?

- a. substitutability
- b. dynamic binding
- c. operation overloading

(c)

Explanation: Operation overriding is the one that is related, not operation overloading.

[W1.4] Java: Collections

W1.4a : ★★★

C++ to Java → Collections → The Collections Framework

Can explain the Collections framework

This section uses extracts from the -- Java Tutorial, with some adaptations.

A *collection* — sometimes called a *container* — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

> Typically, collections represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

The collections framework is a unified architecture for representing and manipulating collections. It contains the following:

- **Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.
  Example: the `List<E>` interface can be used to manipulate list-like collections which may be implemented in different ways such as `ArrayList<E>` or `LinkedList<E>`.

- **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  Example: the `ArrayList<E>` class implements the `List<E>` interface while the `HashMap<K, V>` class implements the `Map<K, V>` interface.

- **Algorithms**: These are the methods that perform useful computations, such as *searching* and *sorting*, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.
  Example: the `sort(List<E>)` method can sort a collection that implements the `List<E>` interface.

= **A well-known example of collections frameworks is the C++ *Standard Template Library (STL)*.** ≠ Although both are collections frameworks and the syntax look similar, note that there are important philosophical and implementation differences between the two.

The following list describes the core collection interfaces:

- **`Collection`** — the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as `Set` and `List`. Also see the `Collection` API.

- **`Set`** — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. Also see the `Set` API.

- **`List`** — an ordered collection (sometimes called a *sequence*). `List`s can contain duplicate elements. The user of a `List` generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). Also see the `List` API.

- **`Queue`** — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Queue` provides additional insertion, extraction, and inspection operations. Also see the `Queue` API.

- **`Map`** — an object that maps keys to values. A `Map` cannot contain duplicate keys; each key can map to at most one value. Also see the `Map` API.

- Others: `Deque`, `SortedSet`, `SortedMap`

W1.4b : ★★

C++ to Java → Collections → The `ArrayList` Class

Can use the ArrayList class

The `ArrayList` class is a resizable-array implementation of the `List` interface. Unlike a normal `array`, an `ArrayList` can grow in size as you add more items to it. The example below illustrate some of the useful methods of the `ArrayList` class using an `ArrayList` of `String` objects.

```java
import java.util.ArrayList;

public class ArrayListDemo {

    public static void main(String args[]) {
        ArrayList<String> items = new ArrayList<>();

        System.out.println("Before adding any items:" + items);

        items.add("Apple");
        items.add("Box");
        items.add("Cup");
        items.add("Dart");
        print("After adding four items: " + items);

        items.remove("Box"); // remove item "Box"
        print("After removing Box: " + items);

        items.add(1, "Banana"); // add "Banana" at index 1
        print("After adding Banana: " + items);

        items.add("Egg"); // add "Egg", will be added to the end
        items.add("Cup"); // add another "Cup"
        print("After adding Egg: " + items);

        print("Number of items: " + items.size());

        print("Index of Cup: " + items.indexOf("Cup"));
        print("Index of Zebra: " + items.indexOf("Zebra"));

        print("Item at index 3 is: " + items.get(2));

        print("Do we have a Box?: " + items.contains("Box"));
        print("Do we have an Apple?: " + items.contains("Apple"));

        items.clear();
        print("After clearing: " + items);
    }

    private static void print(String text) {
        System.out.println(text);
    }
}
```

↓

```
Before adding any items:[]
After adding four items: [Apple, Box, Cup, Dart]
After removing Box: [Apple, Cup, Dart]
After adding Banana: [Apple, Banana, Cup, Dart]
After adding Egg: [Apple, Banana, Cup, Dart, Egg, Cup]
Number of items: 6
Index of Cup: 2
Index of Zebra: -1
Item at index 3 is: Cup
Do we have a Box?: false
Do we have an Apple?: true
After clearing: []
```

[Try the above code on Repl.it]

- `ArrayList` API
- A tutorial on ArrayLists from beginnersbook.com

[Key Exercise] Numbers list

Add the missing methods to the class given below so that it produces the output given.

Use an `ArrayList` to store the numbers.

```java
public class Main {

    //TODO: add your methods here

    public static void main(String[] args) {
        System.out.println("Adding numbers to the list");
        addNumber(3);
        addNumber(8);
        addNumber(24);
        System.out.println("The total is: " + getTotal());
        System.out.println("8 in the list : " + isFound(8) );
        System.out.println("5 in the list : " + isFound(5) );
        removeNumber(8);
        System.out.println("The total is: " + getTotal());
    }

}
```

```
Adding numbers to the list
[3]
[3, 8]
[3, 8, 24]
The total is: 35
8 in the list : true
5 in the list : false
[3, 24]
The total is: 27
```

Partial solution:

```java
import java.util.ArrayList;

public class Main {
    private static ArrayList<Integer> numbers = new ArrayList<>();

    private static void addNumber(int i) {
        numbers.add(Integer.valueOf(i));
        System.out.println(numbers);
    }

    // ...

}
```


W1.4c : ★★★

C++ to Java → Collections → The `HashMap` Class

Can use the HashMap class

`HashMap` is an implementation of the `Map` interface. It allows you to store a collection of *key-value pairs*. The example below illustrates how to use a `HashMap<String, Point>` to maintain a list of coordinates and their identifiers e.g., the identifier `x1` is used to identify the point `0,0` where `x1` is the *key* and `0,0` is the *value*.

```
import java.awt.Point;
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Point> points = new HashMap<>();

        // put the key-value pairs in the HashMap
        points.put("x1", new Point(0, 0));
        points.put("x2", new Point(0, 5));
        points.put("x3", new Point(5, 5));
        points.put("x4", new Point(5, 0));

        // retrieve a value for a key using the get method
        print("Coordinates of x1: " + pointAsString(points.get("x1")));

        // check if a key or a value exists
        print("Key x1 exists? " + points.containsKey("x1"));
        print("Key x1 exists? " + points.containsKey("y1"));
        print("Value (0,0) exists? " + points.containsValue(new Point(0, 0)));
        print("Value (1,2) exists? " + points.containsValue(new Point(1, 2)));

        // update the value of a key to a new value
        points.put("x1", new Point(-1,-1));

        // iterate over the entries
        for (Map.Entry<String, Point> entry : points.entrySet()) {
            print(entry.getKey() + " = " + pointAsString(entry.getValue()));
        }

        print("Number of keys: " + points.size());
        points.clear();
        print("Number of keys after clearing: " + points.size());

    }

    public static String pointAsString(Point p) {
        return "[" + p.x + "," + p.y + "]";
    }

    public static void print(String s) {
        System.out.println(s);
    }
}
```

↓

```
Coordinates of x1: [0,0]
Key x1 exists? true
Key x1 exists? false
Value (0,0) exists? true
Value (1,2) exists? false
x1 = [-1,-1]
x2 = [0,5]
x3 = [5,5]
x4 = [5,0]
Number of keys: 4
Number of keys after clearing: 0
```

[Try the above code on Repl.it]

[Key Exercise] weekly roster

The class given below keeps track of how many people signup to attend an event on each day of the week. Add the missing methods so that it produces the output given.

Use an `HashMap` to store the number of entries for each day.

```
public class Main {
    private static HashMap<String, Integer> roster = new HashMap<>();

    //TODO: add your methods here

    public static void main(String[] args) {
        addToRoster("Monday"); // i.e., one person signed up for Monday
        addToRoster("Wednesday"); // i.e., one person signed up for Wednesday
            addToRoster("Wednesday"); // i.e., another person signed up for Wednesday
        addToRoster("Friday");
        addToRoster("Monday");
        printRoster();
    }

}
```

↓

```
Monday => 2
Friday => 1
Wednesday => 2
```

Partial solution:

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    private static HashMap<String, Integer> roster = new HashMap<>();

    private static void addToRoster(String day) {
        if (roster.containsKey(day)){
            Integer newValue = Integer.valueOf(roster.get(day).intValue() + 1);
            roster.put(day, newValue);
        } else {
            roster.put(day, Integer.valueOf(1));
        }
    }

    // ...
}
```

- `HashMap` API
- A tutorial on HashMaps from journaldev.com

[W1.5] **Exception Handling**

W1.5a : ★★

Implementation → Error Handling → Introduction → What

Can explain error handling

> 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

---

W1.5b : ★

Implementation → Error Handling → Exceptions → What

Can explain exceptions

Exceptions are used to deal with *'unusual' but not entirely unexpected* situations that the program might encounter at run time.

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.

W1.5c : ★★

C++ to Java → Exceptions → What are Exceptions?

Can explain Java Exceptions

Given below is an extract from the -- Java Tutorial, with some adaptations.

There are three basic categories of exceptions In Java:

- ***Checked* exceptions : exceptional conditions that a well-written application should anticipate and recover from.** All exceptions are checked exceptions, except for `Error`, `RuntimeException`, and their subclasses.

  > Suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the `FileReader` object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws `java.io.FileNotFoundException`. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

- ***Errors*: exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.** Errors are those exceptions indicated by `Error` and its subclasses.

  > 📦 Suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw `java.io.IOError`. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

- ***Runtime exceptions*: conditions that are internal to the application, and that the application usually cannot anticipate or recover from.** Runtime exceptions are those indicated by `RuntimeException` and its subclasses. These usually indicate programming bugs, such as logic errors or improper use of an API.

  > 📦 Consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw `NullPointerException`. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Errors and runtime exceptions are collectively known as *unchecked* exceptions.

W1.5d 🎓 : ★

Implementation → Error Handling → Exceptions → How

🏆 Can explain how exception handling is done typically

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.

Which are benefits of exceptions?

- [ ] a. Exceptions allow us to separate normal code from error handling code.
- [ ] b. Exceptions can prevent problems that happen in the environment.

- ☐ c. Exceptions allow us to handle in one location an error raised in another location.

(a) (c)

Explanation: Exceptions cannot *prevent* problems in the environment. They can only be used to handle and recover from such problems.

W1.5e 🎓: ★★★

C++ to Java → Exceptions → How to Use Exceptions

🏆 Can use Java Exceptions

The content below uses extracts from the -- Java Tutorial, with some adaptations.

A program can catch exceptions by using a combination of the `try`, `catch` blocks.

- The `try` block identifies a block of code in which an exception can occur.
- The `catch` block identifies a block of code, known as an exception handler, that can handle a particular type of exception.

📦 The `writeList()` method below calls a method `process()` that can cause two type of exceptions. It uses a try-catch construct to deal with each exception.

```
1  public void writeList() {
2      print("starting method");
3      try {
4          print("starting process");
5          process();
6          print("finishing process");
7
8      } catch (IndexOutOfBoundsException e) {
9          print("caught IOOBE");
10
11     } catch (IOException e) {
12         print("caught IOE");
13
14     }
15     print("finishing method");
16 }
```

Some possible outputs:

No exceptions ↓	`IOException` ↓	`IndexOutOfBoundsException` ↓
starting method starting process finishing process finishing method	starting method starting process ~~finishing process~~ caught IOE finishing method	starting method starting process ~~finishing process~~ caught IOOBE finishing method

You can **use a `finally` block to specify code that is guaranteed to execute with or without the exception.** This is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the `try` block.

📦 The `writeList()` method below has a `finally` block:

```java
public void writeList() {
    print("starting method");
    try {
        print("starting process");
        process();
        print("finishing process");

    } catch (IndexOutOfBoundsException e) {
        print("caught IOOBE");

    } catch (IOException e) {
        print("caught IOE");

    } finally {
        // clean up
        print("cleaning up");
    }
    print("finishing method");
}
```

Some possible outputs:

No exceptions ↓	`IOException` ↓	`IndexOutOfBoundsException` ↓
starting method starting process finishing process cleaning up finishing method	starting method starting process ~~finishing process~~ caught IOE cleaning up finishing method	starting method starting process ~~finishing process~~ caught IOOBE cleaning up finishing method

- The `try` statement should contain at least one `catch` block or a finally block and may have multiple `catch` blocks.
- The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message.

You can use the `throw` statement to throw an exception. The throw statement requires a throwable object as the argument.

Here's an example of a `throw` statement.

```java
if (size == 0) {
    throw new EmptyStackException();
}
```

In Java, **Checked exceptions are subject to the *Catch or Specify Requirement***: code that might throw checked exceptions must be enclosed by either of the following:

- A `try` statement that catches the exception. The `try` must provide a handler for the exception.
- A method that specifies that it can throw the exception. The method must provide a `throws` clause that lists the exception.

Unchecked exceptions are not required to follow to the *Catch or Specify Requirement* but you can apply the requirement to them too.

Here's an example of a method specifying that it throws certain checked exceptions:

```java
public void writeList() throws IOException, IndexOutOfBoundsException {
    print("starting method");
    process();
    print("finishing method");
}
```

```
Some possible outputs:
```

No exceptions ↓	`IOException` ↓	`IndexOutOfBoundsException` ↓
`starting method` `finishing method`	`starting method` ~~`finishing method`~~	`starting method` ~~`finishing method`~~

Java comes with a collection of built-in exception classes that you can use. When they are not enough, it is possible to create your own exception classes.

[Key Exercise] parse rectangle descriptor

The `Main` class below parses a string descriptor of a rectangle of the format `"WIDTHxHEIGHT"` e.g., `"3x4"` and prints the area of the rectangle.

```
1  public class Main {
2
3      public static void printArea(String descriptor){
4          //TODO: modify the code below
5          System.out.println(descriptor + "=" + calculateArea(descriptor));
6      }
7
8      private static int calculateArea(String descriptor) {
9          //TODO: modify the code below
10         String[] dimensions = descriptor.split("x");
11                                 return    Integer.parseInt(dimensions[0])    *
   Integer.parseInt(dimensions[1]);
12     }
13
14     public static void main(String[] args) {
15         printArea("3x4");
16         printArea("5x5");
17     }
18 }
```

↓

```
1  3x4=12
2  5x5=25
```

1. Update the code of `printArea` to print an error message if `WIDTH` and/or `HEIGHT` are not numbers e.g., `"Ax4"`
   (hint) `calculateArea` will throw the unchecked exception `NumberFormatException` if the code tries to parse a non-number to an integer.

2. Update the code of `printArea` to print an error message if the descriptor is missing `WIDTH` and/or `HEIGHT` e.g., `"x4"`
   (hint) `calculateArea` will throw the unchecked exception `IndexOutOfBoundsException` if one or both dimensions are missing.

3. Update the code of `calculateArea` to throw the checked exception `IllegalShapeException` if there are more than 2 dimensions e.g., `"5x4x3"` and update the `printArea` to print an error message for those cases. Here is the code for the `IllegalShapeException.java`

   ```
   1  public class IllegalShapeException extends Exception {
   2    //no other code needed
   3  }
   ```

Here is the expected behavior after you have done the above changes:

```
public class Main {

    //...

    public static void main(String[] args) {
        printArea("3x4");
        printArea("3xy");
        printArea("3x");
        printArea("3");
        printArea("3x4x5");
    }
}
```

```
3x4=12
WIDTH or HEIGHT is not a number: 3xy
WIDTH or HEIGHT is missing: 3x
WIDTH or HEIGHT is missing: 3
Too many dimensions: 3x4x5
```

Partial solution

W1.5f : ★★

Implementation → Error Handling → Exceptions → When

Can avoid using exceptions to control normal workflow

In general, use exceptions only for 'unusual' conditions. Use normal `return` statements to pass control to the caller for conditions that are 'normal'.

This site is not ready yet! The updated version will be available soon.

[W2.4] RCS: Revision History ★

W2.4a ★★

Project Management → Revision Control → What

Can explain revision control

***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.

Revision Control Software

write your answer here...

In the context of RCS, what is a *Revision*? Give an example.

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. For example, take a file containing program code. If you modify the code and save the file, you have a new revision (or a version) of that file.

write your answer here...

- [ ] a. Help a single user manage revisions of a single file
- [ ] b. Help a developer recover from a incorrect modification to a code file
- [ ] c. Makes it easier for a group of developers to collaborate on a project
- [ ] d. Manage the drift between multiple versions of your project
- [ ] e. Detect when multiple developers make incompatible changes to the same file
- [ ] f. All of them are benefits of RCS

f

Suppose You are doing a team project with Tom, Dick, and Harry but those three have not even heard the term RCS. How do you explain RCS to them as briefly as possible, using the project as an example?

[W2.5] RCS: Remote Repos ★

W2.5a ★★

Project Management → Revision Control → Remote Repositories

Can explain remote repositories

Remote repositories are copies of a repo that are hosted on remote computers. 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 code base.

You can *clone* a remote repo onto your computer which will create a copy of a remot repo on your computer, including the version history as the remote repo.

You can *push* new commits in your clone to the remote repo which will copy the nev commits onto the remote repo. Note that pushing to a remote repo requires you to have write-access to it.

You can *pull* from the remote repos to receive new commits in the remote repo. Pulling is used to *sync* your local repo with latest changes to the remote repo.

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

A *fork* is a remote copy of a remote repo. If there is a remote repo that you want to push to but you do not have write access to it, you can fork the remote repo, which gives you your own remote repo that you can push to.

A *pull request* is mechanism for contributing code to a remote repo. It is a formal request sent to the maintainers of the repo asking them to pull your new code to their repo.

Here is a scenario that includes all the concepts introduced above (click on the slide to advance the animation):

W2.5b ★

Tools → Git and GitHub → Clone

Project Management → Revision Control → Remote Respositories

Can clone a remote repo

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 CLI

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

W2.5c ★

Tools → Git and GitHub → Pull

Tools → Git & GitHub → Clone

write your answer here...

W2.4b ★

Project Management → Revision Control → Repositories

Can explain 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 `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 directly, for examples, files of different projects.

In the context of RCS, what is a *repo*?

write your answer here...

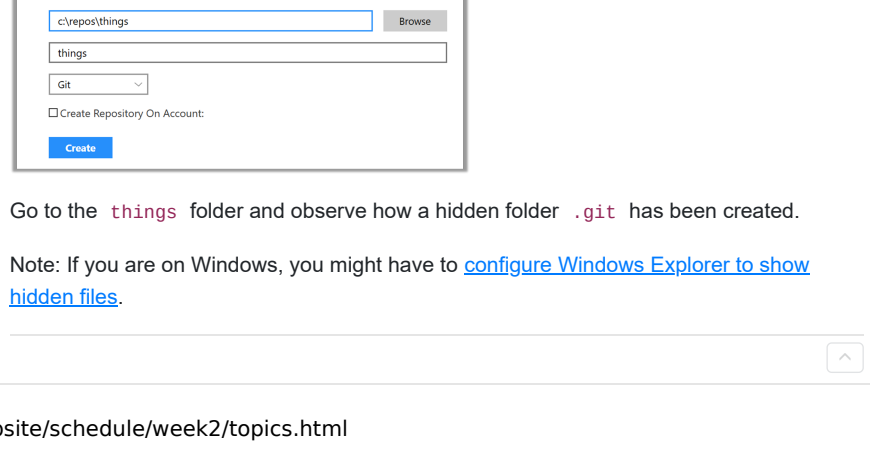
W2.4c ★

Tools → Git and GitHub → Init

Project Management → Revision Control → Repositories

Can create a local Git repo

Soon you are going to take your first step in using Git. If you would like to see a quick overview of the full Git landscape before jumping in, watch the video below.



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.

Suppose you want to create a repository in an empty directory `things`. Here are the steps:

SourceTree | CLI

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.

Note: If you are on Windows, you might have to configure Windows Explorer to show hidden files.

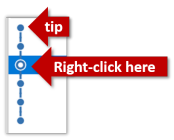
Can pull changes from a repo

Clone the sample repo as explained in [Textbook → Tools → Git & GitHub → Clone].

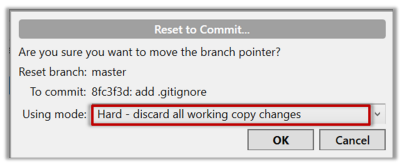
Delete the last two commits to simulate cloning the repo 2 commits ago.

SourceTree | CLI

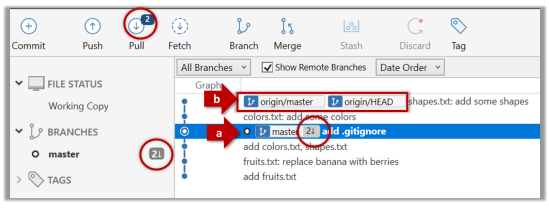
Right-click the target commit (i.e. the commit that is 2 commits behind the tip) and choose `Reset current branch to this commit`.



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



This is what you will see.



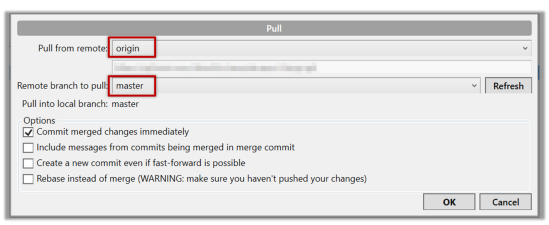
Note the following (cross refer the screenshot above):

Arrow marked as `a`: The local repo is now at this commit, marked by the `master` label.
Arrow marked as `b`: `origin/master` label shows what is the latest commit in the `master` branch in the remote repo.

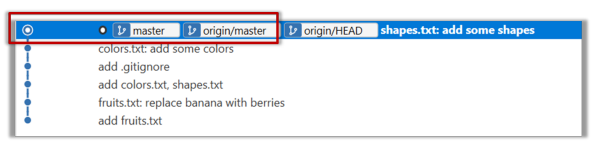
Now, your local repo state is exactly how it would be if you had cloned the repo 2 commits ago, as if somebody has added two more commits to the remote repo since you cloned it. To get those commits to your local repo (i.e. to sync your local repo with upstream repo) you can do a pull.

SourceTree | CLI

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.



W2.5d ★

Tools → Git and GitHub → Push

Tools → Git & GitHub → Pull

Can push to a remote repo

1. Create a GitHub account if you don't have one yet.
2. Fork the samplerepo-things to your GitHub account:
   - How to fork a repo?
3. Clone the fork (not the original) to your computer.
4. Create some commits in your repo.
5. Push the new commits to your fork on GitHub

SourceTree | CLI

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.

W2.4d ★

Project Management → Revision Control → Saving History

Can explain saving history

Tracking and Ignoring

In a repo, we 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, we first *stage* the specific changes we want to commit. This intermediate step allows us to commit only some changes while saving other changes for a later commit.

Identifying Points in History

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` .

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

W2.4e ★

Tools → Git and GitHub → Commit

Git & GitHub → Init

Can commit using Git

Create an empty repo.

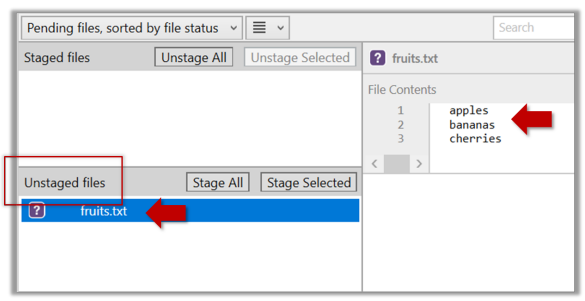
Create a file named `fruits.txt` in the working directory and add some dummy text to it.

> **Working directory**: The directory the repo is based in is called the *working directory*.

Observe how the file is detected by Git.

SourceTree | CLI

The file is shown as 'unstaged'



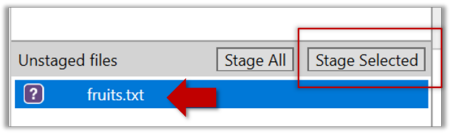
Although git has detected the file in the working directory, it will not do anything with the file unless you tell it to. Suppose we want to commit the current state of the file. First, we should stage the file.

> **Commit**: Saving the current state of the working folder into the Git revision history.

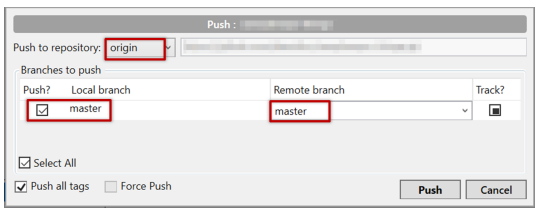
> **Stage**: Instructing Git to prepare a file for committing.

SourceTree | CLI

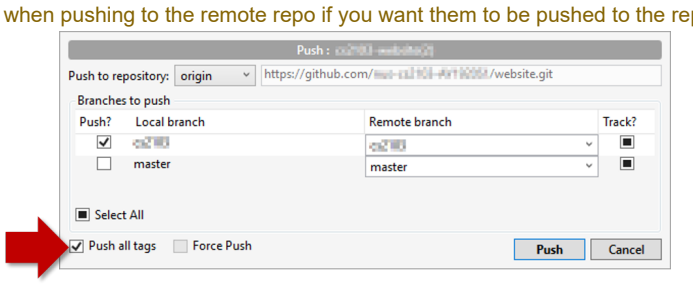
Select the `fruits.txt` and click on the `Stage Selected` button



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

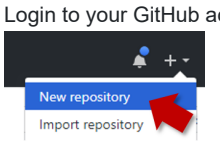


! Tags are not included in a normal push. Remember to tick `Push all tags` when pushing to the remote repo if you want them to be pushed to the repo.

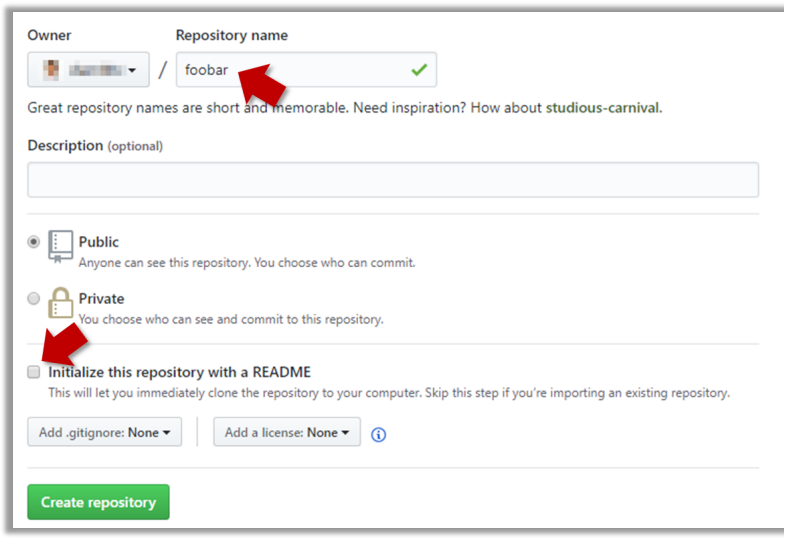


Pushing an existing local repo into a new remote repo on GitHub + extra
First, you need to create an empty remote repo on GitHub.

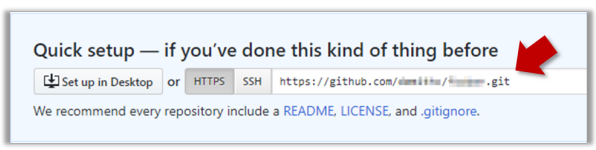
1. Login to your GitHub account and choose to create a new Repo.



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



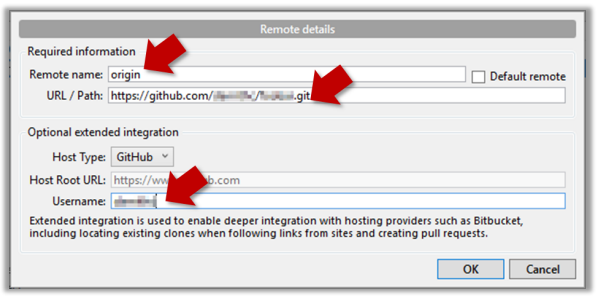
3. Note the URL of the repo. It will be of the form `https://github.com/{your_user_name}/{repo_name}.git`
e.g., `https://github.com/johndoe/foobar.git`



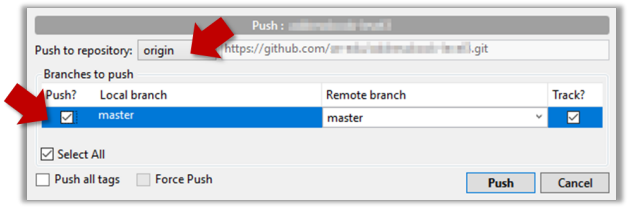
Next, you can push the existing local repo to the new remote repo as follows:

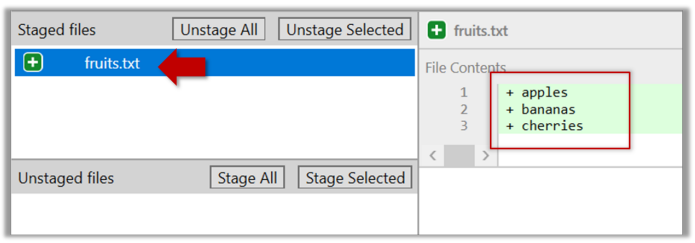
SourceTree | CLI

1. Open the local repo in SourceTree.
2. Choose `Repository` → `Repository Settings` menu option.
3. Add a new *remote* to the repo with the following values.
   - `Remote name` : the name you want to assign to the remote repo. Recommended `origin`
   - `URL/path` : the URL of your repo (ending in `.git`) that you collected earlier.
   - `Username` : your GitHub username



4. Now you can push your repo to the new remote the usual way.

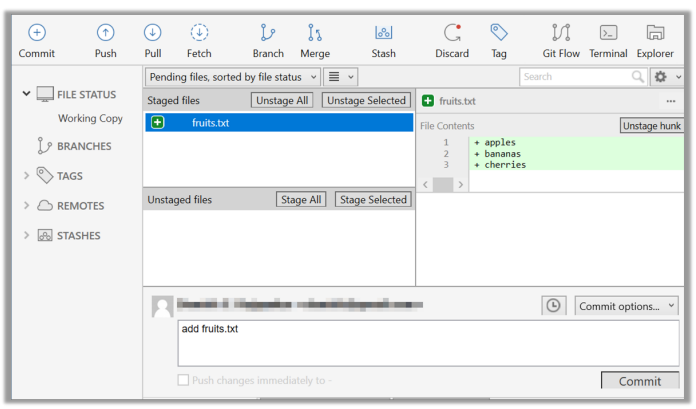




Now, you can commit the staged version of `fruits.txt`

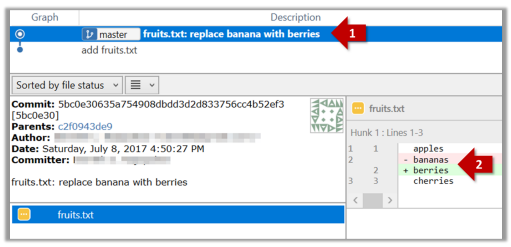
SourceTree | CLI

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

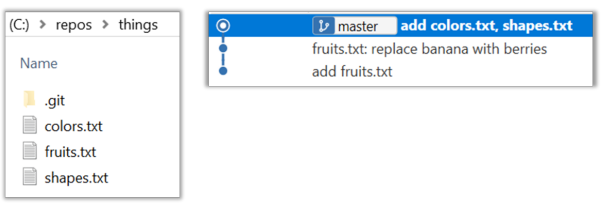


Note the existence of something called the `master` branch. Git allows you to have multiple branches (i.e. it is a way to evolve the content in parallel) and Git creates a default branch named `master` on which the commits go on by default.

Do 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.



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.



- Try Git is an online simulation/tutorial of Git basics. You can try its first few steps to

W2.4f ★★ Tools → Git and GitHub → Ignore

Revision Control: Saving History

Can set Git to ignore files

Add a file named `temp.txt` to the `things` repo you created. Suppose we don't want this file to be revision controlled by Git. Let's instruct Git to ignore `temp.txt`

SourceTree | CLI

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.

```
1 temp.txt
```

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 changed over time), simply commit it as you would commit any other file.
- To ignore it, follow the same steps we followed above when we set Git to ignore the `temp.txt` file.

W2.4g ★★ Project Management → Revision Control → Using History

Can explain using history

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

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., we can traverse the history of the working directory simply by checking out the commits we are interested in.

W2.4h ★★★

Tools → Git and GitHub → Tag

🎓 Project Management → Revision Control → Saving History

🏆 Can tag commits using Git

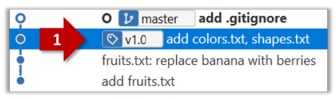
Let's tag a commit in a local repo you have (e.g. the `sampelrepo-things` repo)

SourceTree | CLI

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.



W2.4i ★★

Tools → Git and GitHub → Checkout

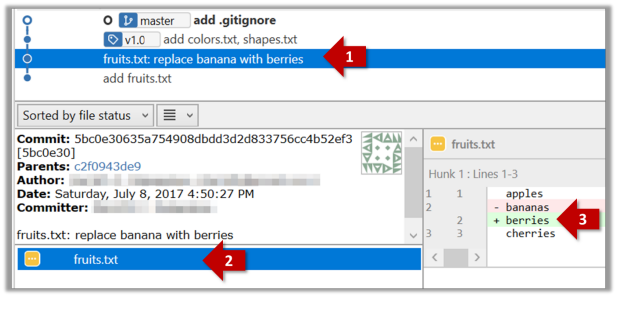
🎓 Project Management → Revision Control → Using History

🏆 Can load a specific version of a Git repo

Git can show you what changed in each commit.

SourceTree | CLI

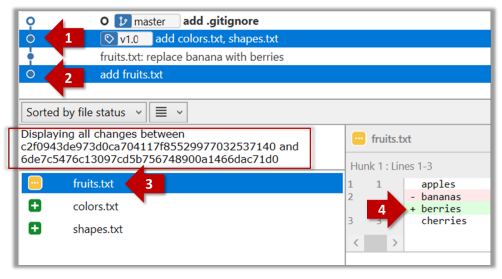
To see which files changed in a commit, click on the commit. To see what changed in a specific file in that commit, click on the file name.



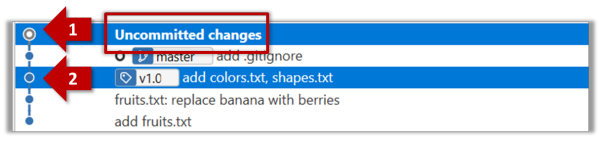
Git can also show you the difference between two points in the history of the repo.

SourceTree | CLI

Select the two points you want to compare using `Ctrl` + `Click` .



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.

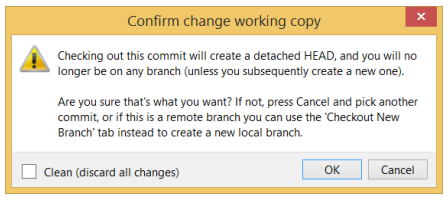


Git can load a specific version of the history to the working directory. Note that if you have uncommitted changes in the working directory, you need to stash them first to prevent them from being overwritten.

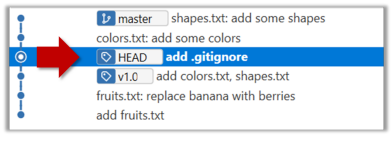
SourceTree | CLI

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

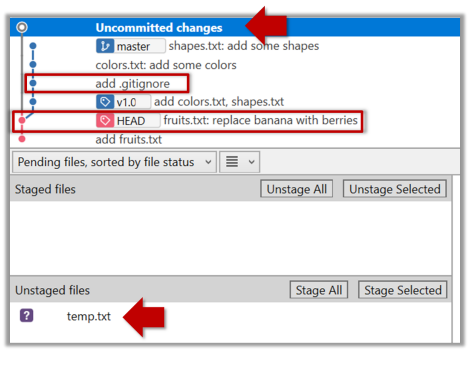
Click `OK` to the warning about 'detached HEAD' (similar to below).



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 come 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.

W2.4j ★★★★: OPTIONAL

Tools → Git and GitHub → Stash

This site is not ready yet! The updated version will be available soon.

Week 3 [Jan 27] - Topics

Detailed Table of Contents

[W3.1] Java ★★

W3.1a ★★ Implementation → Documentation → Tools → JavaDoc → What

Can explain JavaDoc

Javadoc is a tool for generating API documentation in HTML format from doc comments in source. In addition, modern IDEs use JavaDoc comments to generate explanatory tool tips.

An example method header comment in JavaDoc format (adapted from Oracle's Java documentation)

```
1  /**
2   * Returns an Image object that can then be painted on the screen.
3   * The url argument must specify an absolute {@link URL}. The name
4   * argument is a specifier that is relative to the url argument.
5   * <p>
6   * This method always returns immediately, whether or not the
7   * image exists. When this applet attempts to draw the image on
8   * the screen, the data will be loaded. The graphics primitives
9   * that draw the image will incrementally paint on the screen.
10  *
11  * @param url an absolute URL giving the base location of the image
12  * @param name the location of the image, relative to the url argument
13  * @return the image at the specified URL
14  * @see Image
15  */
16  public Image getImage(URL url, String name) {
17        try {
18            return getImage(new URL(url, name));
19        } catch (MalformedURLException e) {
20            return null;
21        }
22  }
```

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.

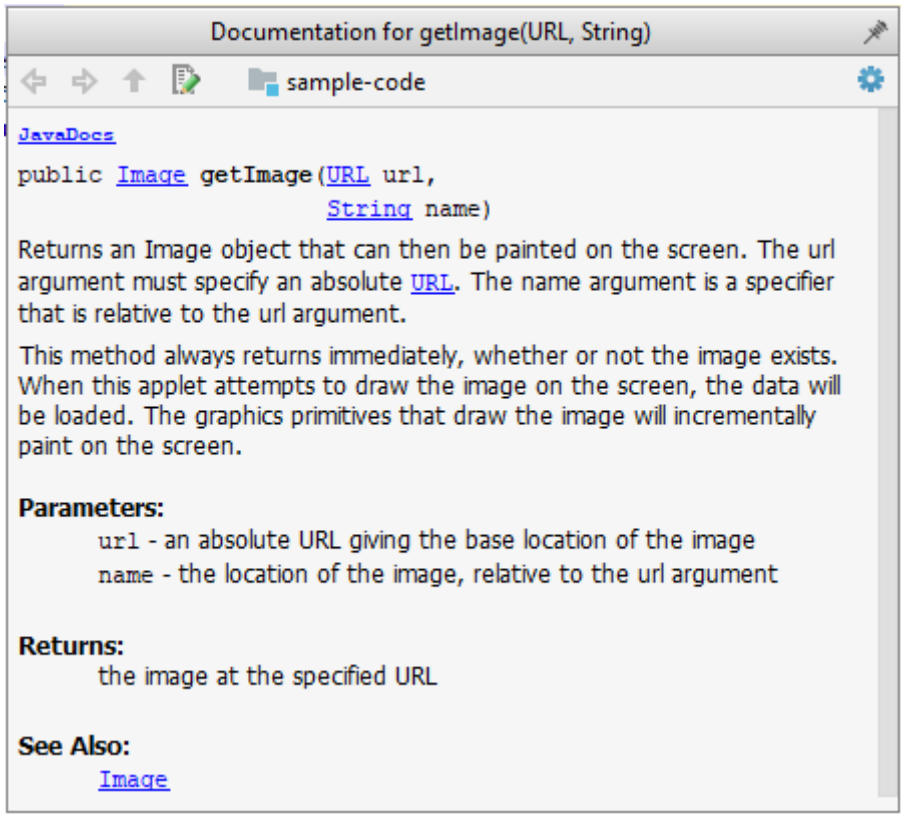
Parameters:
`url` - an absolute URL giving the base location of the image.

`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:



W3.1b ★★★

Implementation → Documentation → Tools → JavaDoc → How

Can write Javadoc comments

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:

```java
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{
    //...
```

W3.1c

C++ to Java → Miscellaneous Topics → File Access

Can read/write text files using Java

You can use the `java.io.File` class to represent a file object. It can be used to access properties of the file object.

This code creates a `File` object to represent a file `fruits.txt` that exists in the `data` directory relative to the current working directory and uses that object to print some properties of the file.

```java
import java.io.File;

public class FileClassDemo {

    public static void main(String[] args) {
        File f = new File("data/fruits.txt");
        System.out.println("full path: " + f.getAbsolutePath());
        System.out.println("file exists?: " + f.exists());
        System.out.println("is Directory?: " + f.isDirectory());
    }

}
```

↓

```
full path: C:\sample-code\data\fruits.txt
file exists?: true
is Directory?: false
```

If you use backslash to specify the file path in a Windows computer, you need to use an additional backslash as an escape character because the backslash by itself has a special meaning. e.g., use ✔ `"data\\fruits.txt"`, not ✖ `"data\fruits.txt"`. Alternatively, you can use forward slash ✔ `"data/fruits.txt"` (even on Windows).

You can read from a file using a `Scanner` object that uses a `File` object as the *source* of data.

This code uses a `Scanner` object to read (and print) contents of a text file line-by-line:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileReadingDemo {

        private static void printFileContents(String filePath) throws FileNotFoundException {
         File f = new File(filePath); // create a File for the given file path
         Scanner s = new Scanner(f); // create a Scanner using the File as the source
        while (s.hasNext()) {
            System.out.println(s.nextLine());
        }
    }

    public static void main(String[] args) {
        try {
            printFileContents("data/fruits.txt");
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
    }

}
```

⬇ i.e., contents of the `data/fruits.txt`

```
5 Apples
3 Bananas
6 Cherries
```

You can use a `java.io.FileWriter` object to write to a file.

📦 The `writeToFile` method below uses a `FileWrite` object to write to a file. The method is being used to write two lines to the file `temp/lines.txt` .

```java
import java.io.FileWriter;
import java.io.IOException;

public class FileWritingDemo {

     private static void writeToFile(String filePath, String textToAdd) throws IOException {
         FileWriter fw = new FileWriter(filePath);
         fw.write(textToAdd);
         fw.close();
    }

    public static void main(String[] args) {
         String file2 = "temp/lines.txt";
         try {
                writeToFile(file2, "first line" + System.lineSeparator() + "second line");
         } catch (IOException e) {
             System.out.println("Something went wrong: " + e.getMessage());
         }
    }

}
```

⬇ Contents of the `temp/lines.txt` :

```
first line
second line
```

Note that you need to call the `close()` method of the `FileWriter` object for the writing operation to be completed.

You can create a `FileWriter` object that appends to the file (instead of overwriting the current content) by specifying an additional boolean parameter to the constructor.

The method below appends to the file rather than overwrites.

```
1 private static void appendToFile(String filePath, String textToAppend)
  throws IOException {
2     FileWriter fw = new FileWriter(filePath, true); // create a FileWriter
  in append mode
3     fw.write(textToAppend);
4     fw.close();
5 }
```

The `java.nio.file.Files` is a utility class that provides several useful file operations. It relies on the `java.nio.file.Paths` file to generate `Path` objects that represent file paths.

This example uses the `Files` class to copy a file and delete a file.

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4
5  public class FilesClassDemo {
6
7      public static void main(String[] args) throws IOException{
8                                     Files.copy(Paths.get("data/fruits.txt"),
   Paths.get("temp/fruits2.txt"));
9          Files.delete(Paths.get("temp/fruits2.txt"));
10     }
11
12 }
```

The techniques above are good enough to manipulate simple text files. Note that **it is also possible to perform file I/O operations using other classes**.

W3.1d

C++ to Java → Miscellaneous Topics → Packages

Can use Java packages

You can organize your *types* (i.e., classes, interfaces, enumerations, etc.) into *packages* for easier management (among other benefits).

To create a package, you put a package statement at the very top of every source file in that package. The package statement must be the first line in the source file and there can be no more than one package statement in each source file. Furthermore, **the package of a type should match the folder path of the source file.** Similarly, the compiler will put the `.class` files in a folder structure that matches the package names.

The `Formatter` class below (in `<source folder>/seedu/tojava/util/Formatter.java` file) is in the package `seedu.tojava.util`. When it is compiled, the `Formatter.class` file will be in the location `<compiler output folder>/seedu/tojava/util`:

```
1  package seedu.tojava.util;
2
3  public class Formatter {
4      public static final String PREFIX = ">>";
5
6      public static String format(String s){
7          return PREFIX + s;
8      }
9  }
```

Package names are written in all lower case (not camelCase), using the dot as a separator. Packages in the Java language itself begin with `java.` or `javax.`. Companies use their reversed Internet domain name to begin their package names.

For example, `com.foobar.doohickey.util` can be the name of a package created by a company with a domain name `foobar.com`

To use a public package member from outside its package, you must do one of the following:

1. **Use the fully qualified name** to refer to the member
2. **Import** the package or the specific package member

The `Main` class below has two import statements:

- `import seedu.tojava.util.StringParser`: imports the class `StringParser` in the `seedu.tojava.util` package
- `import seedu.tojava.frontend.*`: imports all the classes in the `seedu.tojava.frontend` package

```
1  package seedu.tojava;
2
3  import seedu.tojava.util.StringParser;
4  import seedu.tojava.frontend.*;
5
6  public class Main {
7
8      public static void main(String[] args) {
9
10         // Using the fully qualified name to access the Processor class
11         String status = seedu.tojava.logic.Processor.getStatus();
12
13         // Using the StringParser previously imported
14         StringParser sp = new StringParser();
15
16         // Using classes from the tojava.frontend package
17         Ui ui = new Ui();
18         Message m = new Message();
19
20     }
21 }
```

Note how the class can still use the `Processor` without importing it first, by using its fully qualified name `seedu.tojava.logic.Processor`

Importing a package does not import its sub-packages, as packages do not behave as hierarchies despite appearances.

`import seedu.tojava.frontend.*` does not import the classes in the sub-package `seedu.tojava.frontend.widget`.

If you do not use a package statement, your type doesn't have a package -- a practice not recommended (except for small code examples) as it is not possible for a type in a package to import a type that is not in a package.

Optionally, a *static import* can be used to import static members of a type so that the imported members can be used without specifying the type name.

📦 The class below uses static imports to import the constant `PREFIX` and the method `format()` from the `seedu.tojava.util.Formatter` class.

```
1  import static seedu.tojava.util.Formatter.PREFIX;
2  import static seedu.tojava.util.Formatter.format;
3
4  public class Main {
5
6      public static void main(String[] args) {
7
8          String formatted = format("Hello");
9          boolean isFormatted = formatted.startsWith(PREFIX);
10         System.out.println(formatted);
11     }
12 }
```

▶ `Formatter` class

Note how the class can use `PREFIX` and `format()` (instead of `Formatter.PREFIX` and `Formatter.format()`).

When using the commandline to compile/run Java, you should take the package into account.

📦 If the `seedu.tojava.Main` class in defined in the file `Main.java` ,

- when compiling from the `<source folder>` , the command is:
  `javac seedu/tojava/Main.java`
- when running it from the `<compiler output folder>` , the command is:
  `java seedu.tojava.Main`

- Oracle's tutorial on packages: [What is a Package?] [Creating and Using Packages]

W3.1e : ★★★

C++ to Java → Miscellaneous Topics → Using JAR Files

🏆 Can use JAR files

Java applications are typically delivered as JAR (short for *Java Archive*) files. A JAR contains Java classes and other resources (icons, media files, etc.).

An executable JAR file can be launched using the `java -jar` command.

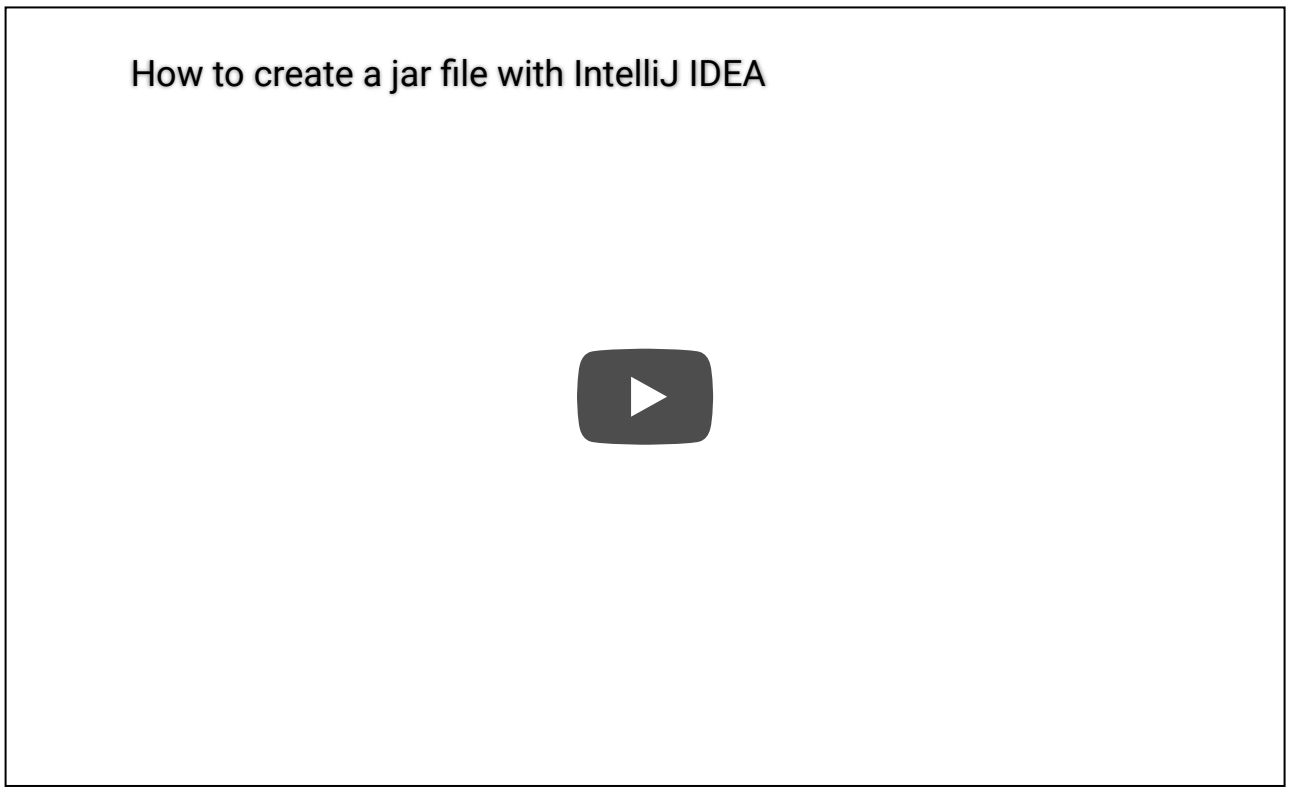
📦 `java -jar foo.jar` launches the `foo.jar` file.

📦 You can download the Collate-TUI.jar from https://se-edu.github.io/collate/ and run it using the command `java -jar Collate-TUI.jar` command. It's usage is given here.

The IDE can help you to package your application as a JAR file.

- Oracle's tutorial on JAR files

Creating a JAR file in Intellij - A video by Artur Spirin:



[W3.2] Coding Standards ★★

W3.2a ★★★

Implementation → Code Quality → Introduction → What

Can explain the importance of code quality

> 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 of software, poor quality code is something we cannot afford to tolerate.

W3.2b ★★

Implementation → Code Quality → Style → Introduction

Can explain the need for following a standard

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 location 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 use the same coding standard and

that standard is not generally inconsistent with typical industry practices. If a company's coding standards is very different from what is used typically 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.

What is the recommended approach regarding coding standards?

- a. Each developer should find a suitable coding standard and follow it in their coding.
- b. A developer should understand the importance of following a coding standard. However, there is no need to follow one.
- c. A developer should find out the coding standards currently used by the project and follow that closely.
- d. Coding standards are lame. Real programmers develop their own individual styles.

c

What is the aim of using a coding standard? How does it help?

write your answer here...

[W3.3] Developer Testing ★★

W3.3a ★★
Quality Assurance → Testing → Developer Testing → What

Can explain developer testing

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

W3.3b ★★★
Quality Assurance → Testing → Developer Testing → Why

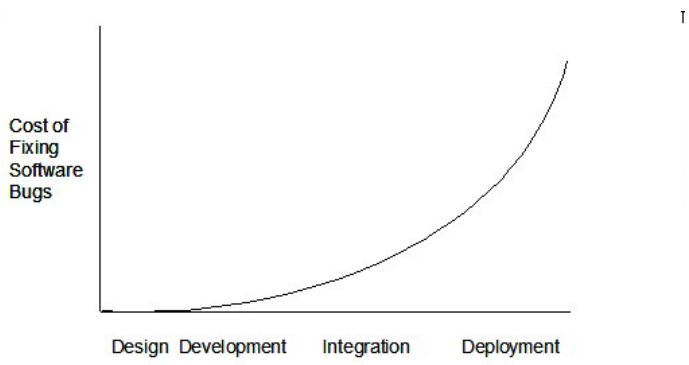
Can explain the need for early developer testing

Delaying testing until the full product is complete has a number of disadvantages:

- **Locating the cause of such a test case failure is difficult due to a large search space;** 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 during 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 were 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.

Discuss pros and cons of developers testing their own code.

Pros:

- Can be done early (the earlier we find a bug, the cheaper it is to fix).
- Can be done at lower levels, for examples, at operation and class level (testers usually test the system at UI level).
- It is possible to do more thorough testing because developers know the expected external behavior as well as the internal structure of the component.
- It forces developers to take responsibility for their own work (they cannot claim that "testing is the job of the testers").

Cons:

- A developer may subconsciously test only situations that he knows to work (i.e. test it too 'gently').
- A developer may be blind to his own mistakes (if he did not consider a certain combination of input while writing code, it is possible for him to miss it again during testing).
- A developer may have misunderstood what the SUT is supposed to do in the first place.
- A developer may lack the testing expertise.

write your answer here...

The cost of fixing a bug goes down as we reach the product release.

- ○ True
- ○ False

False. The cost goes up over time.

Explain why early testing by developers is important.

write your answer here...

[W3.4] **Unit Testing** ★

W3.4a ★★★

Quality Assurance → Testing → Test Automation → Test Automation Using Test Drivers

Can explain 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 the behavior is as expected.

`PayrollTest` 'drives' the `Payroll` class by sending it test inputs and verifies if the output is as expected.

```
1  public class PayrollTestDriver {
2      public static void main(String[] args) throws Exception {
3
4          //test setup
5          Payroll p = new Payroll();
6
7          //test case 1
8          p.setEmployees(new String[]{"E001", "E002"});
9          // automatically verify the response
10         if (p.totalSalary() != 6400) {
11             throw new Error("case 1 failed ");
12         }
13
14         //test case 2
15         p.setEmployees(new String[]{"E001"});
16         if (p.totalSalary() != 2300) {
17             throw new Error("case 2 failed ");
18         }
19
20         //more tests...
21
22         System.out.println("All tests passed");
23     }
24 }
```

W3.4b ★★★

Quality Assurance → Testing → Test Automation → Test Automation Tools

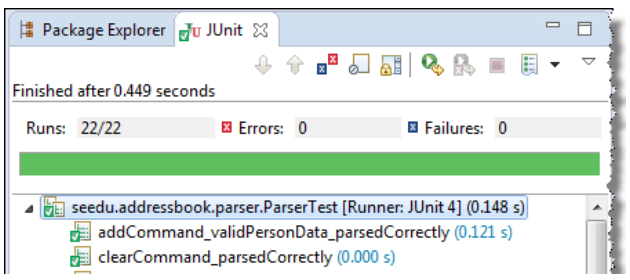
Can explain test automation tools

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 an automated test for a `Payroll` class, written using JUnit libraries.

```java
@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 has integrated support for testing tools. The figure below shows the JUnit output when running some JUnit tests using the Eclipse IDE.



W3.4c ★

Quality Assurance → Testing → Unit Testing → What

Can explain unit testing

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.

```java
class Foo{
    String read(){
        //...
    }

    void write(String input){
        //...
    }

}
```

```java
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(){
```

W3.4d ★★

C++ to Java → JUnit → JUnit: Basic

Can use simple JUnit tests

When writing JUnit tests for a class `Foo`, the common practice is to create a `FooTest` class, which will contain various test methods.

Suppose we want to write tests for the `IntPair` class below.

```java
public class IntPair {
    int first;
    int second;

    public IntPair(int first, int second) {
        this.first = first;
        this.second = second;
    }

    public int intDivision() throws Exception {
        if (second == 0){
            throw new Exception("Divisor is zero");
        }
        return first/second;
    }

    @Override
    public String toString() {
        return first + "," + second;
    }
}
```

Here's a `IntPairTest` class to match (using JUnit 5).

```java
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.fail;

public class IntPairTest {


    @Test
    public void testStringConversion() {
        assertEquals("4,7", new IntPair(4, 7).toString());
    }

    @Test
    public void intDivision_nonZeroDivisor_success() throws Exception {
        assertEquals(2, new IntPair(4, 2).intDivision());
        assertEquals(0, new IntPair(1, 2).intDivision());
        assertEquals(0, new IntPair(0, 5).intDivision());
    }

    @Test
    public void intDivision_zeroDivisor_exceptionThrown() {
        try {
            assertEquals(0, new IntPair(1, 0).intDivision());
            fail(); // the test should not reach this line
        } catch (Exception e) {
            assertEquals("Divisor is zero", e.getMessage());
        }
    }
}
```

Notes:

- Each test method is marked with a `@Test` annotation.
- Tests use `Assert.assertEquals(expected, actual)` methods to compare the expected output with the actual output. If they do not match, the test will fail. JUnit comes with other similar methods such as `Assert.assertNull` and `Assert.assertTrue` .
- Java code normally use camelCase for method names e.g., `testStringConversion` but when writing test methods, sometimes another convention is used: `whatIsBeingTested_descriptionOfTestInputs_expectedOutcome` e.g., `intDivision_zeroDivisor_exceptionThrown`
- There are several ways to verify the code throws the correct exception. The third test method in the example above shows one of the simpler methods. If the exception is thrown, it will be caught and further verified inside the `catch` block. But if it is not thrown as expected, the test will reach `Assert.fail()` line and will fail as a result.
- The easiest way to run JUnit tests is to do it via the IDE. For example, in Intellij you can right-click the folder containing test classes and choose 'Run all tests...'

Adding JUnit 5 to your IntelliJ Project -- by Kevintroko@YouTube

- JUnit Official User Guide
- JUnit 5 Tutorial – Common Annotations With Examples - a short tutorial
- How to test private methods in Java? [short answer] [long answer]

W3.4e ★★★

Quality Assurance → Testing → Unit Testing → Stubs

Quality Assurance → Testing → Unit Testing → What →

Can use stubs to isolate an SUT from its dependencies

A proper unit test requires the *unit* to be tested in isolation so that bugs in the dependencies cannot influence the test i.e. bugs outside of the unit should not affect the unit tests.

If a `Logic` class depends on a `Storage` class, unit testing the `Logic` class requires isolating the `Logic` class from the `Storage` class.

***Stubs* can isolate the SUT from its dependencies**.

Stub: A stub has the same interface as the component it replaces, but its implementation is so simple that it is unlikely to have any bugs. It mimics the responses of the component, but only for the a limited set of predetermined inputs. That is, it does not know how to respond to any other inputs. Typically, these mimicked responses are hard-coded in the stub rather than computed or retrieved from elsewhere, e.g. from a database.

Consider the code below:

```java
class Logic {
    Storage s;

    Logic(Storage s) {
        this.s = s;
    }

    String getName(int index) {
        return "Name: " + s.getName(index);
    }
}
```

```java
interface Storage {
    String getName(int index);
}
```

```java
class DatabaseStorage implements Storage {

    @Override
    public String getName(int index) {
        return readValueFromDatabase(index);
    }

    private String readValueFromDatabase(int index) {
        // retrieve name from the database
    }
}
```

Normally, you would use the `Logic` class as follows (note how the `Logic` object depends on a `DatabaseStorage` object to perform the `getName()` operation):

```java
Logic logic = new Logic(new DatabaseStorage());
String name = logic.getName(23);
```

You can test it like this:

```java
@Test
void getName() {
    Logic logic = new Logic(new DatabaseStorage());
    assertEquals("Name: John", logic.getName(5));
}
```

However, this `logic` object being tested is making use of a `DataBaseStorage` object which means a bug in the `DatabaseStorage` class can affect the test. Therefore, this test is not testing `Logic` *in isolation from its dependencies* and hence it is not a pure unit test.

Here is a stub class you can use in place of `DatabaseStorage` :

```java
class StorageStub implements Storage {

    @Override
    public String getName(int index) {
        if(index == 5) {
            return "Adam";
        } else {
            throw new UnsupportedOperationException();
        }
    }
}
```

Note how the `StorageStub` has the same interface as `DatabaseStorage` , is so simple that it is unlikely to contain bugs, and is pre-configured to respond with a hard-coded response, presumably, the correct response `DatabaseStorage` is expected to return for the given test input.

Here is how you can use the stub to write a unit test. This test is not affected by any bugs in the `DatabaseStorage` class and hence is a pure unit test.

```java
@Test
void getName() {
    Logic logic = new Logic(new StorageStub());
    assertEquals("Name: Adam", logic.getName(5));
}
```

In addition to Stubs, there are other type of replacements you can use during testing. E.g. *Mocks*, *Fakes*, *Dummies*, *Spies*.

- **Mocks Aren't Stubs** by Martin Fowler -- An in-depth article about how Stubs differ from other types of test helpers.

Stubs help us to test a component in isolation from its dependencies.

- ○ True
- ○ False

True

W3.4f ★★★

C++ to Java → JUnit → JUnit: Intermediate

Can use intermediate features of JUnit

Skim through the JUnit 5 User Guide to see what advanced techniques are available. If applicable, feel free to adopt them.

[W3.5] RCS: Branching ★★

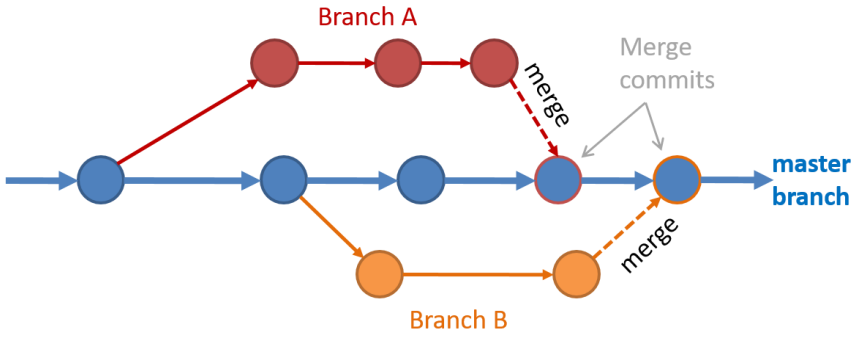
W3.5a ★★

Project Management → Revision Control → Branching

Can explain 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 result in a new commit that represents the changes done in the branch being merged.



Branching and merging

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

In the context of RCS, what is the *branching*? What is the need for branching?.

write your answer here...

In the context of RCS, what is the *merging* branches? How can it lead to *merge conflicts*?.

write your answer here...

W3.5b ★★

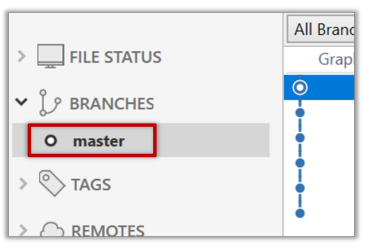
Tools → Git and GitHub → Branch

🎓 Project Management → Revision Control → Branching

Can use Git branching

0. Observe that you are normally in the branch called `master`. For this, you can take any repo you have on your computer (e.g. a clone of the samplerepo-things).

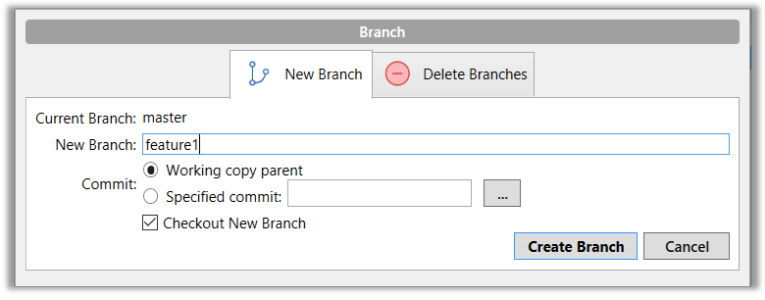
SourceTree CLI



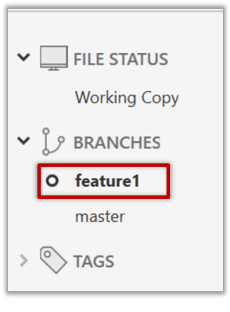
1. Start a branch named `feature1` and switch to the new branch.

SourceTree CLI

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.

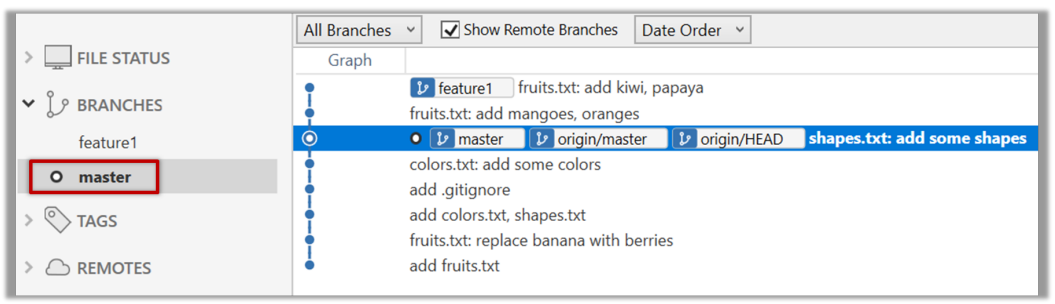


2. Create some commits in the new branch. Just commit as per normal. Commits you add while on a certain branch will become part of that branch.

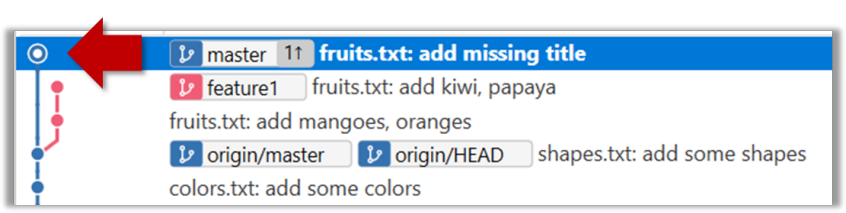
3. Switch to the `master` branch. Note how the changes you did in the `feature1` branch are no longer in the working directory.

SourceTree | CLI

Double-click the `master` branch

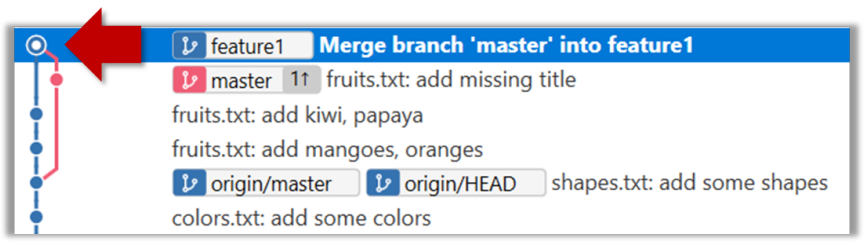


4. Add a commit to the master branch. Let's imagine it's a bug fix.



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 below. Also note how Git has created a *merge commit*.



SourceTree | CLI

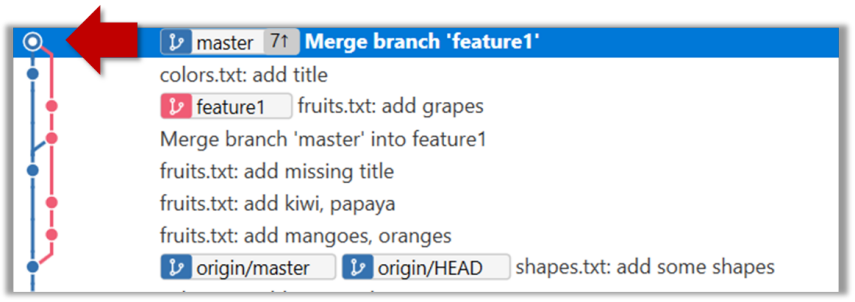
Right-click on the `master` branch and choose `merge master into the current branch`. Click `OK` in the next dialog.

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.

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 and end-result like this:

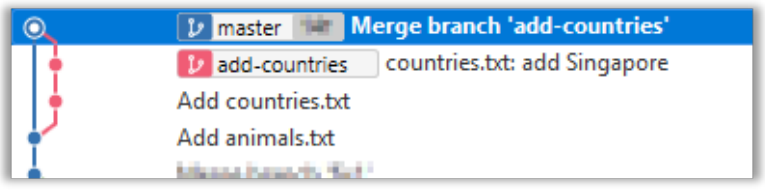


SourceTree | CLI

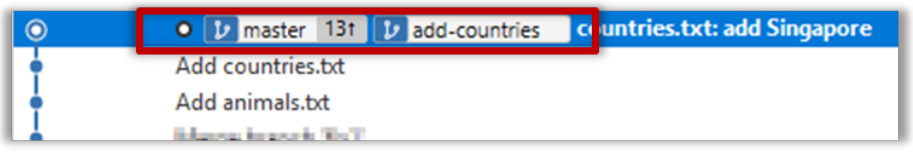
Right-click on the `feature1` branch and choose `Merge...`.

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:

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 below,



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

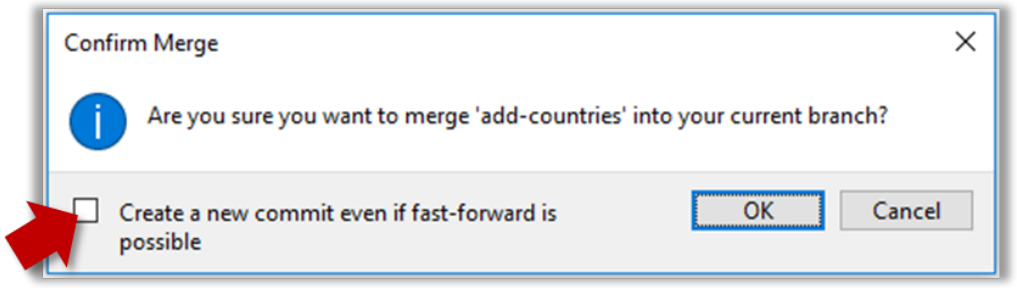


That is because **Git does a *fast forward* merge if possible**. Seeing that the `master` branch has not changed since you started the `add-countries` branch, Git has decided it is simpler to just put the commits of the `add-countries` branch in front of the `master` branch, without going into the trouble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

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

SourceTree | CLI

Tick the box shown below when you merge a branch:



W3.5c ★★

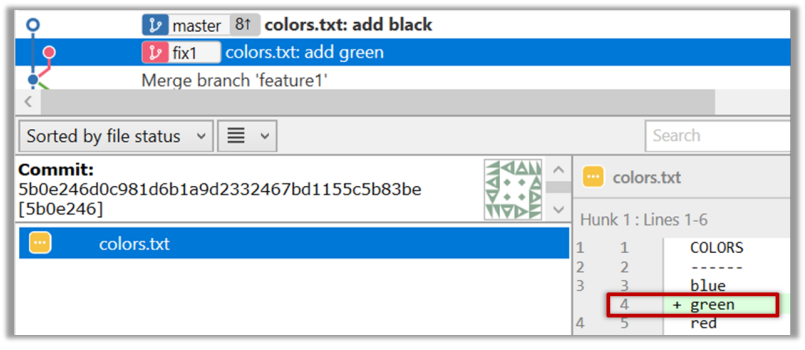
Tools → Git and GitHub → Merge Conflicts

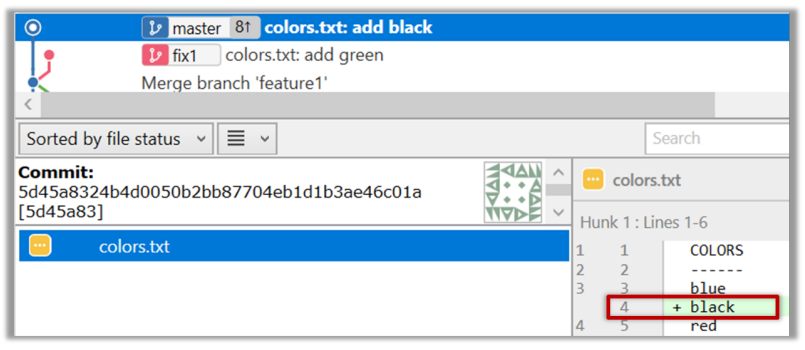
Tools → Git & GitHub → Branching

Can use Git to resolve merge conflicts

1. Start a branch named `fix1` in a local 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.





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 `=======`.

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

```

<<<<<<< HEAD
black
=======

```

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

```

=======
green
>>>>>>> fix1

```

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.

[W3.6] RCS: Creating Pull Requests ★

W3.6a ★

Tools → Git and GitHub → Create PRs

🎓 Tools → Git & GitHub → Branching

🏆 Can create PRs on GitHub

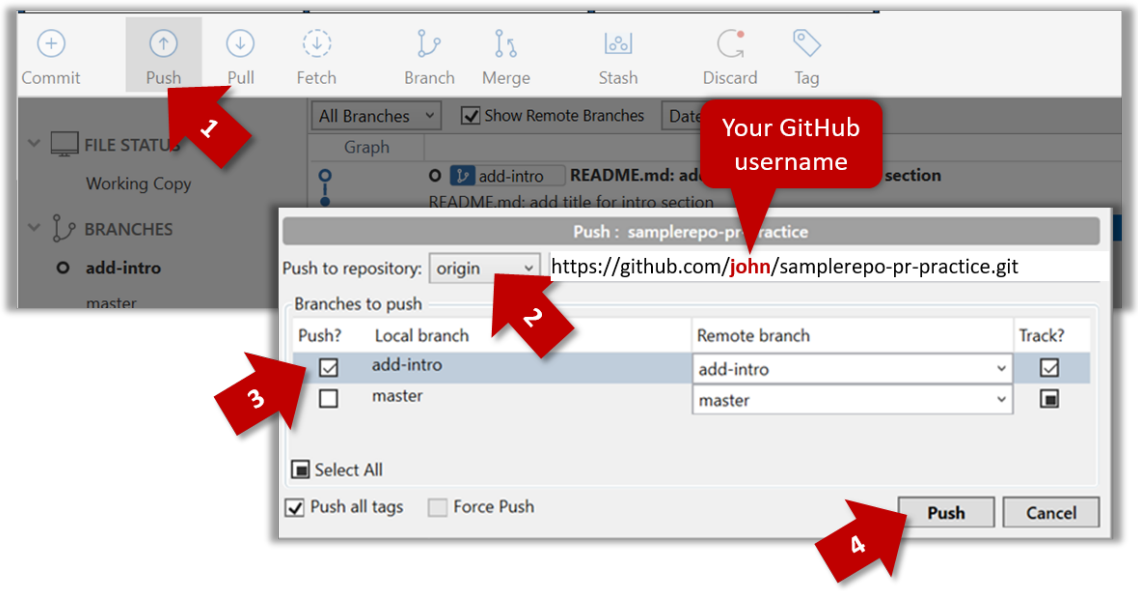
1. Fork the samplerepo-pr-practice onto your GitHub account. Clone it onto your computer.

2. Create a branch named `add-intro` in your clone. Add a couple of commits which adds/modifies an *Introduction* section to the `README.md` . Example:

```
1
2 # Introduction
3 Creating Pull Requests (PRs) is needed when using RCS in a multi-person projects.
4 This repo can be used to practice creating PRs.
5
```

3. Push the `add-intro` branch to your fork.

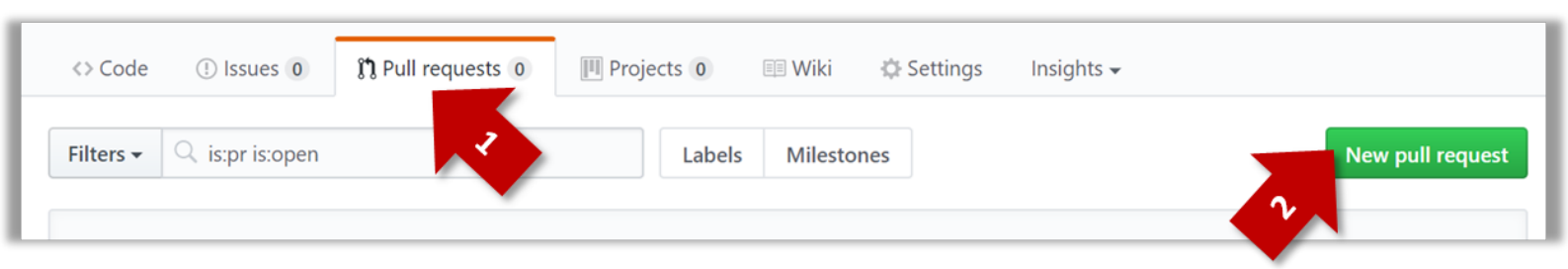
SourceTree | CLI



---

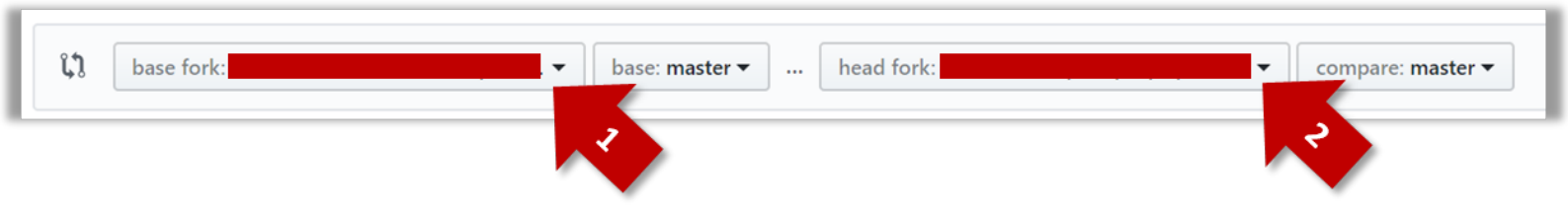
4. Create a Pull Request from the `add-intro` branch in your fork to the `master` branch of the same fork (i.e. `your-user-name/samplerepo-pr-practice` , not `se-edu/samplerepo-pr-practice`), as described below.

4a. Go to the GitHub page of your fork (i.e. `https://github.com/{your_username}/samplerepo-pr-practice`), click on the `Pull Requests` tab, and then click on `New Pull Request` button.



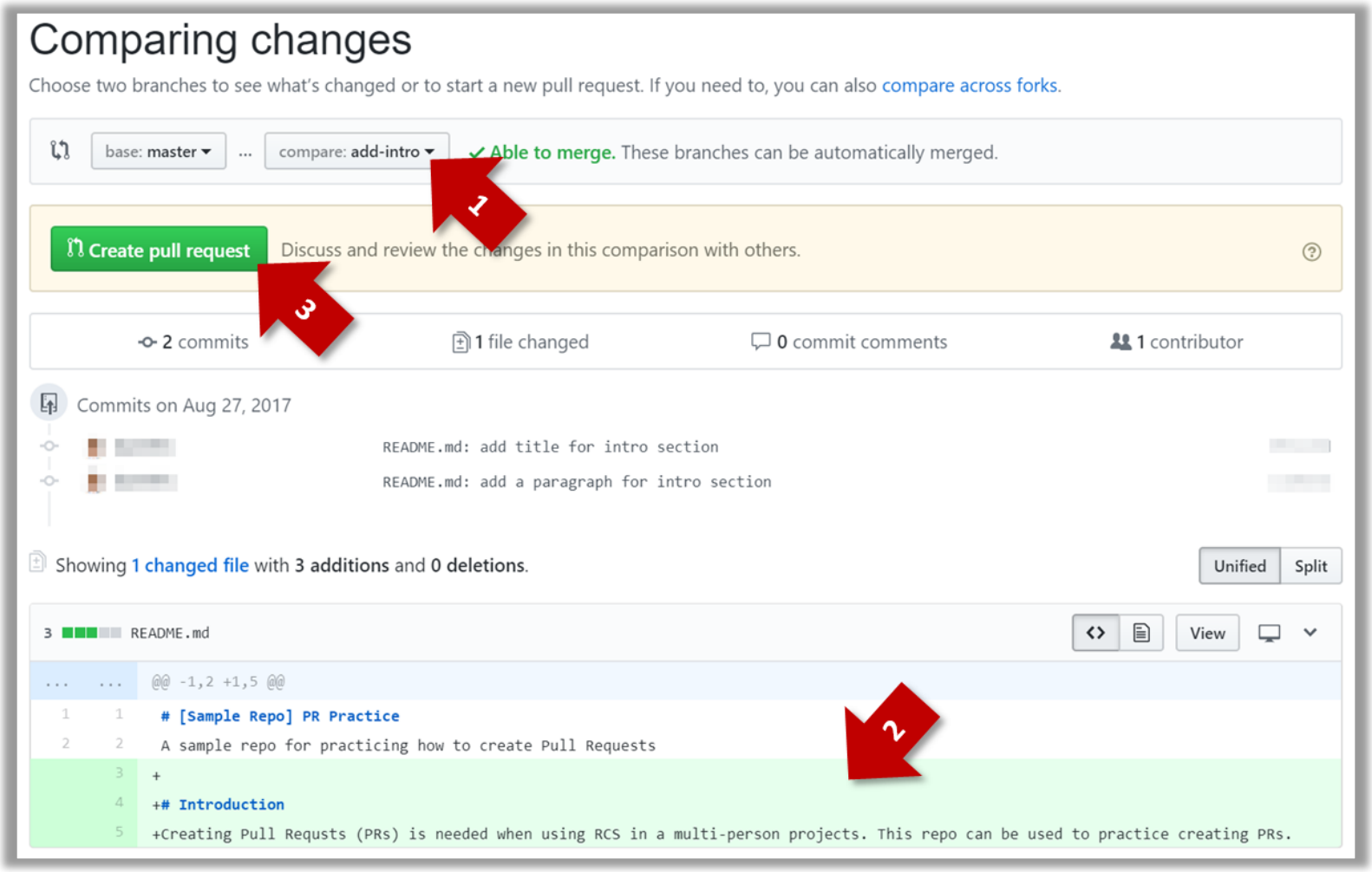
4b. Select `base fork` and `head fork` as follows:

- `base fork` : your own fork (i.e. `{your user name}/samplerepo-pr-practice` , NOT `se-edu/samplerepo-pr-practice`)
- `head fork` : your own fork.

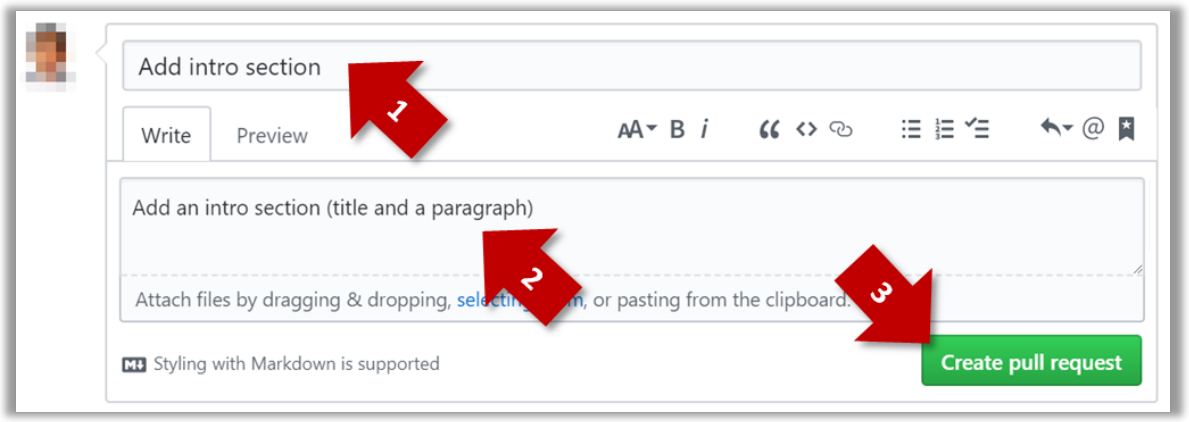


> ℹ The *base fork* is where changes should be applied. The *head fork* contains the changes you would like to be applied.

4c. (1) Set the base branch to `master` and head branch to `add-intro` , (2) confirm the *diff* contains the changes you propose to merge in this PR (i.e. confirm that you did not accidentally include extra commits in the branch), and (3) click the `Create pull request` button.



4d. (1) Set PR name, (2) set PR description, and (3) Click the `Create pull request` button.

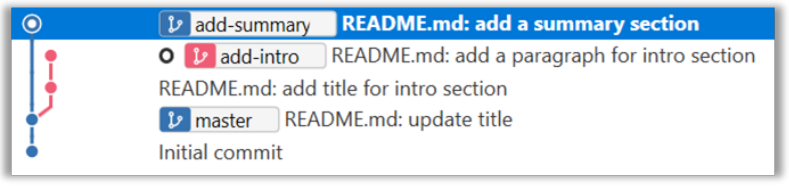


> A common newbie mistake when creating branch-based PRs is to mix commits of one PR with another. To learn how to avoid that mistake, you are encouraged to continue and create another PR as explained below.

5. In your local repo, create a new branch `add-summary` off the `master` branch.

> When creating the new branch, it is very important that you switch back to the `master` branch first. If not, the new branch will be created off the current branch `add-intro` . And that is how you end up having commits of the first PR in the second PR as well.

6. Add a commit in the `add-summary` branch that adds a *Summary* section to the `README.md` , in exactly the same place you added the *Introduction* section earlier.



7. Push the `add-summary` to your fork and create a new PR similar to before.

- GitHub's own documentation on creating a PR

This site was built with MarkBind 2.14.1 at Tue, 21 Apr 2020, 6:01:40 UTC

This site is not ready yet! The updated version will be available soon.

Week 4 [Feb 3] - Topics

Detailed Table of Contents

[W4.1] Design: Models ★★

W4.1a ★★ Design → Modelling → Introduction → What

Can explain models

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.

W4.1b ★★★ Design → Modelling → Introduction → How

Can explain how models are used

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 documentations.

Some examples of using models to communicate:

1. We 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 to as blueprints:

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

Model Driven Development + extra

Choose the correct statements about models.

- [ ] a. Models are abstractions.
- [ ] b. Models can be used for communication.
- [ ] c. Models can be used for analysis of a problem.
- [ ] d. Generating models from code is useless.
- [ ] e. Models can be used as blueprints for generating code.

(a) (b) (c) (e)

Explanation: Models generated from code can be used for understanding, analysing, and communicating about the code.

Explain how models (e.g. UML diagrams) can be used in a class project.

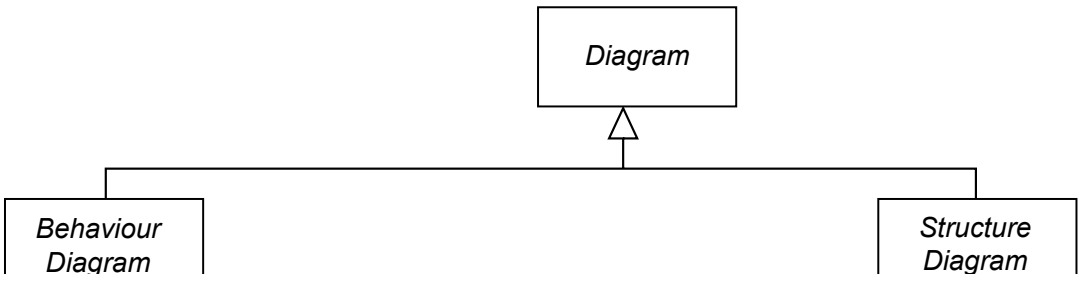
Can models be useful in evaluating the design quality of a software written by students?

W4.1c ★★★

Design → Modelling → Introduction → UML Models

Can identify UML models

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



[W4.2] Class/Object Diagrams: Basics ★

W4.2a ★

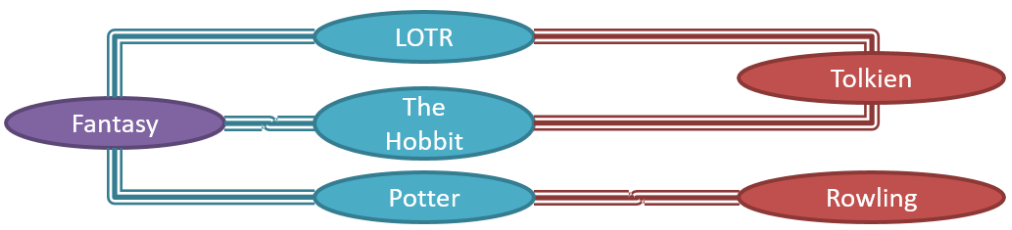
Design → Modelling → Modelling Structure → OO Structures

🎓 Design → OOP → Classes → Basic

🏆 **Can explain structure modelling of OO solutions**

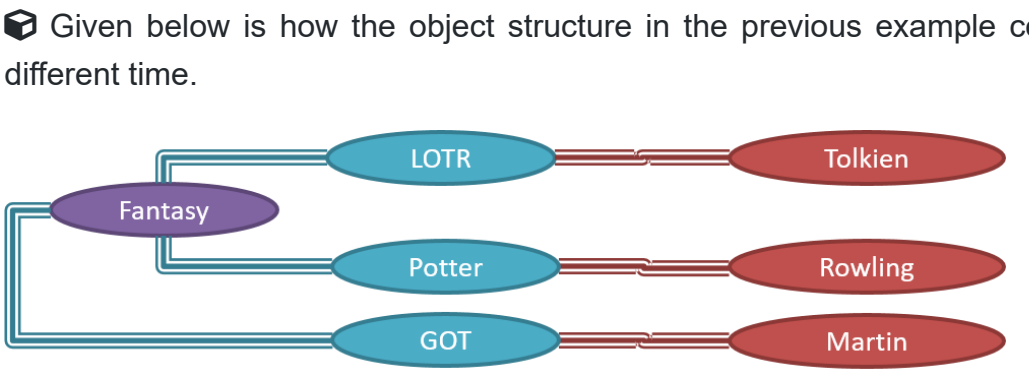
An OO solution is basically a network of objects interacting with each other. Therefore, **it is useful to be able to model how the relevant objects are 'networked' together** inside a software i.e. how the objects are connected together.

📦 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.

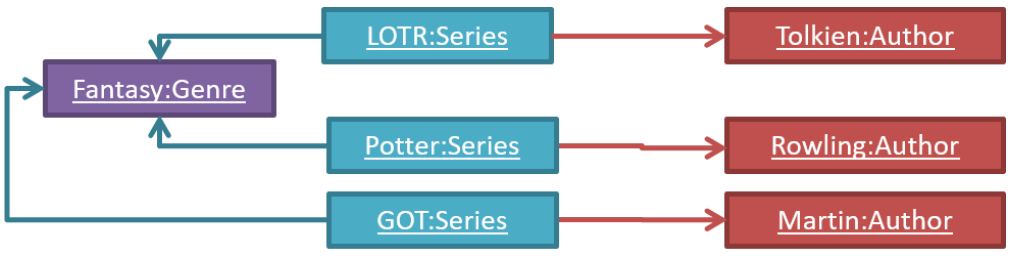


However, object structures do not change at random; they change based on a set of rules, as was decided by the designer of that software. Those **rules that object structures need to follow can be illustrated as a *class structure*** i.e. a structure that exists among the relevant classes.

📦 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:

W4.2b ★

Design → Modelling → Modelling Structure → Class Diagrams (Basics)

Can use basic-level class diagrams

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.

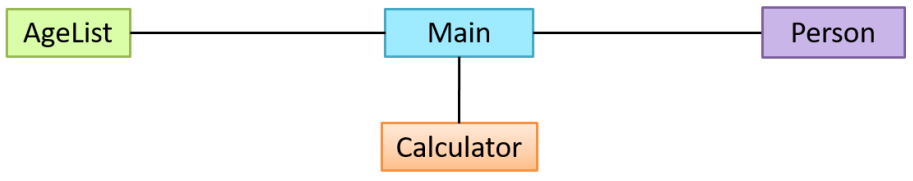
- UML Class Diagrams → Classes → What
- UML Class Diagrams → Class-Level Members → What

***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.

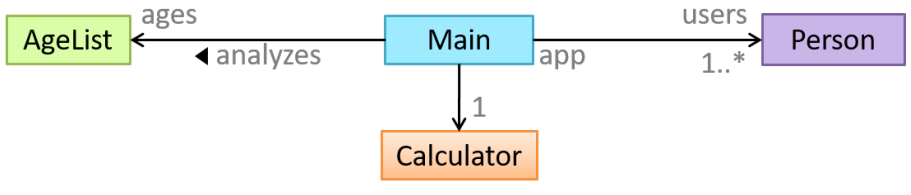


In addition, **associations can show additional decorations such as *association labels*, *association roles*, *multiplicity* and *navigability*** to add more information to a class diagram.

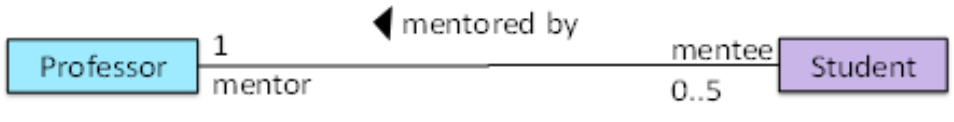
- UML Class Diagrams → Associations → Labels
- UML Class Diagrams → Associations → Roles
- OOP Associations → Multiplicity

- 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:



Which association notations are shown in this diagram?

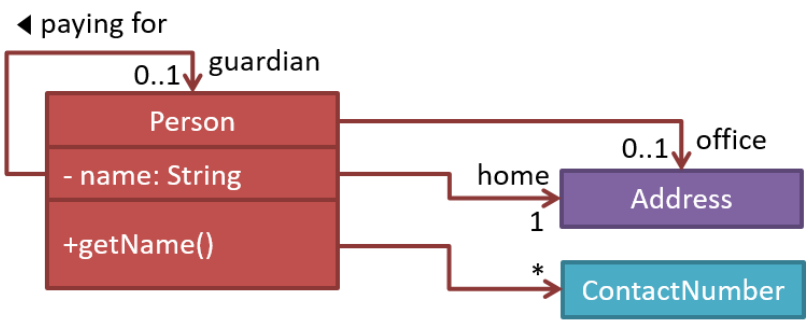


- a. association labels
- b. association roles
- c. association multiplicity
- d. class names

(a) (b) (c) (d)

Explanation: '1' is a *multiplicity*, 'mentored by' is a *label*, and 'mentor' is a *role*.

Explain the associations, navigabilities, and multiplicities in the class diagram below:



Draw a class diagram for the code below. Show the attributes, methods, associations, navigabilities, and multiplicities in the class diagram below:

```
class Box {
    private Item[] parts = new Item[10];
    private Item spareItem;
    private Lid lid; // lid of this box
    private Box outerBox;

    public void open(){
        //...
    }
}
```

```
class Item {
    public static int totalItems;
}
```

```
class Lid {
    Box box; // the box for which this is the lid
}
```

W4.2c ★

Design → Modelling → Modelling Structure → Object Diagrams

Can use basic 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

This question is based on the following question from another topic:

Draw a class diagram for the code below. Show the attributes, methods, associations, navigabilities, and multiplicities in the class diagram below:

```
class Box {
    private Item[] parts = new Item[10];
    private Item spareItem;
    private Lid lid; // lid of this box
    private Box outerBox;

    public void open(){
        //...
    }
}
```

```
class Item {
    public static int totalItems;
}
```

```
class Lid {
    Box box; // the box for which this is the lid
}
```

Draw an object diagram to match the code. Include objects of all three classes in your object diagram.

W4.2d ★★

Tools → UML → Object vs Class Diagrams

Can distinguish between class diagrams and object diagrams

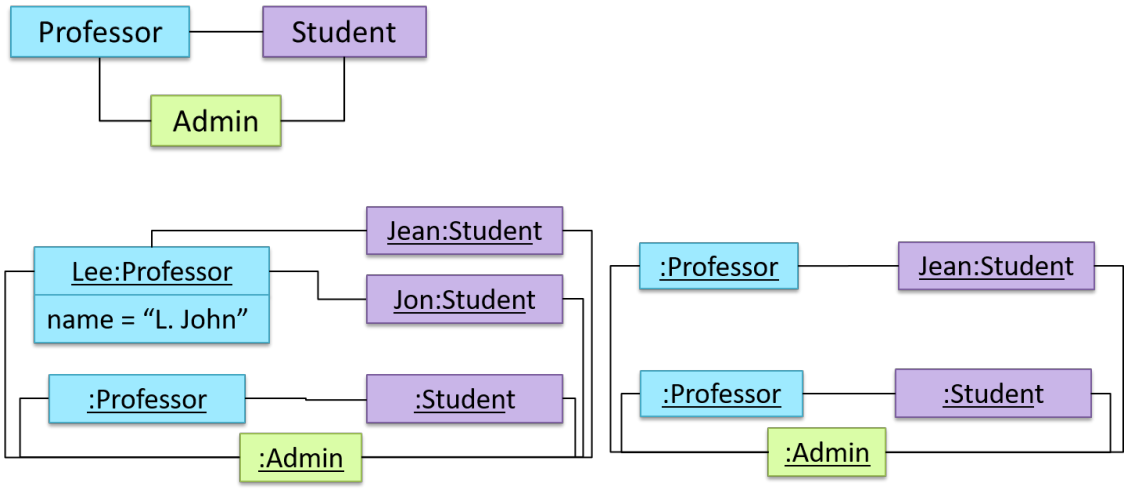
Compared to the notation for a class diagrams, object diagrams differ in the following ways:

- Shows 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

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,

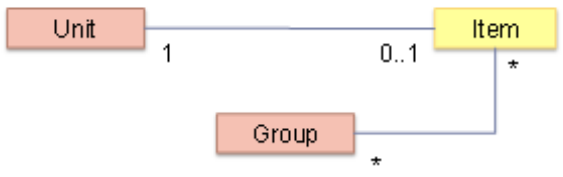
each of these object diagrams shows 'an instance of' the same class diagram.



Which of these class diagrams match the given object diagram?

(1)

(2)



- [ ] 1
- [ ] 2

(1) (2)

Explanation: Both class diagrams allow one `Unit` object to be linked to one `Item` object.

[W4.3] Class Diagrams: Intermediate-Level ★★

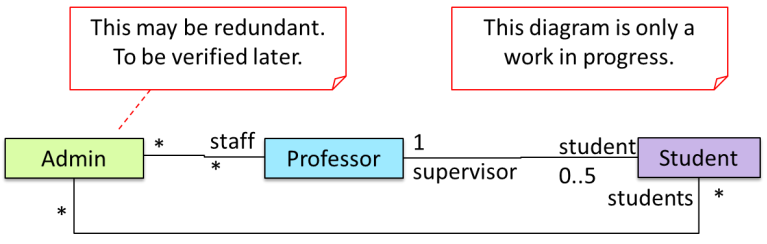
W4.3a ★★★

Tools → UML → Notes

Can use UML 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:



W4.3b ★★★★: OPTIONAL

Tools → UML → Constraints

W4.3c ★★★

Tools → UML → Class Diagrams → Associations as Attributes

Can show an association as an attribute

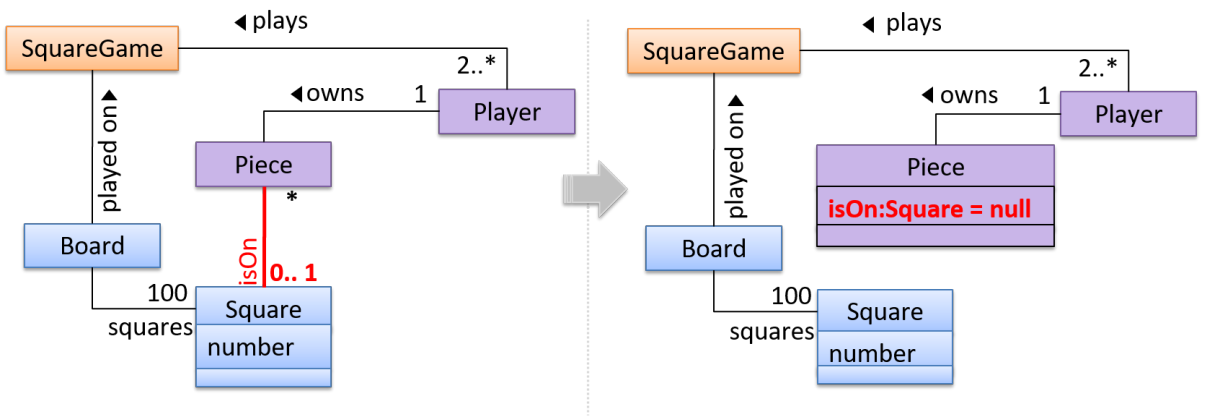
An association can be shown as an attribute instead of a line.

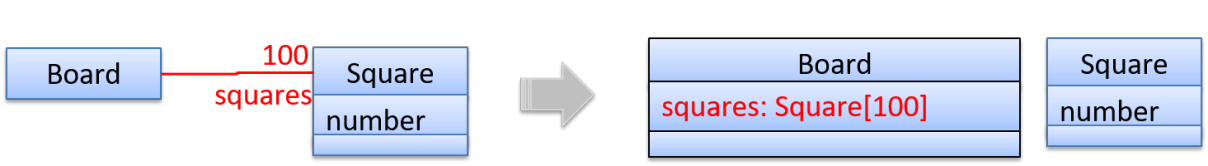
Association multiplicities and the default value too can be shown as part of the attribute using the following notation. Both are optional.

```
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`.





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

W4.3d ★★

Design → Modelling → Modelling Structure → Class Diagrams - Intermediate

Design → Modeling → Class Diagrams (Basic)

Can use intermediate-level class diagrams

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

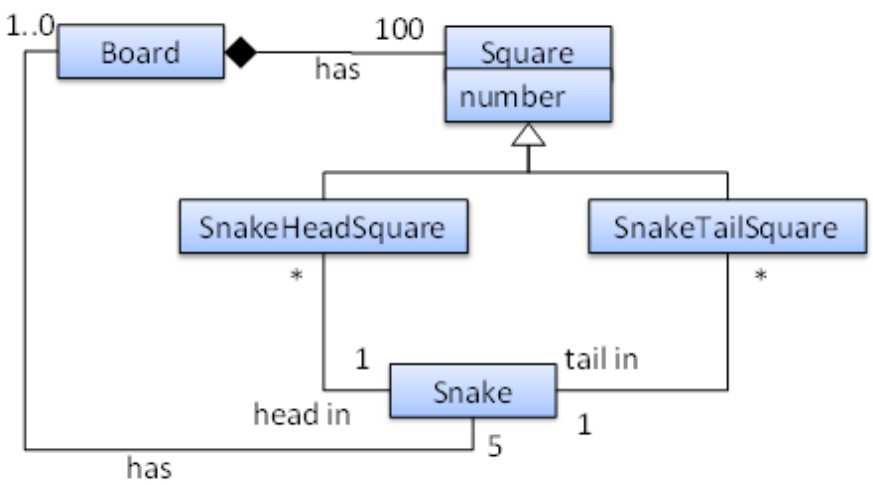
Modeling abstract classes

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

Modeling interfaces

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

Which of these statements match the class diagram?



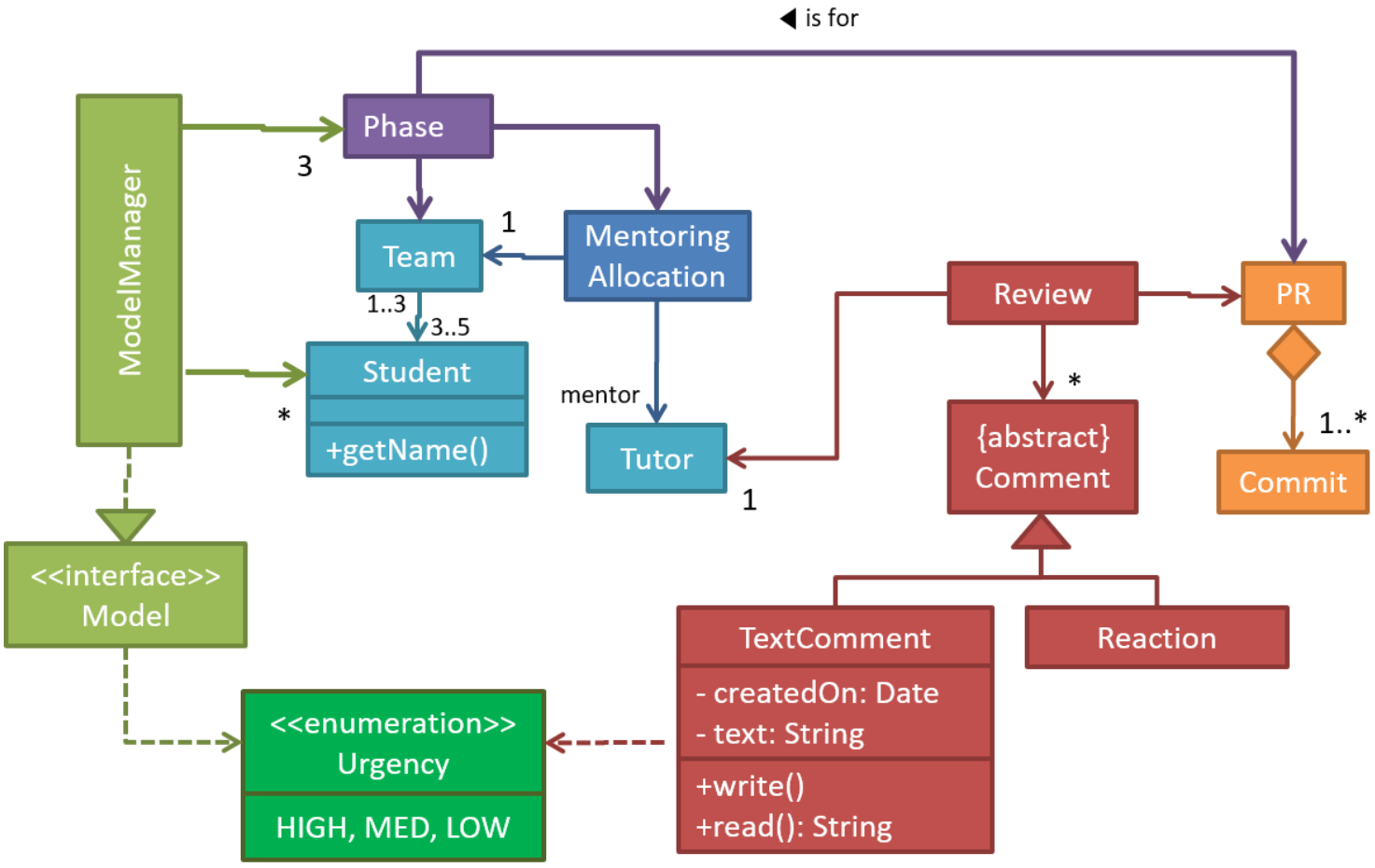
- a. A `Snake` must belong to at least one `Board` .
- b. A `SnakeHeadSquare` can contain only one `Snake` head.
- c. A `Square` can contain a `Snake` head.
- d. A `Snake` head can be in more than one `SnakeHeadSquare`
- e. The `Board` has exactly 5 `Snake` s.

~~(a)~~(b)(c)(d)(e)

Explanation:

(a) does not match because a `Snake` may or may not belong to a `Board` (multiplicity is `0..1`)
(b) matches the diagram because the multiplicity given in `1`
(c) matches the diagram because `SnakeHeadSquare` is a `Square` (due to inheritance)
(d) matches the diagram because the multiplicity given is `*`
(e) matches the diagram because the multiplicity given in `5`

Explain the meaning of various class diagram notations in the following class diagram:



Consider the code below:

```
1 public interface Billable {
2     void bill();
3 }
```

```
1 public enum Rating {
2     GOOD, OK, POOR
3 }
```

```
1 public abstract class Item
2          implements Billable {
3     public abstract void print();
4 }
```

```
1 public class Review {
2     private final Rating rating;
3
4     public Review(Rating rating) {
5         this.rating = rating;
6     }
7 }
```

```java
public class StockItem extends Item
{
    private Review review;
    private String name;

    public StockItem(
         String name, Rating rating)
{

        this.name = name;
                this.review  =  new
Review(rating);
    }

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

```java
import java.util.List;

public class Inventory {
    private List<Item> items;

    public int getItemCount(){
        return items.size();
    }

                        public     void
generateBill(Billable b){
        // ...
    }

    public void add(Item s) {
        items.add(s);
    }
}
```

W4.3e ★★★

Paradigms → OOP → Associations → Association Classes

Can explain the meaning of association classes

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

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

UML Class Diagrams → Association Classes → What

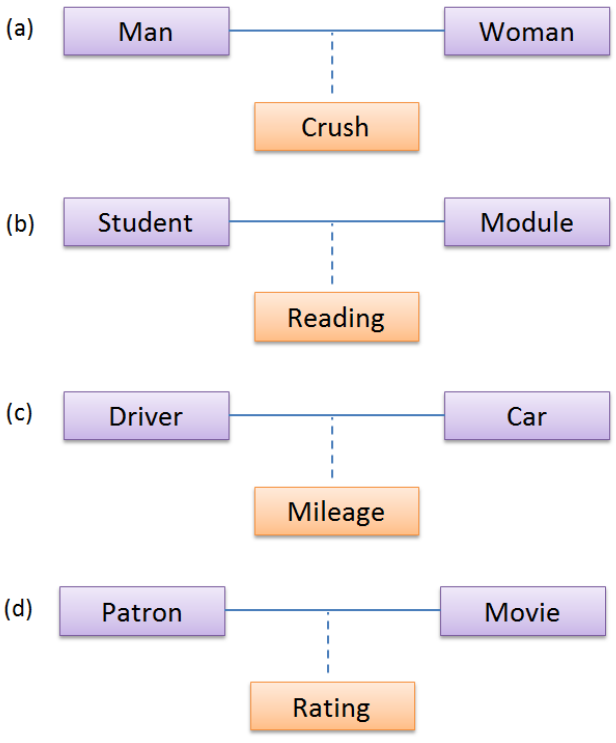
Implementing association classes

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

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

```java
class Transaction{

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

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

Which one of these is the suitable as an Association Class?



- [ ] a
- [ ] b
- [ ] c
- [ ] d

(a)(b)~~(c)~~(d)

Explanation: Mileage is a property of the car, and not specifically about the association between the Driver and the Car. If Mileage was defined as the total number of miles that car was driven by that driver, then it would be suitable as an association class.

[W4.4] **Java: JavaFX** ★★★

W4.4a ★★★

C++ to Java → Miscellaneous Topics → JavaFX

Can use JavaFX to build a simple GUI

JavaFX is a technology for building Java-based GUIs. Previously it was a part Java itself, but has become a third-party dependency since then. It is now being maintained by OpenJDK.

JavaFx 9 tutorial

Adapted (with permissions) from Marco Jakob's JavaFX 8 tutorial. Thanks to Marco Jakob for allowing us to adapt his tutorials for IntelliJ.

Table of contents:

After going through the two parts, you should be familiar with working with IntelliJ. You can continue with the original tutorial (which is written for Eclipse), with the following links:

- Part 3: Interacting with the User
- Part 4: CSS Styling
- Part 5: Storing Data as XML

[W4.5] Java: varargs

W4.5a ★★★

C++ to Java → Miscellaneous Topics → Varargs

Can use Java varargs feature

Variable Arguments (Varargs) is a *syntactic sugar* type feature that allows writing a methods that can take a variable number of arguments.

The `search` method below can be called as `search()`, `search("book")`, `search("book", "paper")`, etc.

```
public static void search(String ... keywords){
   // method body
}
```

Resources:

- Java Varargs feature (from Oracle.com)
- Java Varargs tutorial (for javaTpoint.com)

[W4.6] RCS: Managing Pull Requests ★★

W4.6a ★★

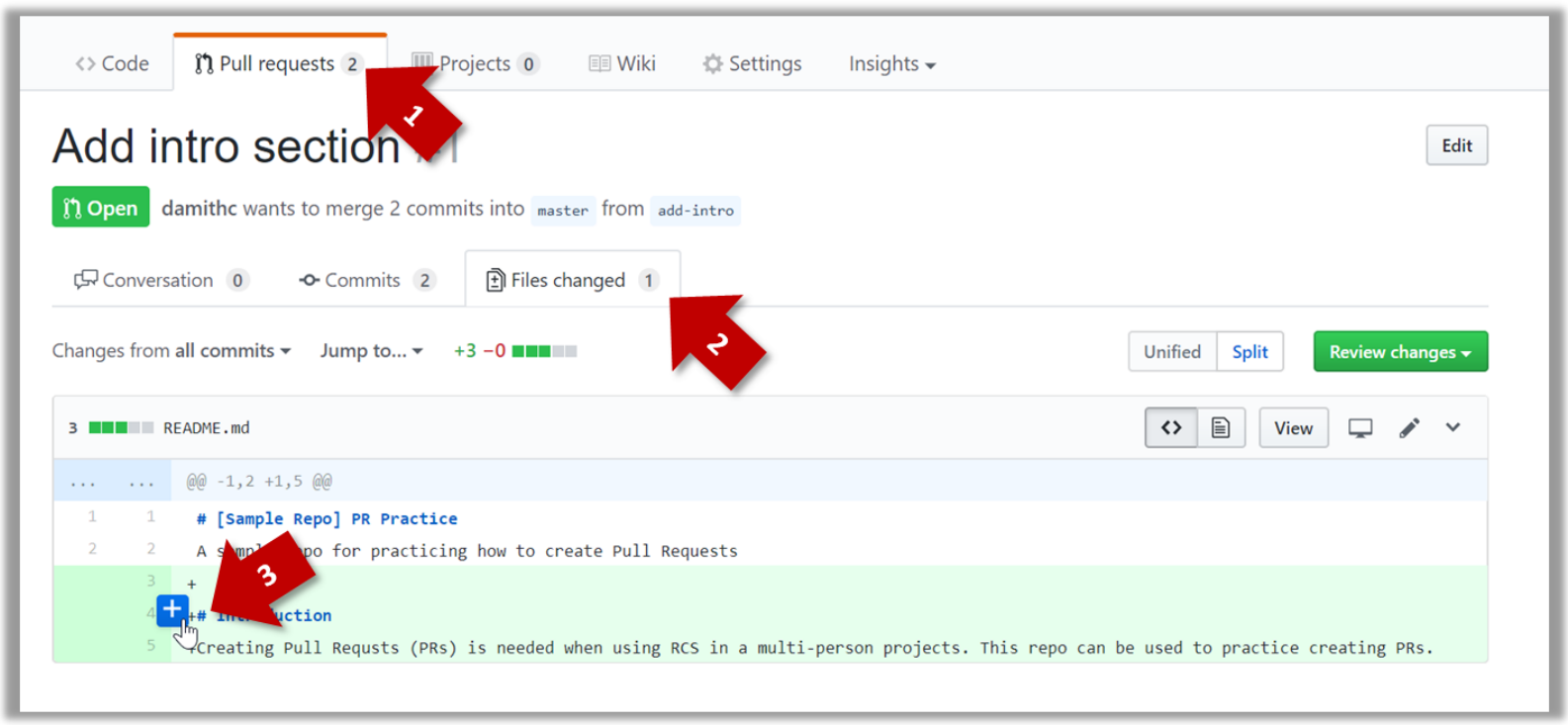
Tools → Git and GitHub → Manage PRs

- Tools → Git & GitHub → Branching
- Tools → Git & GitHub → Create PRs
- Tools → Git & GitHub → Merge Conflicts

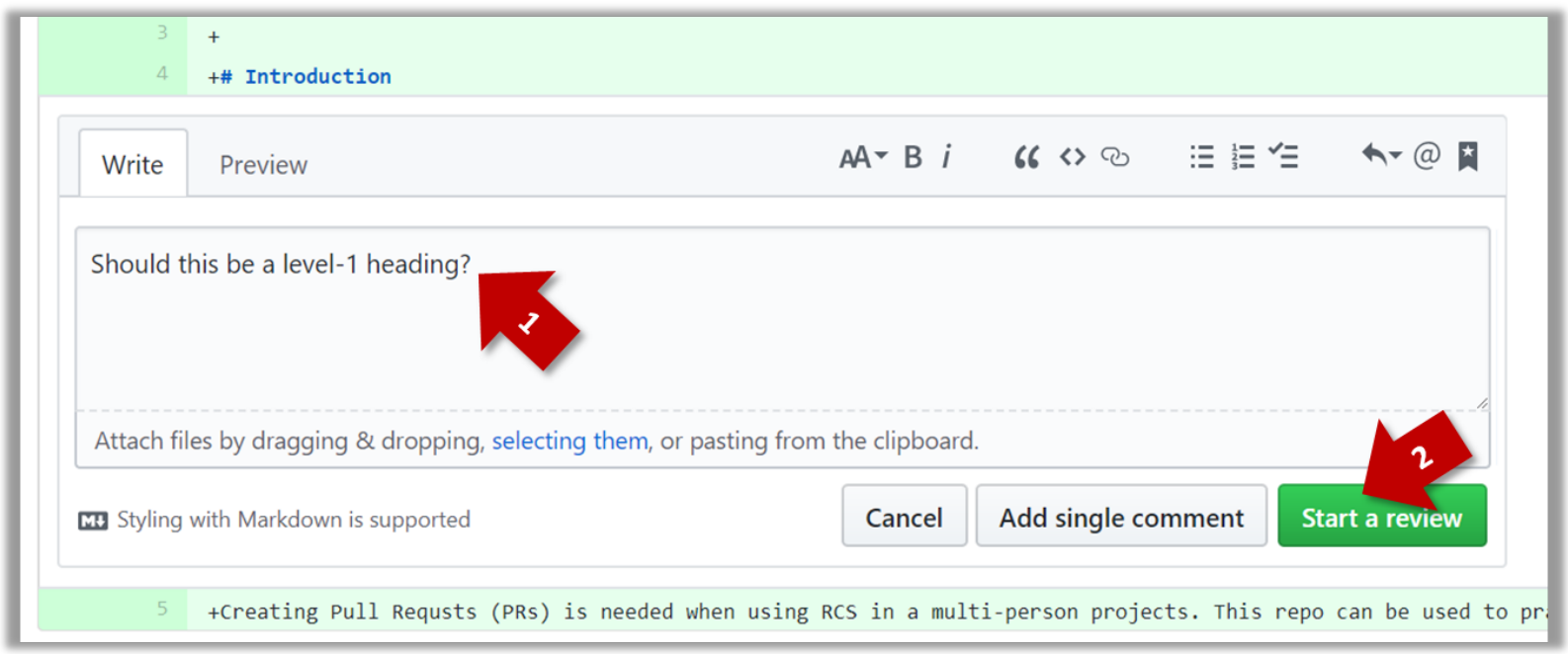
Can review and merge PRs on GitHub

1. Go to the GitHub page of your fork and review the `add-intro` PR you created previously in [Tools → Git & GitHub → Create PRs] to simulate the PR being reviewed by another developer, as explained below. Note that some features available to PR reviewers will be unavailable to you because you are also the author of the PR.

1a. Go to the respective PR page and click on the `Files changed` tab. Hover over the line you want to comment on and click on the ➕ icon that appears on the left margin. That should create a text box for you to enter your comment.

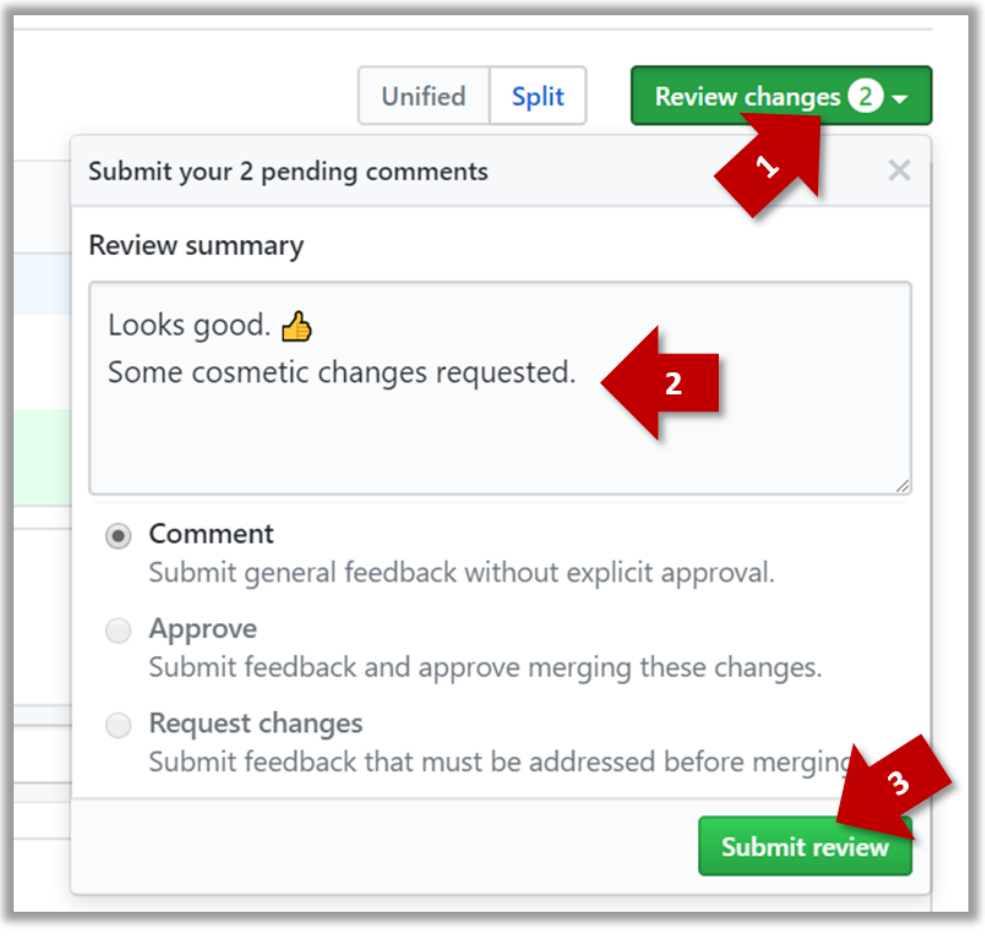


1b. Enter some dummy comment and click on `Start a review` button.



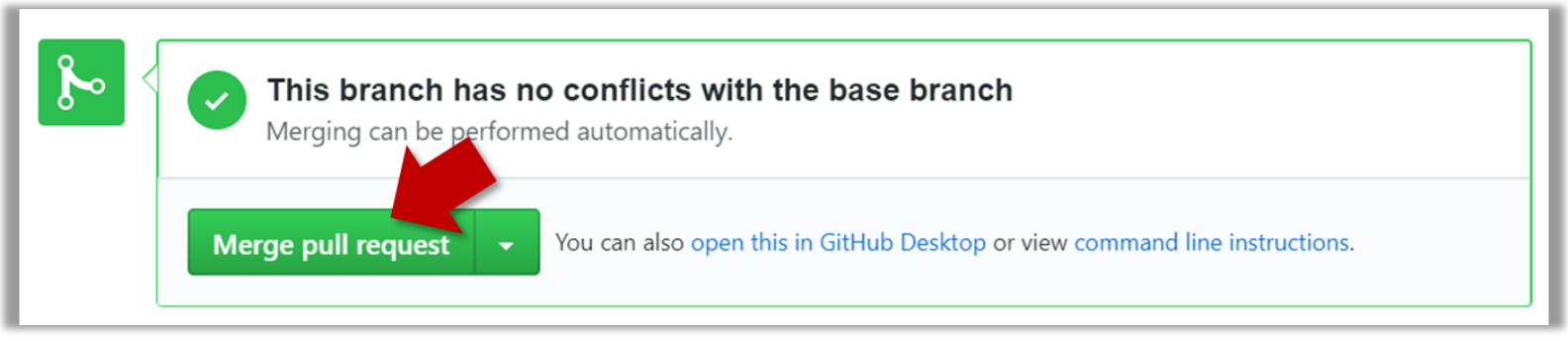
1c. Add a few more comments in other places of the code.

1d. Click on the `Review Changes` button, enter an overall comment, and click on the `Submit review` button.



2. Update the PR to simulate revising the code based on reviewer comments. Add some more commits to the `add-intro` branch and push the new commits to the fork. Observe how the PR is updated automatically to reflect the new code.

3. Merge the PR. Go to the GitHub page of the respective PR, scroll to the bottom of the `Conversation` tab, and click on the `Merge pull request` button, followed by the `Confirm merge` button. You should see a `Pull request successfully merged and closed` message after the PR is merged.



4. Sync the local repo with the remote repo. Because of the merge you did on the GitHub, the `master` branch of your fork is now ahead of your local repo by one commit. To sync the local repo with the remote repo, pull the `master` branch to the local repo.

```
1 git checkout master
2 git pull origin master
```

Observe how the `add-intro` branch is now merged to the `master` branch in your local repo as well.

5. De-conflict the `add-summary` PR that you created earlier. Note that GitHub page for the `add-summary` PR is now showing a conflict (when you scroll to the bottom of that page, you should see a message This

[W4.7] Code reviews ★★★

W4.7a ★★★

Quality Assurance → Quality Assurance → Code Reviews → What

Can explain code reviews

Code review is the systematic examination 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 a 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 reviews over testing:

- It can detect functionality defects as well as other problems such as coding standard violations.
- Can verify non-code artifacts and incomplete code
- Do not require test drivers or stubs.

Disadvantages:

- It is a manual process and therefore, error prone.
- **10 tips for reviewing code you don’t like** - a blog post by David Lloyd (a Red Hat developer).

[W4.8] Static Analysis ★★★

W4.8a ★★★ Quality Assurance → Quality Assurance → Static Analysis → What

Can explain static analysis

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

Static analysis of code can find useful information such 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.

Higher-end static analyzer tools can perform more complex analysis such as locating potential bugs, memory leaks, inefficient code structures etc.

Some example static analyzer 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'.

This site was built with MarkBind 2.14.1 at Tue, 21 Apr 2020, 6:01:40 UTC

This site is not ready yet! The updated version will be available soon.

Week 5 [Feb 10] - Topics

Detailed Table of Contents

[W5.1] Requirements: Intro ★★

W5.1a ★★ Requirements → Requirements → Introduction

Can explain requirements

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.

W5.1b ★★ Requirements → Requirements → Non-Functional Requirements

Can explain 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 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 system would operate or need 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.

We 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.

Given below are some requirements of TEAMMATES (an online peer evaluation system for education). Which one of these are non-functional requirements?

- [ ] a. The response to any use action should become visible within 5 seconds.
- [ ] b. The application admin should be able to view a log of user activities.
- [ ] c. The source code should be open source.
- [ ] d. A course should be able to have up to 2000 students.
- [ ] e. As a student user, I can view details of my team members so that I can know who they are.
- [ ] f. The user interface should be intuitive enough for users who are not IT-savvy.
- [ ] g. The product is offered as a free online service.

(a)(c)(d)(f)(g)

Explanation: (b) are (e) are functions available for a specific user types. Therefore, they are functional requirements. (a), (c), (d), (f) and (g) are either constraints on functionality or constraints on how the project is done, both of which are considered non-functional requirements.

W5.1c ★★★

Requirements → Requirements → Quality of Requirements

Can explain quality of requirements

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

- Unambiguous
- Testable (verifiable)
- Clear (concise, terse, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)

- Independent
- Atomic
- Necessary
- Implementation-free (i.e. abstract)

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

- Consistent
- Non-redundant
- Complete

W5.1d ★★★

Requirements → Requirements → Prioritizing Requirements

Can explain prioritizing requirements

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

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

📦 An example scheme for categorizing requirements:

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

📦 Other schemes:

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

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

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

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

[W5.2] Requirements: Gathering ★★★

W5.2a ★★★

Requirements → Gathering Requirements → Brainstorming

Can explain 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.

What is the key characteristic about brainstorming?

- a. There should be at least 5 participants.
- b. All ideas are welcome. There are no *bad* ideas.
- c. Only the best people in the team should take part.
- d. They are a good way to eliminate *bad* ideas.

(b)

W5.2b ★★★

Requirements → Gathering Requirements → Product Surveys

Can explain 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.

W5.2c ★★★

Requirements → Gathering Requirements → Observation

Can explain 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.

W5.2d ★★★

Requirements → Gathering Requirements → User Surveys

Can explain 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.

W5.2e ★★★

Requirements → Gathering Requirements → Interviews

Can explain interviews

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

W5.2f ★★★

Requirements → Gathering Requirements → Focus Groups

Can explain focus groups

[source]

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.

: How do focus groups work? - Hector Lanz + extra

W5.2g ★★★

Requirements → Gathering Requirements → Prototyping

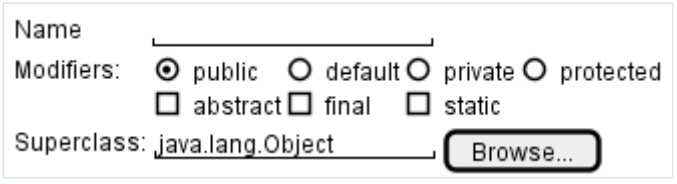
Can explain prototyping

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

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

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

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



[source: plantuml.com]

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

[W5.3] Requirements: Specifying

Prose

W5.3a ★★★

Requirements → Specifying Requirements → Prose → What

Can explain prose

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

W5.3b ★★

Requirements → Specifying Requirements → Feature Lists → What

Can explain feature list

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

W5.3c ★

Requirements → Specifying Requirements → User Stories → Introduction

Can write simple user stories

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}`

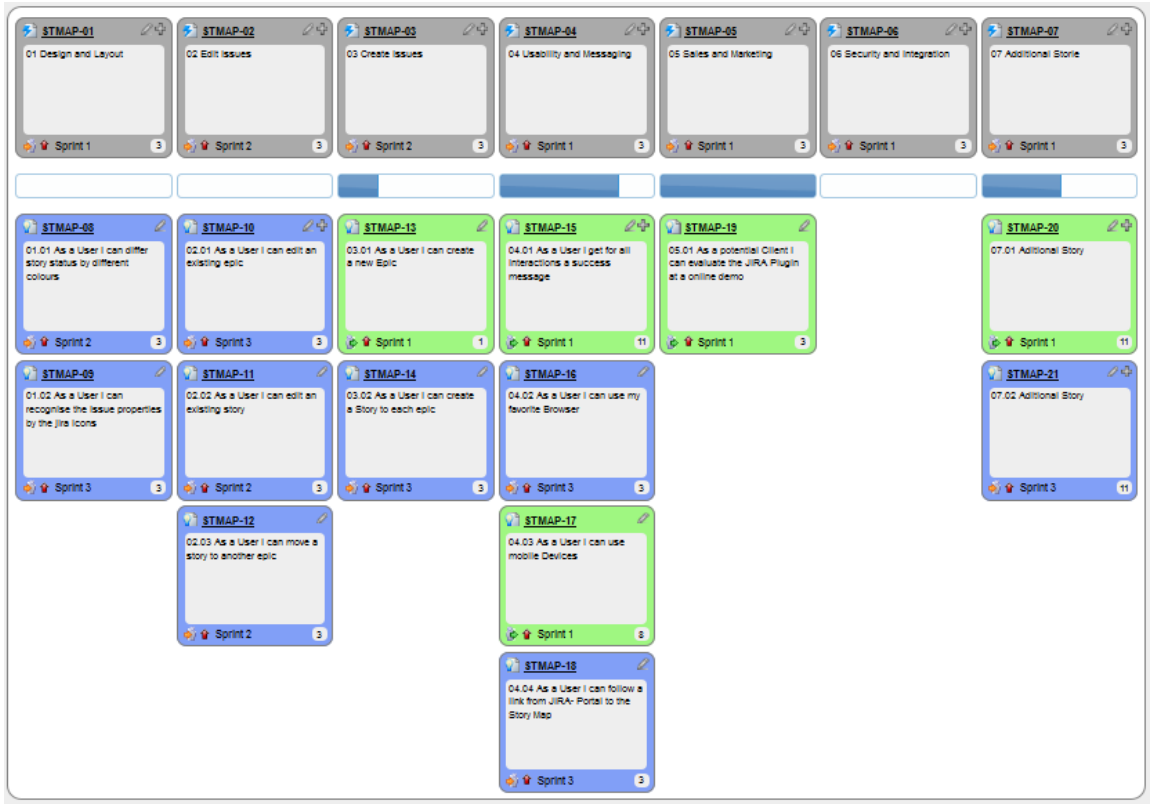
Examples (from a *Learning Management System*):

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`

We can write user stories on index cards or sticky notes, and arrange on walls or tables, to facilitate planning and discussion. Alternatively, we can use a software (e.g., GitHub Project Boards, Trello, Google Docs, ...) to manage user stories digitally.

[credit: https://www.flickr.com/photos/jakuza/2682466984/]

[credit: https://www.flickr.com/photos/jakuza/with/2726048607/]



[credit: https://commons.wikimedia.org/wiki/File:User_Story_Map_in_Action.png]

- [ ] a. They are based on stories users tell about similar systems
- [ ] b. They are written from the user/customer perspective
- [ ] c. They are always written in some physical medium such as index cards or sticky notes

- [ ] a. Reason: Despite the name, user stories are not related to 'stories' about the software.
- [x] b.
- [ ] c. Reason: It is possible to use software to record user stories. When the team members are not co-located this may be the only option.

Critique the following user story taken from a software project to build an e-commerce website.

```
As a developer, I want to use Python to implement the software, so that we can resue existing Python modules.
```

Refer to the definition of a user story.

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]

This user story is not written from the perspective of the user/customer.

write your answer here...

Bill wants you to build a Human Resource Management (HRM) system. He mentions that *the system will help employees to view their own leave balance*. What are the user stories you can extract from that statement?

Remember to follow the correct format when writing user stories.

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

`As an employee, I can view my leave balance, so that I can know how many leave days I have left.`

Note: the `{benefit}` part may vary as it is not specifically mentioned in the question.

write your answer here...

- [ ] a. They are based on stories users tell about similar systems
- [ ] b. They are written from the user/customer perspective
- [ ] c. They are always written in some physical medium such as index cards or sticky notes

- [ ] a. Reason: Despite the name, user stories are not related to 'stories' about the software.
- [x] b.
- [ ] c. Reason: It is possible to use software to record user stories. When the team members are not co-located this may be the only option.

Critique the following user story taken from a software project to build an e-commerce website.

`As a developer, I want to use Python to implement the software, so that we can resue existing Python modules.`

Refer to the definition of a user story.

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]

This user story is not written from the perspective of the user/customer.

write your answer here...

Bill wants you to build a Human Resource Management (HRM) system. He mentions that *the system will help employees to view their own leave balance*. What are the user stories you can extract from that statement?

Remember to follow the correct format when writing user stories.

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

`As an employee, I can view my leave balance, so that I can know how many leave days I have left.`

Note: the `{benefit}` part may vary as it is not specifically mentioned in the question.

write your answer here...

W5.3d ★★

Requirements → Specifying Requirements → User Stories → Details

Can write more detailed user stories

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 do 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

More examples + extra

Choose the correct statements

- ☐ a. User stories are short and written in a formal notation.
- ☐ b. User stories is another name for use cases.
- ☐ c. User stories describes past experiences users had with similar systems. These are helpful in developing the new system.
- ☐ d. User stories are not detailed enough to tell us exact details of the product.

d

Explanation: User stories are short and written in natural language, NOT in a formal language. They are used for estimation and scheduling purposes but do not contain enough details to form a complete system specification.

W5.3e ★★

Requirements → Specifying Requirements → User Stories → Usage

Can use user stories to manage requirements of project

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 a 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**.

Here are some tips for using user stories for early stages of requirement gathering:

- **Define the *target user*:**
  Decide your target user's profile (e.g. a student, office worker, programmer, sales person) 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 give this user a name. e.g. Target user 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. ...
- **Define the *problem scope*:** Decide that exact problem you are going to solve for the target user. e.g. Help Jean keep track of all her school contacts
- **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 details:**
  For example, consider this user story: `As a user, I want to see a list of tasks that needs 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.

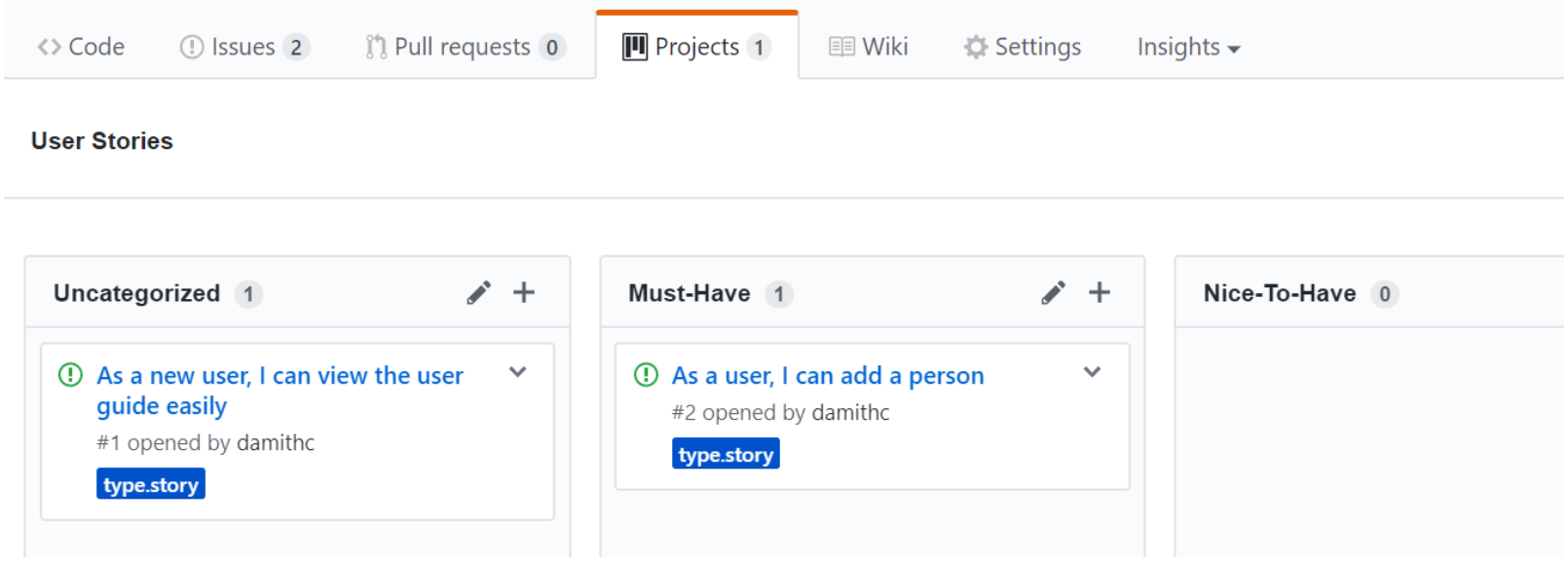
- **Don't discuss implementation details or whether you are actually going to implement it:**
  When gathering requirements, your decision is whether the user's need is important enough for you to want to fulfil it. Implementation details can be discussed later. If a user story turns out to be too difficult 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.

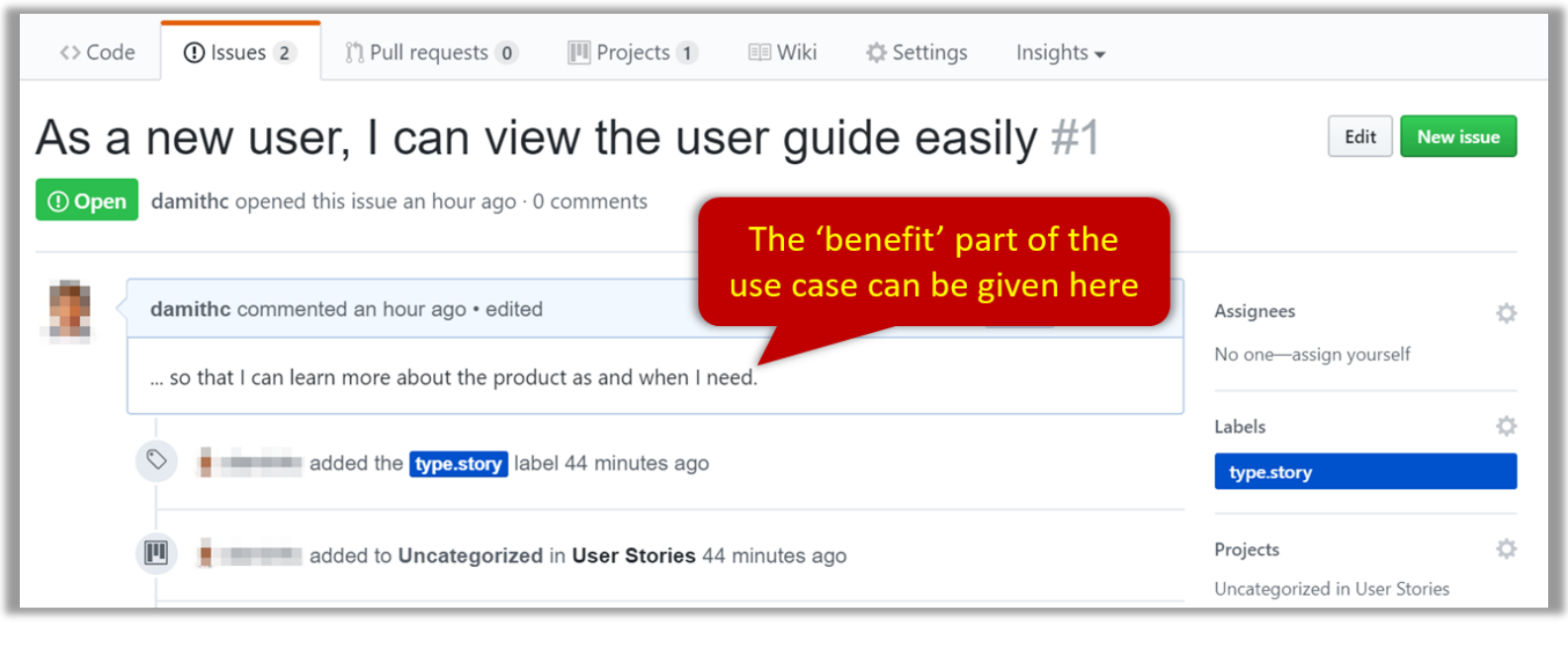
GitHub Project Boards | Google Sheets | Trello

You can create issues for each of the user stories and use a GitHub *Project Board* to sort them into categories.

📦 Example Project Board:



📦 Example Issue to represent a user story:



A video on GitHub Project Boards:

- This article by Mike Cohn from MountainGoatSoftware explains how to use user stories to capture NFRs.

Glossary

W5.3f ★★★

Requirements → Specifying Requirements → Glossary → What

Can explain glossary

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

W5.3g ★★★

Requirements → Specifying Requirements → Supplementary Requirements → What

Can explain supplementary requirements

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

[W5.4] Code Quality ★

Readability

W5.4a ★

Implementation → Code Quality → Readability → Introduction

Can explain the importance of readability

> 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 we do not intend to pass the code to someone else, code quality is still important because we all become 'strangers' to our 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 (isAlreadySubsidised) {
        subsidy = SUBSIDISED_SUBSIDY;
    } else if (isPartTime) {
         subsidy = FULLTIME_SUBSIDY * RATIO;
    } else {
        subsidy = FULLTIME_SUBSIDY;
    }
    return subsidy;
}
```

W5.4b ★★

Implementation → Code Quality → Readability → Basic → Avoid Long Methods

Can improve code quality using technique: 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.

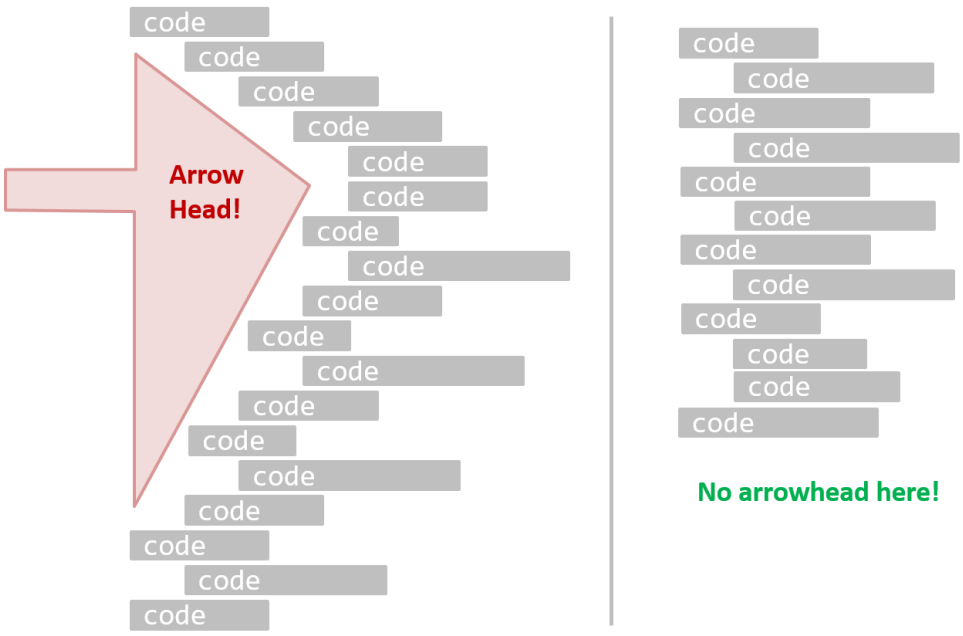
W5.4c ★★

Implementation → Code Quality → Readability → Basic → Avoid Deep Nesting

Can improve code quality using technique: 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 CodingStyle

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;
    }
```

Good

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

W5.4d ★★

Implementation → Code Quality → Readability → Basic → Avoid Complicated Expressions

Can improve code quality using technique: 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

W5.4e ★★

Implementation → Code Quality → Readability → Basic → Avoid Magic Numbers

Can improve code quality using technique: avoid magic numbers

When the code has a number that does not explain the meaning of the number, we call that 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, we can have ‘magic’ values of other data types.

Bad

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

In general, try to avoid any magic literals.

W5.4f ★★

Implementation → Code Quality → Readability → Basic → Make the Code Obvious

Can improve code quality using technique: 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 grouping 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'` .

W5.4g ★★★

Implementation → Code Quality → Readability → Intermediate → Structure Code Logically

🏆 **Can improve code quality using technique: structure code logically**

Lay out the code so that it adheres to the logical structure. The code should read like a story. Just like we 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.

W5.4h ★★★

Implementation → Code Quality → Readability → Intermediate → Do Not 'Trip Up' Reader

🏆 **Can improve code quality using technique: do not 'trip up' reader**

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

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

W5.4i ★★★

Implementation → Code Quality → Readability → Intermediate → Practice KISSing

🏆 **Can improve code quality using technique: practice kissing**

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

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

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

W5.4j ★★★

Implementation → Code Quality → Readability → Intermediate → Avoid Premature Optimizations

Can improve code quality using technique: avoid premature optimizations

Optimizing code prematurely has several drawbacks:

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

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

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

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

W5.4k ★★★

Implementation → Code Quality → Readability → Intermediate → SLAP Hard

Can improve code quality using technique: SLAP hard

Avoid varying the level of abstraction within a code fragment. Note: The *Productive Programmer* (by Neal Ford) calls this the *SLAP principle* i.e. Single Level of Abstraction Per method.

Example:

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

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

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

```
1 readData();
2 processData();
3 displayResult();
```

W5.4l ★★★

Implementation → Code Quality → Readability → Advanced → Make the Happy Path Prominent

Can improve code quality using technique: make the happy path prominent

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

Example:

👎 Bad

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

In the code above,

- Unusual condition detection is separated from their handling.
- Main path is nested deeply.

👍 Good

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

if (isErrorCase) { //Guard Clause
    handleError();
    return;
}

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

In contrast, the above code

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

Naming

W5.4m ★★

Implementation → Code Quality → Naming → Introduction

🏆 **Can explain the need for good names in code**

Proper naming improves the readability. 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

W5.4n ★★

Implementation → Code Quality → Naming → Basic → Use Nouns for Things and Verbs for Actions

Can improve code quality using technique: 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	`CheckLimit`	`LimitChecker`
method	`result()`	`calculate()`

Distinguish clearly between single-valued and multivalued variables.

Examples:

Good

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

W5.4o ★★★

Implementation → Code Quality → Naming → Basic → Use Standard Words

Can improve code quality using technique: 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

W5.4p ★★★

Implementation → Code Quality → Naming → Intermediate → Use Name to Explain

Can improve code quality using technique: use name to explain

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

Examples:

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

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

Examples:

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

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

Examples:

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

W5.4q ★★★

Implementation → Code Quality → Naming → Intermediate → Not Too Long, Not Too Short

🏆 Can improve code quality using technique: not too long, not too short

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

W5.4r ★★★

Implementation → Code Quality → Naming → Intermediate → Avoid Misleading Names

🏆 Can improve code quality using technique: avoid misleading names

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

Example: Consider these variables

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

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

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

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

Examples:

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

Unsafe Practices

W5.4s ★★

Implementation → Code Quality → Error-Prone Practices → Introduction

🏆 Can explain the need for avoiding error-prone shortcuts

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

W5.4t ★★

Implementation → Code Quality → Error-Prone Practices → Basic → Use the Default Branch

🏆 Can improve code quality using technique: use the default branch

Always include a default branch in `case` statements.

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

Bad

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

Good

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

W5.4u ★★

Implementation → Code Quality → Error-Prone Practices → Basic → Don't Recycle Variables or Parameters

Can improve code quality using technique: don't recycle variables or parameters

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

Bad

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

Good

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

W5.4v ★★

Implementation → Code Quality → Error-Prone Practices → Basic → Avoid Empty Catch Blocks

Can improve code quality using technique: avoid empty catch blocks

Never write an empty `catch` statement. At least give a comment to explain why the `catch` block is left empty.

W5.4w ★★

Implementation → Code Quality → Error-Prone Practices → Basic → Delete Dead Code

Can improve code quality using technique: delete dead code

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

W5.4x ★★★

Implementation → Code Quality → Error-Prone Practices → Intermediate → Minimize Scope of Variables

Can improve code quality using technique: minimise scope of variables

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

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

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

📎 Resources:

- Refactoring: Reduce Scope of Variable

W5.4y ★★★

Implementation → Code Quality → Error-Prone Practices → Intermediate → Minimize Code Duplication

Can improve code quality using technique: minimize code duplication

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

This guideline is closely related to the DRY Principle.

Code Comments

W5.4z ★★

Implementation → Code Quality → Comments → Introduction

Can explain the need for commenting minimally but sufficiently

> 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'. This is not so. Avoid writing comments to explain bad code. Improve the code to make it self-explanatory.

W5.4A ★★

Implementation → Code Quality → Comments → Basic → Do Not Repeat the Obvious

Can improve code quality using technique: 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

```
1 //increment x
2 x++;
3
4 //trim the input
5 trimInput();
```

W5.4B ★★

Implementation → Code Quality → Comments → Basic → Write to the Reader

Can improve code quality using technique: 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 comments 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

```
1 // a quick trim function used to fix bug I detected overnight
2 void trimInput(){
3     ....
4 }
```

Good

```
1 /** Trims the input of leading and trailing spaces */
2 void trimInput(){
3     ....
4 }
```

W5.4C ★★★

Implementation → Code Quality → Comments → Intermediate → Explain WHAT and WHY, not HOW

Can improve code quality using technique: explain what and why, not how

Comments should explain *what* and *why* aspect of the code, rather than the *how* aspect.

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

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

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

✔ **Why** : The rationale for the current implementation.

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

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

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

Example:

Bad Reason: Comment explains how the code works.

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

Good Reason: Code refactored to be self-explanatory. Comment no longer needed.

```

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

[W5.5] Refactoring ★★

W5.5a ★★

Implementation → Refactoring → What

Can explain refactoring

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 built in 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.

Choose the correct statements

- [ ] a. Refactoring can improve understandability
- [ ] b. Refactoring can uncover bugs
- [ ] c. Refactoring can result in better performance
- [ ] d. Refactoring can change the number of methods/classes

a, b, c, d

Explanation:

- (a, b, c) Although the primary aim of refactoring is to improve internal code structure, there are other secondary benefits.
- (d) Some refactorings result in adding/removing methods/classes.

Do you agree with the following statement? Justify your answer.

Statement: `Whenever we refactor code to fix bugs, we need not do regression testing if the bug fix was minor.`

There are two flaws in the given statement.

DISAGREE.

1. Even a minor change can have major repercussions on the system. We MUST do regression testing after each change, no matter how minor it is.
2. Fixing bugs is technically not refactoring.

write your answer here...

Explain what is refactoring and why it is not the same as rewriting, bug fixing, or adding features.

write your answer here...

W5.5b ★★

Tools → Intellij IDEA → Refactoring

Implementation → Refactoring → What

Can use automated refactoring features of the IDE

This video explains how to automate the 'Extract parameter' refactoring using Intellij IDEA. Most other refactorings available works similarly. i.e. `select the code to refactor` → `find the refactoring in the context menu` or `use the keyboard shortcut` .

Here's another video explaining how to change a method signature as part of refactoring.

- Introduction to Refactoring (in Intellij IDEA) : An article on refactorings available in Intellij IDEA.

W5.5c ★★★

Implementation → Refactoring → How

Can apply some basic refactoring

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

W5.5d ★★★★: OPTIONAL

Implementation → Refactoring → When

[W5.6] Assertions

W5.6a : ★★

Implementation → Error Handling → Assertions → What

Can explain assertions

***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 take 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 we set an assertion that `timeout` returned by `Config.getTimeout()` is greater than `0`. Now, if the `Config.getTimeout()` returned `-1` in a specific execution of this line, the runtime can detect it as a *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.

```
1 int timeout = Config.getTimeout();
```

W5.6b : ★★★

Implementation → Error Handling → Assertions → How

Can use assertions

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.

```
1 x = getX();
2 assert x == 0 : "x should be 0";
3 ...
```

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.

Tutorials:

- **Java Assertions** -- a simple tutorial from *javatpoint.com*
- **Programming with Assertions** (first half) -- a more detailed tutorial from *Oracle*

Best practices:

- **Programming with Assertions** (second half) -- *from Oracle* (also listed above as a tutorial) contains some best practices towards the end of the article.

W5.6c

Implementation → Error Handling → Assertions → When

Can use assertions optimally

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.

```
1 ...
2 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).

A Calculator program crashes with an ‘assertion failure’ message when you try to find the square root of a negative number.

- a. This is a correct use of assertions.
- b. The application should have terminated with an exception instead.
- c. The program has a bug.
- d. All statements above are incorrect.

(c)

Explanation: An assertion failure indicates a bug in the code. (b) is not acceptable because of the word "terminated". The application should not fail at all for this input. But it could have used an exception to handle the situation internally.

Which statements are correct?

- a. Use assertions to indicate the programmer messed up; Use exceptions to indicate the user or the environment messed up.
- b. Use exceptions to indicate the programmer messed up; Use assertions to indicate the user or the environment messed up.

(a)

[W5.7] Java: streams

W5.7a : ★★★

C++ to Java → Miscellaneous Topics → Streams: Basic

Can use Java8 streams

Java 8 introduced a number of new features (e.g. Lambdas, Streams) that are not trivial to learn but also extremely useful to know.

Here is an overview of new Java 8 features .(written by Benjamin Winterberg)

Tutorials:

- **Java 8 Tutorial** -- from tutorialspoint.com. Also provides a way to try out code online
- **Tutorials from Oracle**: [Lambdas][Streams]

[W5.8] Continuous Integration/Deployment ★

W5.8a ★

Implementation → Integration → Introduction → What

Can explain integration

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.

W5.8b ★

Implementation → Integration → Build Automation → What

Can explain build automation tools

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 source code can be a complex multi-step process. For example, it can include steps such as to 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 to compile, link and package 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 to 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.

- **Getting Started with Gradle** -- Documentation from Gradle
- **Gradle Tutorial** -- from tutorialspoint.com

Working With Gradle in Intellij IDEA (6 minutes)

Gradle_is used used for,

- [ ] a. better revision control
- [ ] b. build automation
- [ ] c. UML diagramming
- [ ] d. project collaboration

(b)

W5.8c ★★

Implementation → Integration → Build Automation → Continuous Integration and Continuous Deployment

Can explain continuous integration and continuous deployment

An extreme application of build automation is called *continuous integration (CI)* in which integration, building, and testing happens automatically after each code change.

A natural extension of CI is *Continuous Deployment (CD)* where the changes are not only integrated continuously, but also deployed to end-users at the same time.

Some examples of CI/CD tools: Travis, Jenkins, Appveyor, CircleCI, GitHub Actions

- **Travis CI for Beginners** -- Documentation from Travis

This site is not ready yet! The updated version will be available soon.

Week 6 [Feb 17] - Topics

Detailed Table of Contents

[W6.1] Modeling: Sequence Diagrams ★★

W6.1a ★★ Design → Modelling → Modelling Behaviors Sequence Diagrams - Basic

Can draw basic sequence diagrams

UML Sequence Diagrams → Introduction

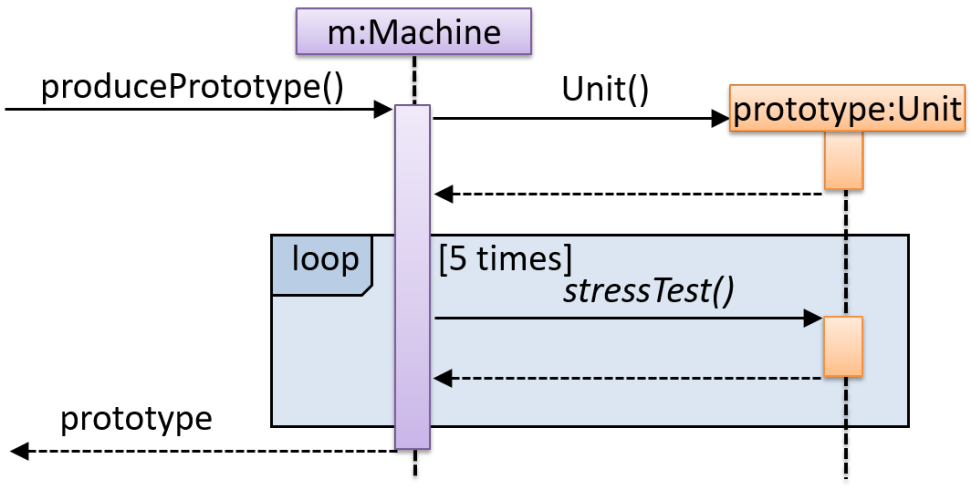
UML Sequence Diagrams → Basic Notation

UML Sequence Diagrams → Loops

UML Sequence Diagrams → Object Creation

UML Sequence Diagrams → Minimal Notation

Explain in your own words the interactions illustrated by this Sequence Diagram:



Consider the code below:

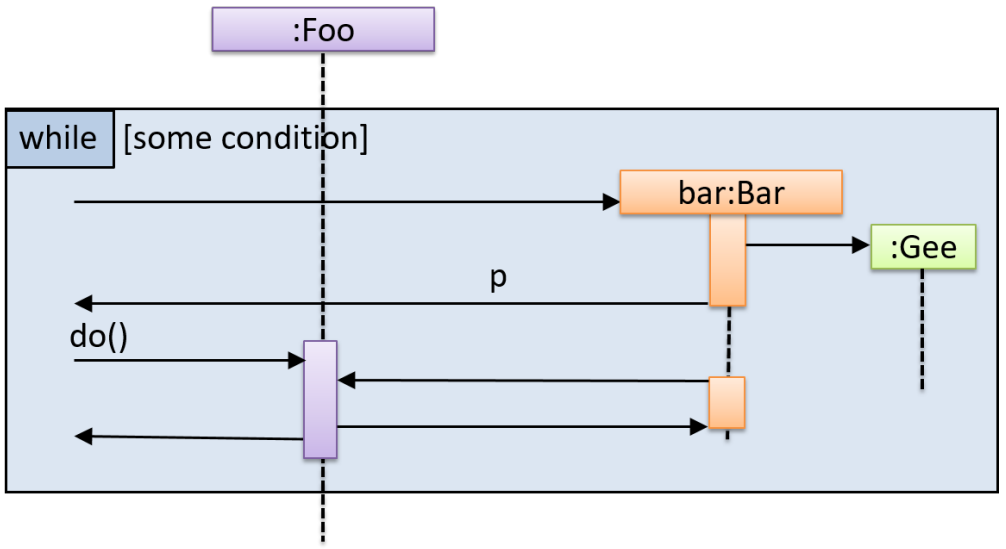
```
1  class Person{
2      Tag tag;
3      String name;
4
5           Person(String  personName,  String
   tagName){
6          name = personName;
7          tag = new Tag(tagName);
8      }
9  }
```

```
1  class Tag{
2      Tag(String value){
3          //...
4      }
5  }
6
7  class PersonList{
8        void  addPerson(Person
   p){
9          //...
10     }
11 }
```

Draw a sequence diagram to illustrate the object interactions that happen in the code snippet below:

```
1 PersonList personList = new PersonList();
2 while (hasRoom){
3     Person p = new Person("Adam", "friend");
4     personList.addPerson(p);
5 }
```

Find notation mistakes in the sequence diagram below:



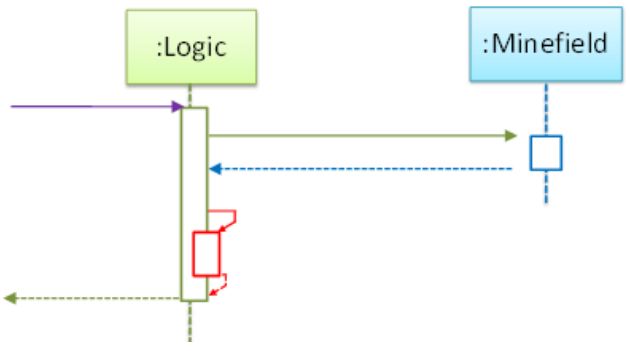
W6.1b ★★★

Design → Modelling → Modelling Behaviors Sequence Diagrams - Intermediate

Can draw intermediate-level sequence diagrams

- 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

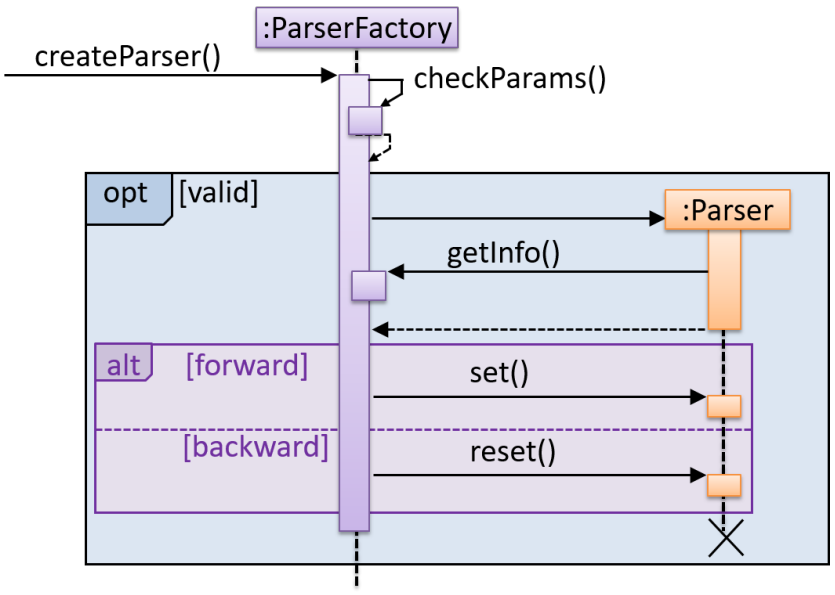
What's going on here?



- a. `Logic` object is executing a parallel thread.
- b. `Logic` object is executing a loop.
- c. `Logic` object is creating another `Logic` instance.
- d. One of `Logic` object's methods is calling another of its methods.
- e. `Minefield` object is calling a method of `Logic` .

(d)

Explain the interactions depicted in this sequence diagram.



First, the `createParser()` method of an existing `ParserFactory` object is called. Then, ...

Draw a sequence diagram to represent this code snippet.

```
if (isFirstPage) {
    new Quote().print();
}
```

The `Quote` class:

```
class Quote{

    String q;

    Quote(){
        q = generate();
    }

    String generate(){
        // ...
    }

    void print(){
        System.out.println(q);
    }

}
```

- Show `new Quote().print();` as two method calls.
- As the created Quote object is not assigned to a variable, it can be considered as 'deleted' soon after its `print()` method is called.

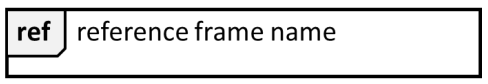
W6.1c ★★★

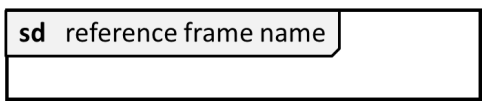
Tools → UML → Sequence Diagrams → Reference Frames

Can interpret sequence diagrams with reference frames

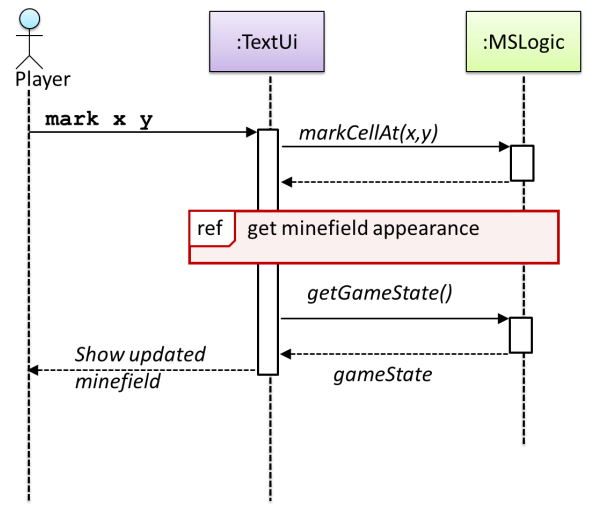
UML uses *ref frame* to allow a segment of the interaction to be omitted and shown as a separate sequence diagram. Reference frames help us to break complicated sequence diagrams into multiple parts or simply to omit details we are not interested in showing.

Notation:

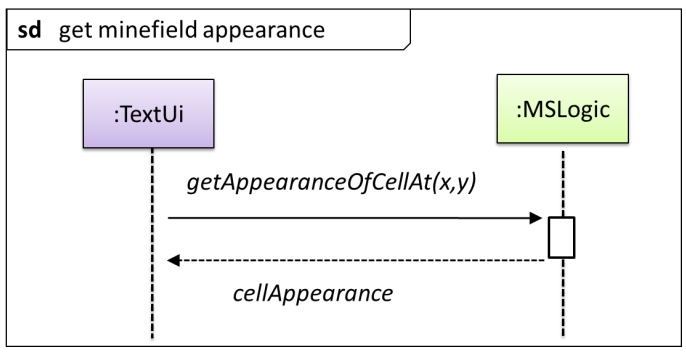




The details of the `get minefield appearance` interactions have been omitted from the diagram.



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



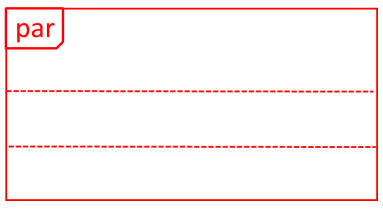
W6.1d ★★★

Tools → UML → Sequence Diagrams → Parallel Paths

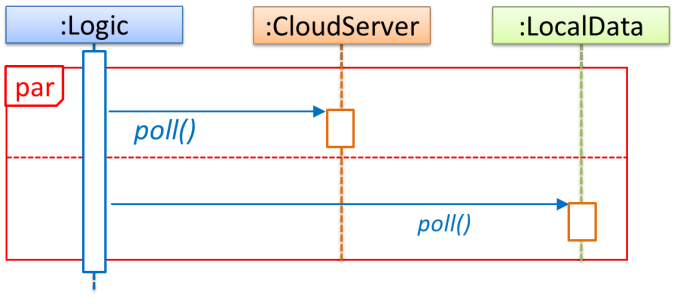
Can interpret sequence diagrams with parallel paths

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.

[W6.2] Architecture Diagrams ★

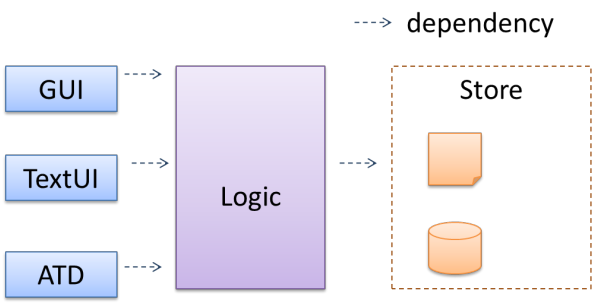
W6.2a ★
Design → Architecture → Introduction → What

Can explain Software Architecture

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

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

A possible architecture for a *Minesweeper* game



Main components:

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

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

Choose the correct statement

- [ ] a. The architecture of a system should be simple enough for all team members to understand it.
- [ ] b. The architecture is primarily a high-level design of the system.
- [ ] c. The architecture is usually decided by the project manager.
- [ ] d. The architecture can contain details private to a component.

(a)(b)

~~(c)~~ Reason: Architecture is usually designed by the Architect

~~(d)~~ Reason:

> ... private details of elements—details having to do solely with internal implementation—are not architectural.

W6.2b ★

Design → Architecture → Architecture Diagrams → Reading

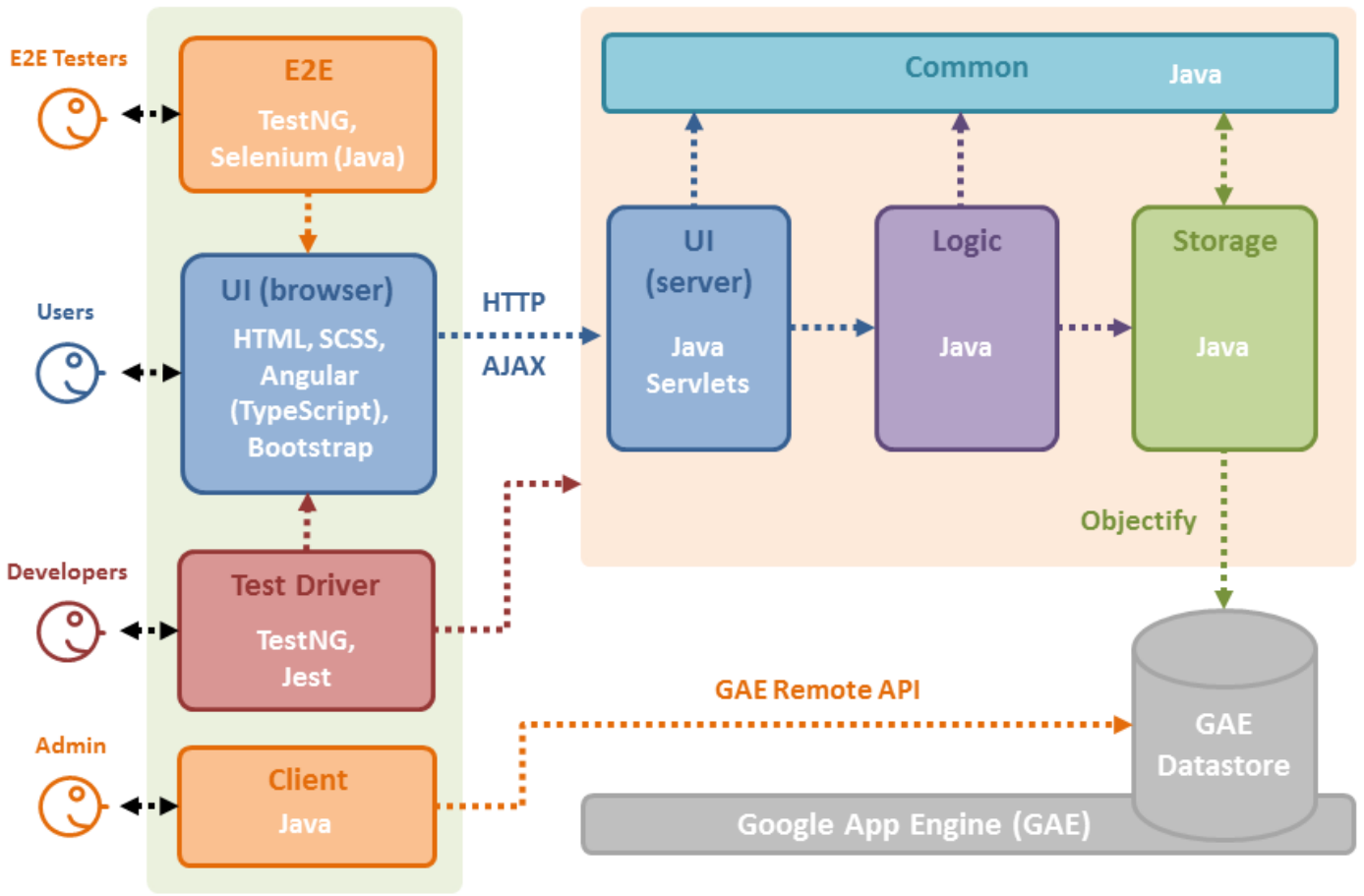
🎓 Design → Architecture → Introduction → What

🏆 **Can interpret an architecture diagram**

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

📦 Some example architecture diagrams:

TEAMMATES | se-edu/addressbook-level3 | Example 1 | Example 2 | Example 3



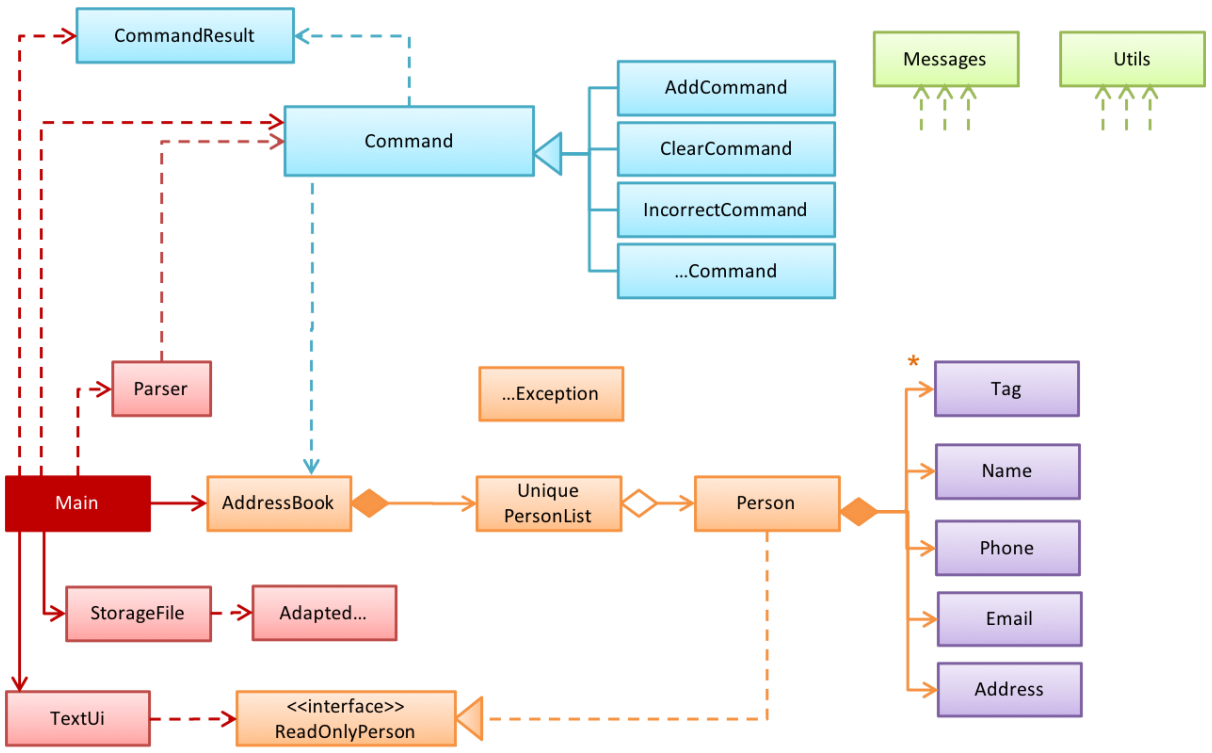
W6.2c ★★

Design → Introduction → Multi-Level Design

🏆 **Can explain multi-level design**

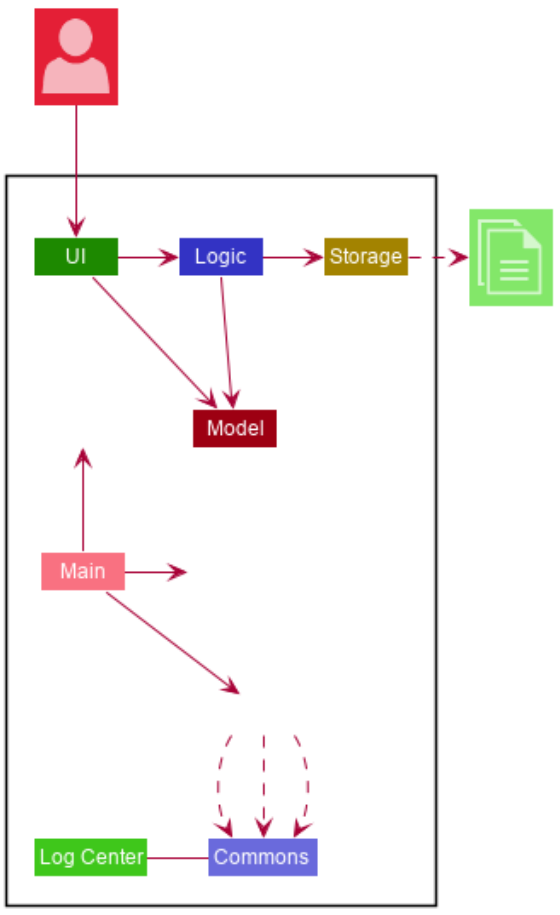
In a smaller system, design of the entire system can be shown in one place.

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



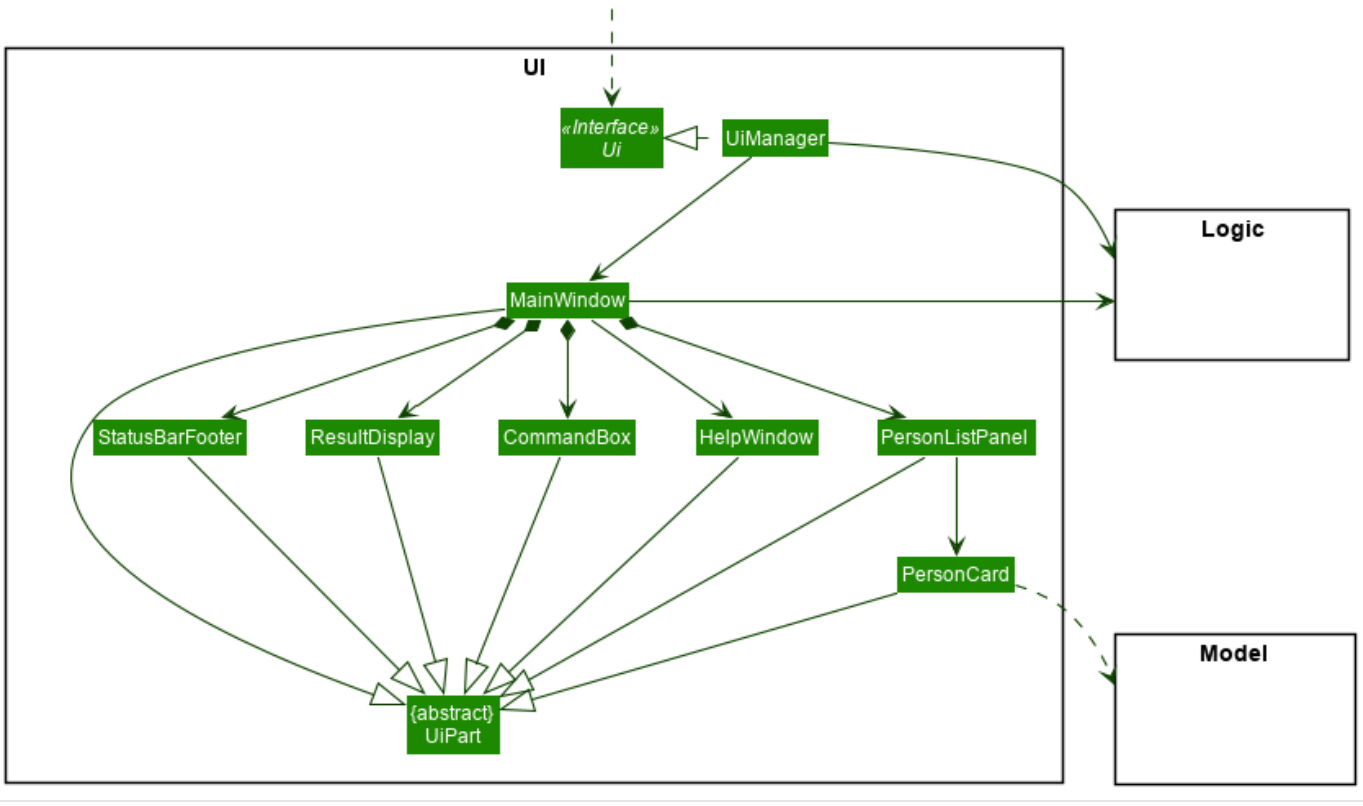
Design of bigger systems needs to be done/shown at multiple levels.

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



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

UI Logic Storage



[W6.3] **IDEs: Basic Features** ★★

W6.3a ★★★

Implementation → IDEs → Debugging → What

Can explain debugging

***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 increases the risk of introducing errors into the program.
  - These print statements, if not removed promptly after the debugging, may even appear unexpectedly in the production version.
- 👎 **Bad** -- **By manually tracing through the code**: Otherwise known as ‘eye-balling’, this approach doesn't have the cons of the previous approach, but it too is not recommended (other than as a 'quick try') due to these reasons:
  - It is difficult, time consuming, and error-prone technique.
  - If you didn't spot the error while writing code, you might not spot the error when reading code too.
- 👍 **Good** -- **Using a debugger**: A debugger tool allows you to pause the execution, then step through 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**.

W6.3b ★★★

Tools → Intellij IDEA → Debugging: Basic

Can step through a program using a debugger

This video (from LaunchCode) gives a pretty good explanation of how to use the Intellij IDEA debugger.

- Intellij IDEA Documentation: Debugging Basics - Can be used as a reference document when you want to recall how to use a debugging feature.

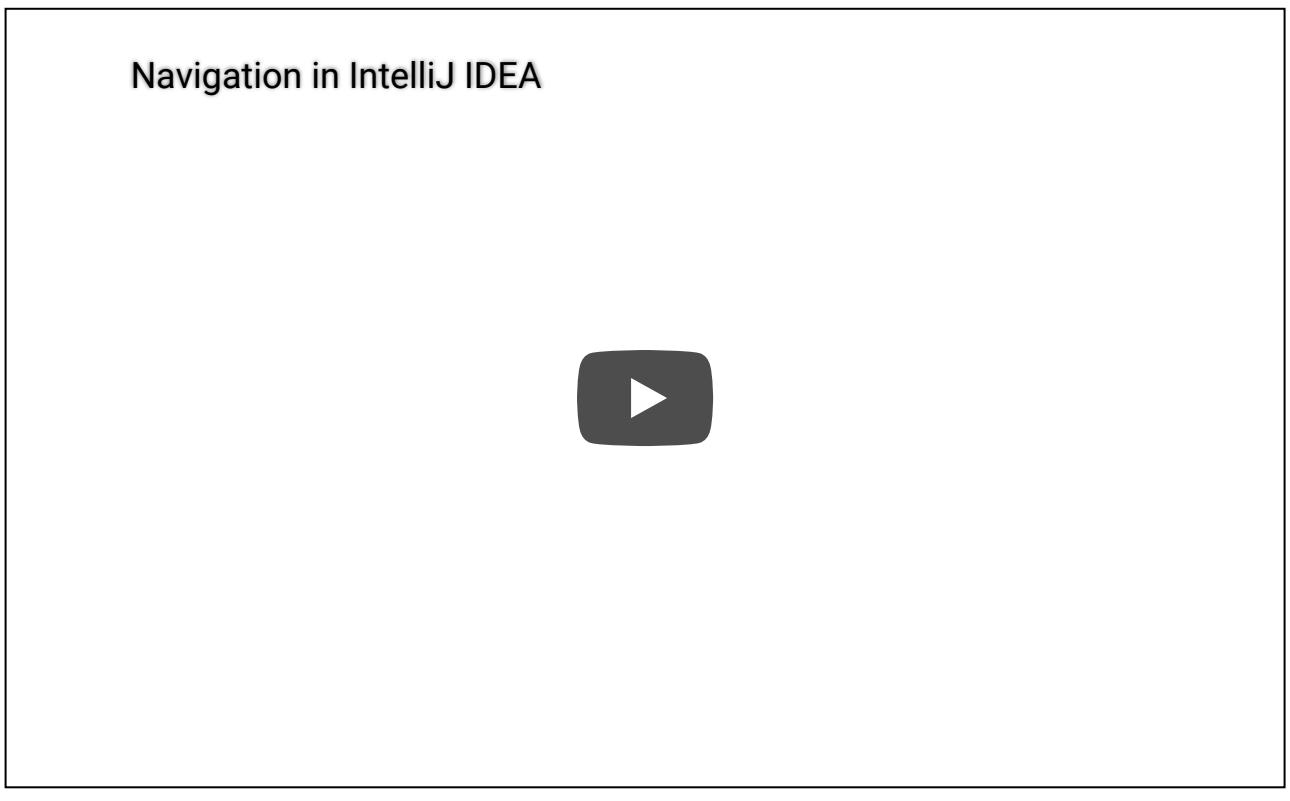
W6.3c ★★

Tools → Intellij IDEA → Code Navigation

Can navigate code effectively using IDE features

Some useful navigation shortcuts:

1. Quickly locate a file by name.
2. Go to the definition of a method from where it is used.
3. Go back to the previous location.
4. View the documentation of a method from where the method is being used, without navigating to the method itself.
5. Find where a method/field is being used.



[W6.4] Logging ★★

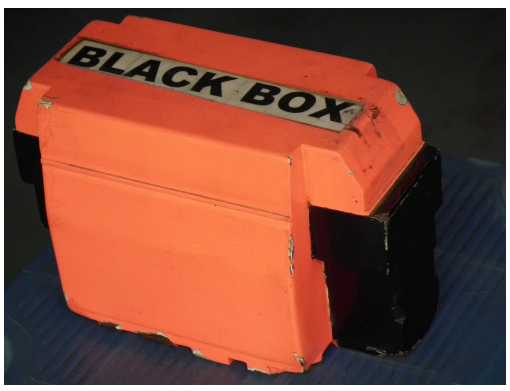
W6.4a ★★
Implementation → Error Handling → Logging → What

Can explain logging

***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 action 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.



source:https://commons.wikimedia.org

Why is logging like having the 'black box' in an airplane?

- a. Because they can be used to find more info about how the system behaved before/during an ‘incident’.
- b. Because they are hidden from the view of normal users.
- c. Because they both can be in hardcopy form or digital form.
- d. Because they are hard to open by normal users.

(a)

W6.4b ★★★
Implementation → Error Handling → Logging → How

Can use logging

Most programming environments come with logging systems that allow sophisticated forms of logging. They have features such as the ability to enable and disable logging easily or to change the logging intensity.

This sample Java code uses Java’s default logging mechanism.

First, import the relevant Java package:

```
1 import java.util.logging.*;
```

Next, create a `Logger` :

```
1 private static Logger logger = Logger.getLogger("Foo");
```

Now, you can use the `Logger` object to log information. Note the use of logging level for each message. When running the code, the logging level can be set to `WARNING` so that log messages specified as `INFO` level (which is a lower level than `WARNING`) will not be written to the log file at all.

```
1  // log a message at INFO level
2  logger.log(Level.INFO, "going to start processing");
3  //...
4  processInput();
5  if(error){
6      //log a message at WARNING level
7      logger.log(Level.WARNING, "processing error", ex);
8  }
9  //...
10 logger.log(Level.INFO, "end of processing");
```

Tutorials:

- **Java Logging API - Tutorial** -- A tutorial by *Lars Vogella*
- **Java Logging Tutorial** -- An alternative tutorial by *Jakob Jenkov*

- A video tutorial by *SimplyCoded*:

Best Practices:

- **10 Tips for Proper Application Logging** -- by *Tomasz Nurkiewicz*
- **What each logging level means** -- conventions recommended by *Apache Project*

[W6.5] Documentation Tools ★★

Markdown

W6.5a ★★★★: OPTIONAL

Implementation → Documentation → Tools → Markdown → What

W6.5b ★★★★: OPTIONAL

Implementation → Documentation → Tools → Markdown → How

AsciiDoc

W6.5c ★★

Implementation → Documentation → Tools → AsciiDoc → What

🏆 Can explain AsciiDoc

AsciiDoc is similar to Markdown but has more powerful (but also more complex) syntax.

- AsciiDoc Writers Guide --from Asciidoctor.org

This site is not ready yet! The updated version will be available soon.

Week 7 [Mar 2] - Topics

Detailed Table of Contents

[W7.1] Requirements: Use Cases ★

W7.1a ★ Requirements → Specifying Requirements → Use Cases → Introduction

Can explain use cases

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.

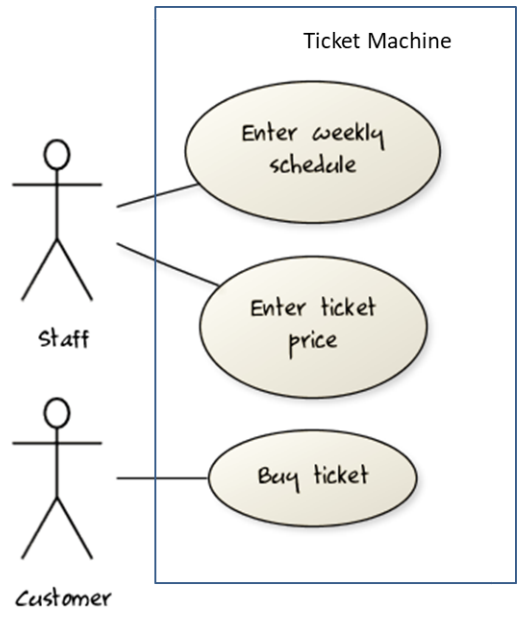
```
System: Online Banking System (OBS)
Use case: UC23 - Transfer Money
Actor: User
MSS:
  1. User chooses to transfer money.
  2. OBS requests for details of the transfer.
  3. User enters the requested details.
  4. OBS requests for confirmation.
  5. User confirms.
  6. OBS transfers the money and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

- System: A Learning Management System (LMS)
- Actor: Student
- Use Case: Upload file
    1. Student requests to upload file
    2. LMS requests for the file location
    3. Student specifies the file location
    4. LMS uploads the file

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.

W7.1b ★★

Requirements → Specifying Requirements → Use Cases → Identifying

Can use use cases to list functional requirements of a simple system

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

Actors in Use Cases

Actor: An actor (in a use case) is a role played by a user. An actor can be a human or another system. Actors are not part of the system; they reside outside the system.

Some example actors for a Learning Management System

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

A use case can involve multiple actors.

- Software System: LearnSys
- Use case: UC01 conduct survey
- Actors: Staff, Student

An actor can be involved in many use cases.

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

A single person/system can play many roles.

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

Many persons/systems can play a single role.

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

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

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

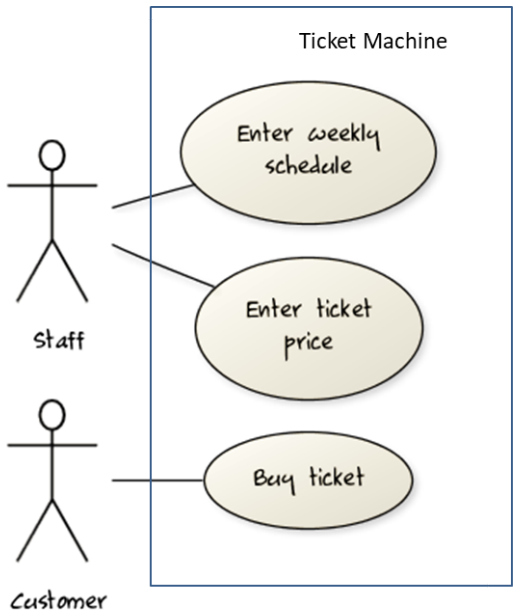
- System: LearnSys
- Use cases:
  a. Conduct a survey
  b. Take the survey
  c. Answer survey question

💡 While modeling user-system interactions,

- Start with high level use cases and progressively work toward lower level use cases.
- Be mindful at which level of details you are working on and not to mix use cases of different levels.

Consider a simple movie ticket vending machine application. Every week, the theatre staff will enter the weekly schedule as well as ticket price for each show. A customer sees the schedule and the ticket price displayed at the machine. There is a slot to insert money, a keypad to enter a code for a movie, a code for the show time, and the number of tickets. A display shows the customer's balance inside the machine. A customer may choose to cancel a transaction before pressing the “buy” button. Printed tickets can be collected from a slot at the bottom of the machine. The machine also displays messages such as "Please enter more money”, “Request fewer tickets" or "SOLD OUT!”. Finally, a "Return Change" button allows the customer to get back his unspent money.

Draw a use case diagram for the above requirements.

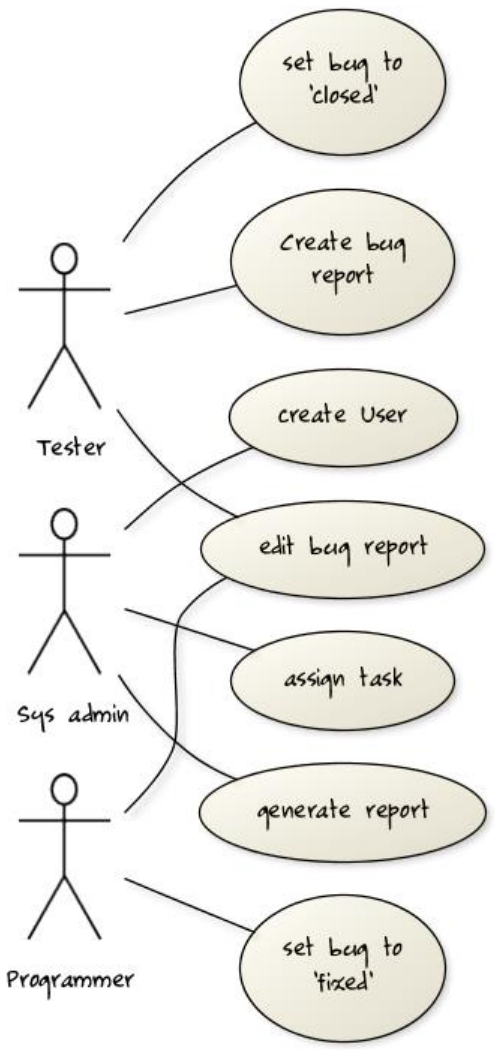


Note that most of the details in the description are better given as part of the use case description rather than as low-level use cases in the diagram.

A software house wishes to automate its Quality Assurance division.

The system is to be used by Testers, Programmers and System Administrators. Only an administrator can create new users and assign tasks to programmers. Any tester can create a bug report, as well as set the status of a bug report as ‘closed’. Only a programmer can set the state of a bug report to ‘fixed’, but a programmer cannot set the status of a bug report to ‘closed’. Each tester is assigned just one task at a time. A task involves testing of a particular component for a particular customer. Tester must document the bugs they find. Each bug is given a unique identifier. Other information recorded about the bug is component id, severity, date and time reported, programmer who is assigned to fix it, date fixed, date retested and date closed. The system keeps track of which bugs are assigned to which programmer at any given time. It should be able to generate reports on the number of bugs found, fixed and closed e.g. number of bugs per component and per customer; number of bugs found by a particular tester ; number of bugs awaiting to be fixed; number of bugs awaiting to be retested; number of bugs awaiting to be assigned to programmers etc.

Develop a use case diagram to capture their requirements given below.



Explanation: The given description contains information not relevant to use case modeling. Furthermore, the description is not enough to complete the use case diagram All these are realities of real projects. However, the process of trying to create this use case diagram prompts us to investigate issues such as:

- Is ‘edit bug report’ a use case or editing the bug report is covered by other use cases such as those for setting the status of bug reports? If it is indeed a separate use case, who are the actors of that use case?
- Does ‘assign task’ simply means ‘assign bug report’ or is there any other type of tasks?
- There was some mention about Customers and Components. Does the system have to support use cases for creating and maintaining details about those entities? For example, should we have a ‘create customer record’ use case?
- Which actors can perform the ‘generate report’ use case? Are reports generated automatically by the system at a specific time or generated ‘on demand’ when users request to view them? Do we have to treat different types of reports as different use cases (in case some types of reports are restricted to some types of users)? The above diagram assumes (just for illustration) that the report is generated on demand and only the system admin can generate any report.

W7.1c ★★

Requirements → Specifying Requirements → Use Cases → Details

🎓 Requirements → Specifying Requirements → Use Cases → Introduction

🎓 Requirements → Specifying Requirements → Use Cases → Identifying

🏆 **Can specify details of a use case in a structured format**

Writing use case steps

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

> 📦 An example of the main body of a use case.
>
> ```
> 1. Student requests to upload file
> 2. LMS requests for the file location
> 3. Student specifies the file location
> 4. LMS uploads the file
> ```

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

> 📦 This example use case step refers to behaviors not externally visible.
>
> ```
> 1. LMS saves the file into the cache and indicates success.
> ```

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

> 📦 The first example below is not a good use case step because contains UI-specific details. The second one is better because it omits UI-specific details.
>
> 👎 **Bad** : User right-clicks the text box and chooses ‘clear’
>
> 👍 **Good** : User clears the input

A use case description can show loops too.

> 📦 An example of how you can show a loop:
>
> ```
> Software System: Square game
> Use case: UC02 - Play a Game
> Actors: Player (multiple players)
>
>  1. A Player starts the game.
>  2. SquareGame asks for player names.
>  3. Each Player enters his own name.
>  4. SquareGame shows the order of play.
>  5. SquareGame prompts for the current Player to throw die.
>  6. Current Player adjusts the throw speed.
>  7. Current Player triggers the die throw.
>  8. Square Game shows the face value of the die.
>  9. Square Game moves the Player's piece accordingly.
>     Steps 5-9 are repeated for each Player, and for as many rounds as
>     required until a Player reaches the 100th square.
> 10. Square Game shows the Winner.
>
> Use case ends.
> ```

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

> 📦 Note how the MSS in the example below assumes that all entered details are correct and

ignores problems such as timeouts, network outages etc. For example, MSS does not tell us what happens if the user enters an incorrect data.

```
System: Online Banking System (OBS)
Use case: UC23 - Transfer Money
Actor: User MSS:

 1. User chooses to transfer money.
 2. OBS requests for details of the transfer.
 3. User enters the requested details.
 4. OBS requests for confirmation.
 5. OBS transfers the money and displays the new account balance.

Use case ends.
```

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

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

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

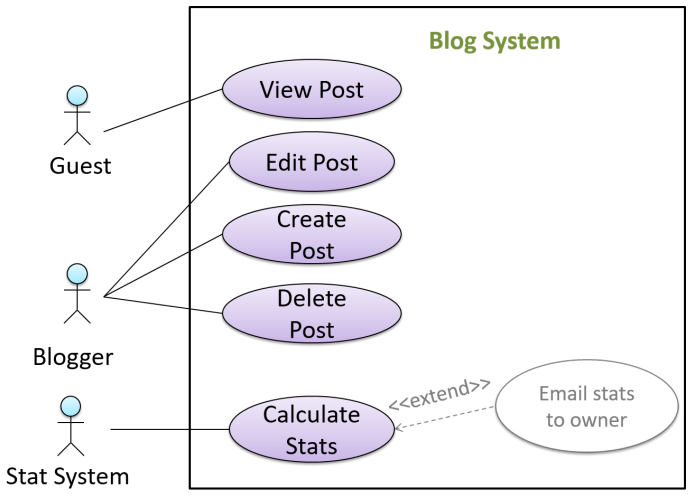
```
Extensions:
  3a. OBS detects an error in the entered data.
    3a1. OBS requests for the correct data.
    3a2. User enters new data
    Steps 3a1-3a2 are repeated until the data entered are correct.
    Use case resumes from step 4.

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

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

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

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

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

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

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



In use case diagrams you can use the `<<extend>>` arrows to show extensions. Note the direction of the arrow is from the extension to the use case it extends and the arrow uses a dashed line.

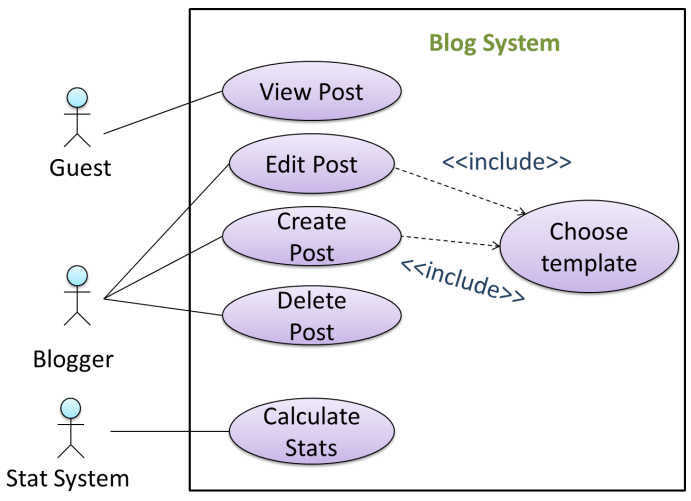
A use case can include another use case. Underlined text is commonly used to show an inclusion of a use case.

> This use case includes two other use cases, one in step 1 and one in step 2.
>
> - Software System: LearnSys
> - Use case: UC01 - Conduct Survey
> - Actors: Staff, Student
> - MSS:
>
> 1. Staff creates the survey (UC44).
> 2. Student completes the survey (UC50).
> 3. Staff views the survey results.
>
> Use case ends.

Inclusions are useful,

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

We use a dotted arrow and a `<<include>>` annotation to show use case inclusions in a use case diagram. Note how the arrow direction is different from the `<<extend>>` arrows.



Preconditions **specify the specific state we expect the system to be in before the use case starts.**

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

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

Guarantees **specify what the use case promises to give us at the end of its operation.**

```
Software System: Online Banking System
```

```
Use case: UC23 - Transfer Money
Actor: User
Preconditions: User is logged in.
Guarantees:

  • Money will be deducted from the source account only if the transfer
    to the destination account is successful
  • The transfer will not result in the account balance going below the
    minimum balance required.

MSS:

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

Complete the following use case (MSS, extensions, etc.). Note that you should not blindly follow how the existing EZ-Link machine operates because it will prevent you from designing a better system. You should consider all possible extensions without complicating the use case too much.

- System: EZ-Link machine
- Use case: UC2 top-up EZ-Link card
- Actor: EZ-Link card user

- System: EZ-Link machine (those found at MRTs)
- Use case: UC2 top-up EZ-Link card
- Actor: EZ-Link card user
- Preconditions: All hardware in working order.
- Guarantees: MSS → the card will be topped-up.
- MSS:
    1. User places the card on the reader.
    2. System displays card details and prompts for desired action.
    3. User selects top-up.
    4. System requests for top-up details (amount, payment option, receipt required?).
    5. User enters details.
    6. System processes cash payment (UC02) or NETS payment (UC03).
    7. System updates the card value.
    8. System indicates transaction as completed.
    9. If requested in step 5, system prints receipt.
    10. User removes the card.

    Use case ends.
- Extensions:

    *a. User removed card or other hardware error detected.

    *a1. System indicates the transaction has been aborted.

    Use case ends.

Notes:

- We assume that the only way to cancel a transaction is by removing the card.
- By not breaking step 4 into further steps, we avoid committing to a particular mechanism to enter data. For example, we are free to accept all data in one screen.
- In step 5, we assume that the input mechanism does not allow any incorrect data.

- System: EZ-Link machine
- Use case: UC03 process NETS payment
- Actor: EZ-Link card user
- Preconditions: A transaction requiring payment is underway.
- Guarantees: MSS → Transaction amount is transferred from user account to EZ-Link company account.
- MSS:
    1. System requests to insert ATM card.
    2. User inserts the ATM card.

3. System requests for PIN.
4. User enters PIN.
5. System reports success.

Use case ends.

- Extensions:

2a. Unacceptable ATM card (damaged or inserted wrong side up).

...

4a. Wrong PIN.

...

4b. Insufficient funds.

...

*a. Connection to the NETS gateway is disrupted.

...

Note: UC02 can be written along similar lines.

write your answer here...

Complete the following use case (MSS, extensions, etc.).

- System: LearnSys (an online Learning Management System)
- Use case: UC01 reply to post in the forum
- Actor: Student

- System: LearnSys
- Use case: UC01 reply to post in the forum
- Actor: Student
- Preconditions: Student is logged in and has permission to post in the forum. The post to which the Student replies already exists.
- MSS:
  1. Student chooses to reply to an existing post.
  2. LearnSys requests the user to enter post details.
  3. Student enters post details.
  4. Student submits the post.
  5. LearnSys displays the post.

  Use case ends.
- Extensions:

*a. Internet connection goes down.

...

*b. LearnSys times out

...

3a. Student chooses to ‘preview’ the post.

3a1. LearnSys shows a preview.

3a2. User chooses to go back to editing.

Use case resumes at step 3.

3b. Student chooses to attach picture/file

...

3c. Student chooses to save the post as a draft.

3c1. LearnSys confirms draft has been saved.

Use case ends.

3d. Student chooses to abort the operation.

...

4a. The post being replied to is deleted by the owner while the reply is being entered.

...

4b. Unacceptable data entered.

...

write your answer here...

Which of these cannot appear as part of a use case description?

- [ ] a. Use case identifier
- [ ] b. Preconditions
- [ ] c. Guarantees
- [ ] d. References to another use case
- [ ] e. Main Success Scenario
- [ ] f. Performance requirements
- [ ] g. Extensions
- [ ] h. Inclusions

(f)

Explanation: Performance requirements are non-functional requirements. They are not captured in use cases.

Identify problems with this use case description.

- System: EZ-Link machine (those found at MRTs)
- Use case: UC2 top-up EZ-Link card
- Actor: EZ-Link card user
- Preconditions: All hardware in working order.
- Guarantees: If MSS completes at least until step 7, the card will be topped-up.
- MSS:
    1. User places the card on the reader.
    2. System displays card details and prompts for desired action.
    3. User selects top-up.
    4. System requests for top-up details (amount, payment option, receipt required?).
    5. User enters details.
    6. System processes cash payment (UC02) or NETS payment (UC03).
    7. System updates the card value.
    8. System indicates transaction as completed.
    9. If requested in step 5, system prints receipt.
    10. User removes the card.

    Use case ends.
- Extensions:

    *a. User removed card or other hardware error detected.

    *a1. System indicates the transaction has been aborted.

    Use case ends.

- [ ] a. It does not consider ‘system crash’ scenario.
- [ ] b. It does not contain enough UI details.
- [ ] c. The extension given is in fact an inclusion.
- [ ] d. No post conditions are given.
- [ ] e. ‘Use case ends’ is duplicated.

None.

Explanation: Catastrophic failures such as ‘system crash’ need not be included in a use case. A use case is not supposed to contain UI details. Post conditions are optional. It is not a problem to have multiple exit points for a use case.

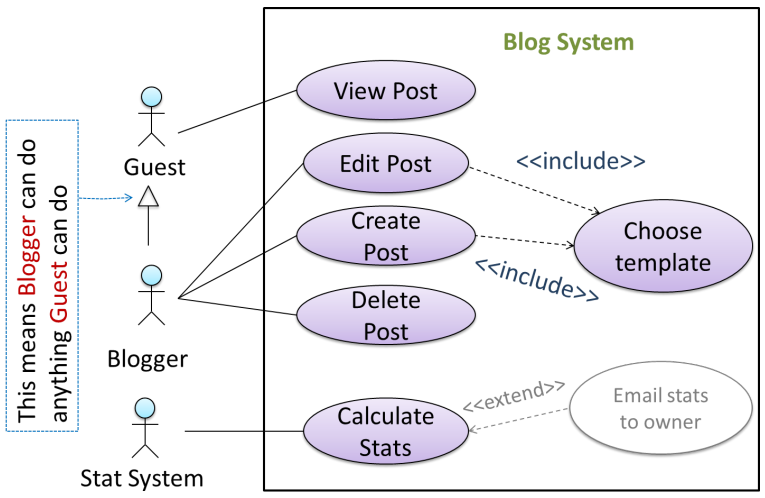
W7.1d ★★★

Requirements → Specifying Requirements → Use Cases → Usage

Can optimize the use of use cases

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

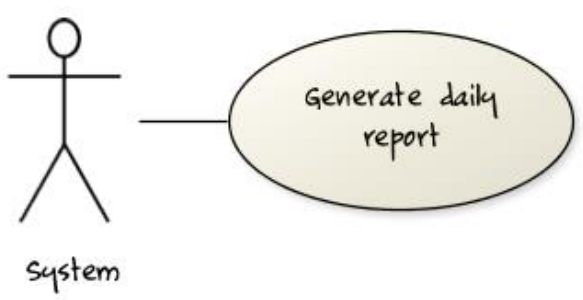
In this example, actor `Blogger` can do all the use cases the actor `Guest` can do, as a result of the actor generalization relationship given in the diagram.



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

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

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



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

We recommend that you follow the latter view (i.e. not to use System as a user). Limit use cases for modeling behaviors that involve an external actor.

UML is not very specific about the text contents of a use case. Hence, there are many styles for writing use cases. For example, the steps can be written as a continuous paragraph. Use cases should be easy to read. Note that there is no strict rule about writing all details of all steps or a need to use all the elements of a use case.

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

- Because they use a simple notation and plain English descriptions, they are easy for users to understand and give feedback.
- They decouple user intention from mechanism (note that use cases should not include UI-specific details), allowing the system designers more freedom to optimize how a functionality is provided to a user.

- Identifying all possible extensions encourages us to consider all situations that a software product might face during its operation.
- Separating typical scenarios from special cases encourages us to optimize the typical scenarios.

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

What are the advantages of using use cases (the textual form) for requirements modelling?

- ☐ a. They can be fairly detailed but still natural enough for users for users to understand and give feedback.
- ☐ b. The UI-independent nature of use case specification allows the system designers more freedom to decide how a functionality is provided to a user.
- ☐ c. Extensions encourage us to consider all situations a software product might face during its operations.
- ☐ d. They encourage us to identify and optimize the typical scenario of usage over exceptional usage scenarios.

(a) (b) (c) (d)

Which of these are correct?

- ☐ a. Use case are not very suitable for capturing non-functional requirements.
- ☐ b. Use case diagrams are less detailed than textual use cases.
- ☐ c. Use cases are better than user stories.
- ☐ d. Use cases can be expressed at different levels of abstraction.

(a)(b)(d)

Explanation: It is not correct to say one format is better than the other. It depends on the context.

[W7.2] Design Principles: Basics ★★

W7.2a ★★★ Design → Introduction → What

Can explain what is software design

> Design in 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 the 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.

Abstraction

W7.2b ★★

Design → Design Fundamentals → Abstraction → What

Can explain abstraction

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 needs 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, we 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 abstractions*.

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 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

W7.2c ★★

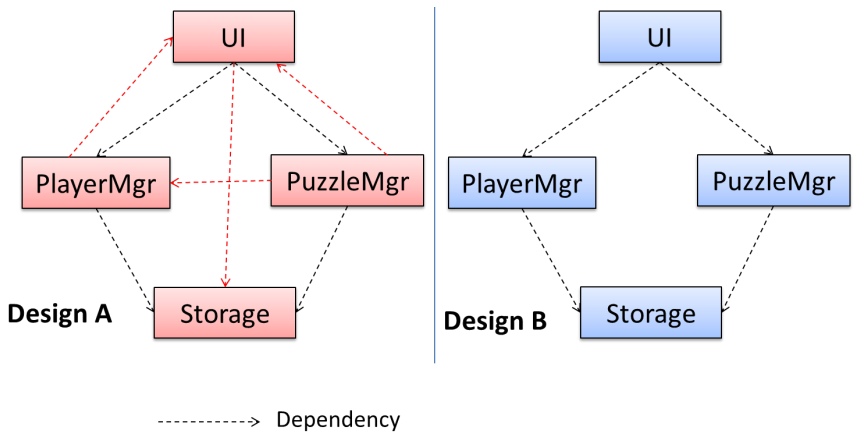
Design → Design Fundamentals → Coupling → What

Can explain coupling

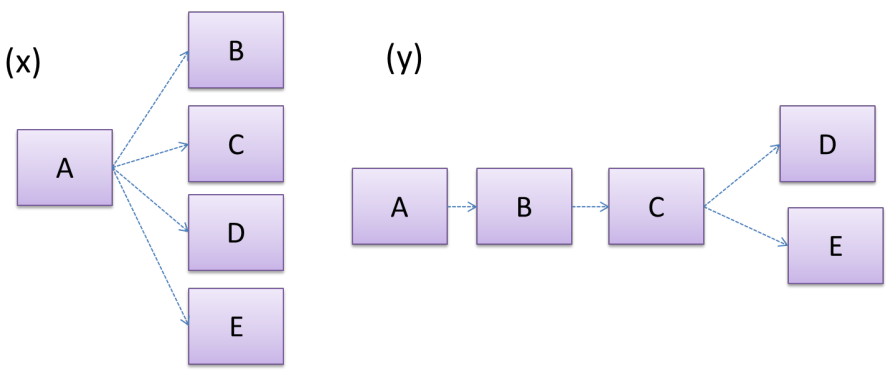
Coupling* is a measure of the degree of *dependence between components, classes, methods, etc. Low coupling indicates that a component is less dependent on other components. **High coupling (aka *tight* coupling or *strong* coupling) is discouraged** due to the following disadvantages:

- **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 a more coupling between the components than design `B`.



Discuss the coupling levels of alternative designs *x* and *y*.



Overall coupling levels in *x* and *y* seem to be similar (neither has more dependencies than the other). (Note that the number of dependency links is not a definitive measure of the level of coupling. Some links may be stronger than the others.). However, in *x*, `A` is highly-coupled to the rest of the system while `B`, `C`, `D`, and `E` are standalone (do not depend on anything else). In *y*, no component is as highly-coupled as `A` of *x*. However, only `D` and `E` are standalone.

write your answer here...

Explain the link (if any) between regressions and coupling.

When the system is highly-coupled, the risk of regressions is higher too e.g. when component `A` is modified, all components 'coupled' to component `A` risk 'unintended behavioral changes'.

write your answer here...

Discuss the relationship between coupling and testability.

Coupling decreases testability because if the SUT is coupled to many other components it becomes difficult to test the SUI in isolation of its dependencies.

write your answer here...

Choose the correct statements.

- [ ] a. As coupling increases, testability decreases.
- [ ] b. As coupling increases, the risk of regression increases.
- [ ] c. As coupling increases, the value of automated regression testing increases.
- [ ] d. As coupling increases, integration becomes easier as everything is connected together.
- [ ] e. As coupling increases, maintainability decreases.

(a)(b)(c)~~(d)~~(e)

Explanation: High coupling means either more components require to be integrated at once in a big-bang fashion (increasing the risk of things going wrong) or more drivers and stubs are required when integrating incrementally.

W7.2d ★★★

Design → Design Fundamentals → Coupling → How

Design → Design Fundamentals → Coupling → What

Can reduce coupling

X is *coupled* to Y if a change to Y can potentially require a change in X.

If `Foo` class calls the method `Bar#read()`, `Foo` is coupled to `Bar` because a change to `Bar` can potentially (but not always) require a change in the `Foo` class e.g. if the signature of the `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

Which of these indicate a coupling between components A and B?

- [ ] a. component A has access to internal structure of component B.
- [ ] b. component A and B are written by the same developer.
- [ ] c. component A calls component B.
- [ ] d. component A receives an object of component B as a parameter.
- [ ] e. component A inherits from component B.
- [ ] f. components A and B have to follow the same data format or communication protocol.

(a)~~(b)~~(c)(d)(e)(f)

Explanation: Being written by the same developer does not imply a coupling.

W7.2e ★★★★: OPTIONAL

Design → Design Fundamentals → Coupling → Types of Coupling

Cohesion

W7.2f ★★

Design → Design Fundamentals → Cohesion → What

🏆 Can explain cohesion

***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 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.

W7.2g ★★★

Design → Design Fundamentals → Cohesion → How

🎓 Design → Design Fundamentals → Cohesion → What

🏆 Can increase cohesion

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 include 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.

Compare the cohesion of the following two versions of the `EmailMessage` class. Which one is more cohesive and why?

```java
// version-1
class EmailMessage {
    private String sendTo;
          private String subject;
    private String message;

    public EmailMessage(String sendTo, String subject, String message) {
        this.sendTo = sendTo;
        this.subject = subject;
        this.message = message;
    }

    public void sendMessage() {
        // sends message using sendTo, subject and message
    }
}

// version-2
class EmailMessage {
    private String sendTo;
    private String subject;
    private String message;
    private String username;

    public EmailMessage(String sendTo, String subject, String message) {
        this.sendTo = sendTo;
        this.subject = subject;
        this.message = message;
    }

    public void sendMessage() {
        // sends message using sendTo, subject and message
    }

    public void login(String username, String password) {
        this.username = username;
        // code to login
    }
}
```

Version 2 is less cohesive.

Explanation: Version 2 is handling functionality related to login, which is not directly related to the concept of ‘email message’ that the class is supposed to represent. On a related note, we can improve the cohesion of both versions by removing the sendMessage functionality. Although sending message is related to emails, this class is supposed to represent an email message, not an email server.

write your answer here...

[W7.3] **Basic Design Approaches** ★★★

W7.3a ★★★

Design → Design Approaches → Top-Down and Bottom-Up Design

🎓 Design → Design Approaches → Multi-Level Design → What

🏆 Can explain top-down and bottom-up design

Multi-level design can be done in a top-down manner, bottom-up manner, or as a mix.

- Top-down: Design the high-level design first and flesh out the lower levels later. This is especially useful when designing big and novel systems where the high-level design needs to be stable before lower levels can be designed.
- Bottom-up: Design lower level components first and put them together to create the higher-level systems later. This is not usually scalable for bigger systems. One instance where this approach might work is when designing a variations of an existing system or re-purposing existing components to build a new system.
- Mix: Design the top levels using the top-down approach but switch to a bottom-up approach when designing the bottom levels.

Top-down design is better than bottom-up design.

- ○ True
- ○ False

False

Explanation: Not necessarily. It depends on the situation. Bottom-up design may be preferable when there are lot of existing components we want to reuse.

W7.3b ★★★

Design Approaches → Agile Design → Agile Design

🏆 Can explain agile design

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

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

Agile design camp expects the design to change over the product's lifetime.

- ○ True
- ○ False

True

Explanation: Yes, that is why they do not believe in spending too much time creating a detailed and full design at the very beginning. However, the architecture is expected to remain relatively stable even in the agile design approach.

[W7.4] IDEs: Intermediate Features ★★★★: OPTIONAL

W7.4a ★★★★: OPTIONAL
Tools → Intellij IDEA → Productivity Shortcuts

[W7.5] Integration Approaches ★★

W7.5a ★★
Implementation → Integration → Approaches → Late-and-One-Time vs Early-and-Frequent

🎓 Implementation → Integration → Introduction → What

🏆 **Can explain how integration approaches vary based on timing and frequency**

In terms of timing and frequency, there are two general approaches to integration: *late and one-time*, *early and frequent*.

***Late and one-time*: wait till all components are completed and integrate all finished components near the end of the project.**

✖ This approach is not recommended because integration often causes many component incompatibilities (due to previous miscommunications and misunderstandings) to surface which can lead to delivery delays i.e. Late integration → incompatibilities found → major rework required → cannot meet the delivery date.

***Early and frequent*: integrate early and evolve each part in parallel, in small steps, re-integrating frequently.**

📦 A *walking skeleton* can be written first. This can be done by one developer, possibly the one in charge of integration. After that, all developers can flesh out the skeleton in parallel, adding one feature at a time. After each feature is done, simply integrate the new code to the main system.

Here is an animation that compares the two approaches:

W7.5b ★★★

Implementation → Integration → Approaches → Big-Bang vs Incremental Integration

🏆 Can explain how integration approaches vary based on amount merged at a time

***Big-bang integration*: integrate all components at the same time.**

✖ Big-bang is not recommended because it will uncover too many problems at the same time which could make debugging and bug-fixing more complex than when problems are uncovered incrementally.

***Incremental integration*: integrate few components at a time.** This approach is better than the big-bang integration because it surfaces integration problems in a more manageable way.

Here is an animation that compares the two approaches:

Give two arguments in support and two arguments against the following statement.

Because there is no external client, it is OK to use big bang integration for a school project.

Arguments for:

- It is relatively simple; even big-bang can succeed.
- Project duration is short; there is not enough time to integrate in steps.
- The system is non-critical, non-production (demo only); the cost of integration issues is relatively small.

Arguments against:

- Inexperienced developers; big-bang more likely to fail
- Too many problems may be discovered too late. Submission deadline (fixed) can be missed.

- Team members have not worked together before; increases the probability of integration problems.

write your answer here...

W7.5c ★★★★: OPTIONAL

Implementation → Integration → Approaches → Top-Down vs Bottom-Up Integration

[W7.6] Project Mgt: Scheduling and Tracking ★★

W7.6a ★★

Project Management → Project Planning → Milestones

Can explain milestones

A *milestone* is the end of a stage which indicates a significant progress. We 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, we 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

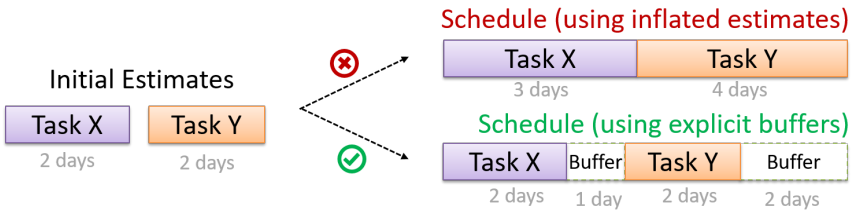
W7.6b ★★★

Project Management → Project Planning → Buffers

Can explain buffers

A *buffer* is a 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 is 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 a feedback to improve future effort estimates.



W7.6c ★★

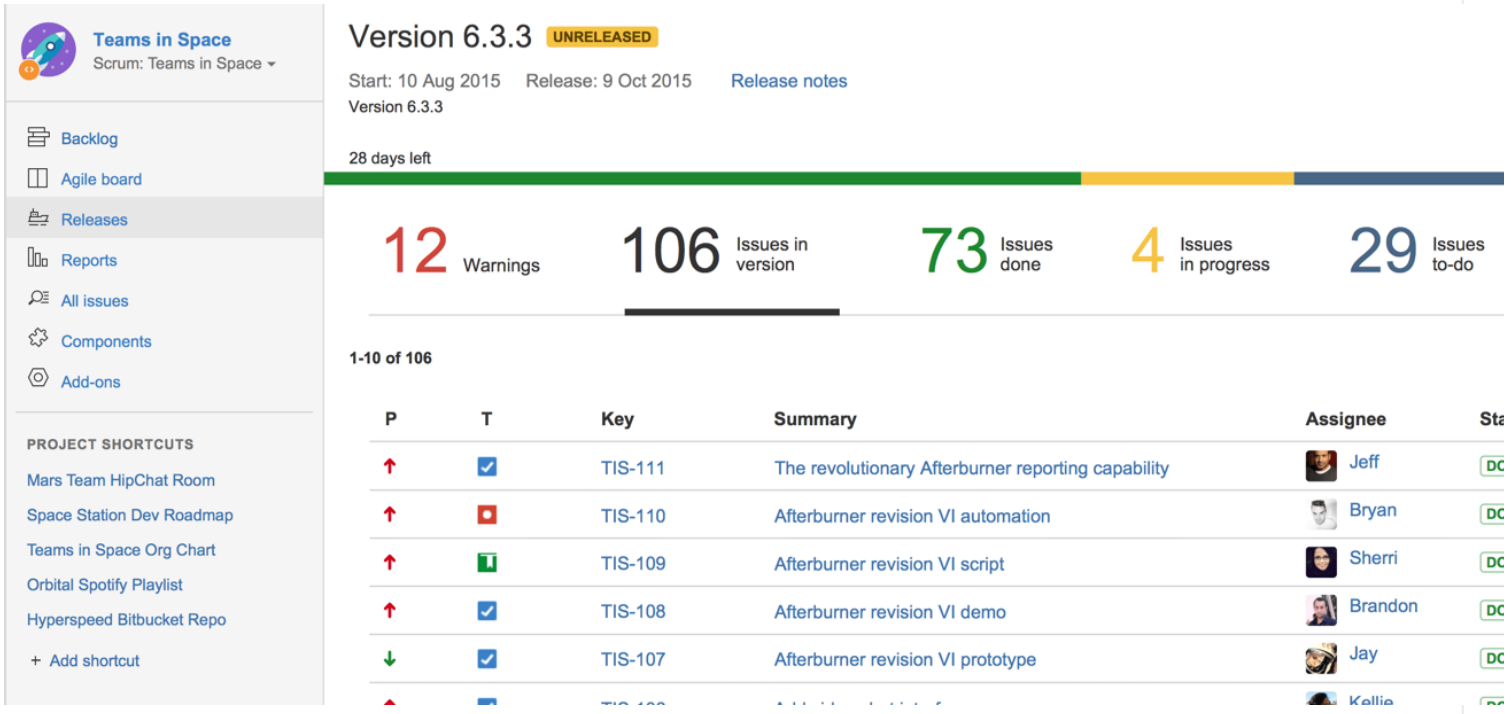
Project Management → Project Planning → Issue Trackers

Can explain 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 track tasks using simple tools as online spreadsheets or general-purpose/light-weight tasks 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):



W7.6d ★★★

Project Management → Project Planning → Work Breakdown Structure

Can explain work breakdown structures

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

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 project for developing a game.

Task ID	Task	Estimated Effort	Prerequisite Task
A	High level design	1 man day	-
B	Detail design 1. User Interface 2. Game Logic 3. Persistency Support	2 man day • 0.5 man day • 1 man day • 0.5 man day	A
C	Implementation 1. User Interface 2. Game Logic 3. Persistency Support	4.5 man day • 1.5 man day • 2 man day • 1 man day	• B.1 • B.2 • B.3
D	System Testing	1 man day	C
E	Planning for next version	1 man day	D

All tasks should be well-defined. In particular, it should be clear as to when the task will be considered *done*.

Some examples of ill-defined tasks and their better-defined counterparts:

👎 Bad	👍 Better
more coding	implement component X
do research on UI testing	find a suitable tool for testing the UI

Which one these project tasks is not well-defined?

- a. Implement the 'delete' feature.
- b. Fix bug #12315.

c. Do some more testing.

d. Choose a tool to draw UML diagrams.

e. Clarify submission requirements with the customer.

(c)

Explanation: ‘More testing’ is not well-defined. How much is ‘more’? ‘Test the delete functionality’ is a better-defined task.

W7.6e ★★★★: OPTIONAL

Project Management → Project Planning → GANTT Charts

W7.6f ★★★★: OPTIONAL

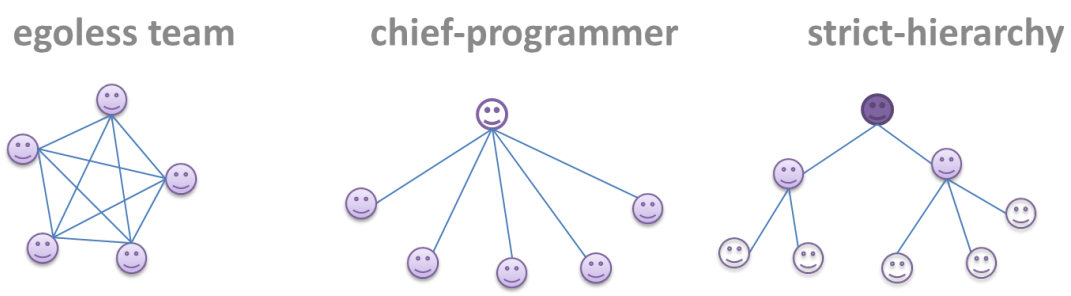
Project Management → Project Planning → PERT Charts

W7.6g ★★★

Project Management → Teamwork → Team Structures

🏆 Can explain common team structures

Given below are three commonly used team structures in software development. Irrespective of the team structure, it is a good practice to assign roles and responsibilities to different team members so that someone is clearly in charge of each aspect of the project. In comparison, the ‘everybody is responsible for everything’ approach can result in more chaos and hence slower progress.



Egoless team

In this structure, **every team member is equal in terms of responsibility and accountability**. When any decision is required, consensus must be reached. This team structure is also known as a *democratic* team structure. This team structure usually finds a good solution to a relatively hard problem as all team members contribute ideas.

However, the democratic nature of the team structure bears a higher risk of falling apart due to the absence of an authority figure to manage the team and resolve conflicts.

Chief programmer team

Frederick Brooks proposed that software engineers learn from the medical surgical team in an operating room. In such a team, there is always a chief surgeon, assisted by experts in other areas. Similarly, in a chief programmer team structure, **there is a single authoritative figure, the chief programmer**. Major decisions, e.g. system architecture, are made solely by him/her and obeyed by all other team members. The chief programmer directs and coordinates the effort of other team members. When necessary, the chief will be assisted by domain specialists e.g. business specialists, database expert, network technology expert, etc. This allows individual group members to concentrate solely on the areas where they have sound knowledge and expertise.

The success of such a team structure relies heavily on the chief programmer. Not only must he be a superb technical hand, he also needs good managerial skills. Under a suitably qualified leader, such a team structure is known to produce successful work. .

Strict hierarchy team

In the opposite extreme of an egoless team, a strict hierarchy team has **a strictly defined organization among the team members**, reminiscent of the military or bureaucratic government. Each team member only works on his assigned tasks and reports to a single “boss”.

In a large, resource-intensive, complex project, this could be a good team structure to reduce communication overhead.

Which team structure is the most suitable for a school project?

- a. Egoless
- b. Chief programmer
- c. Strict hierarchy

(a)

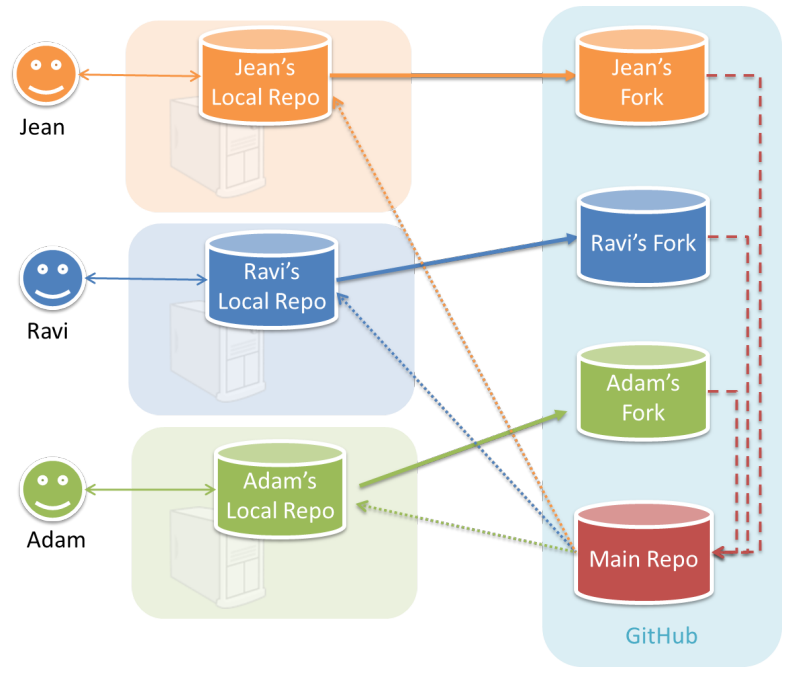
Explanation: Given that students are all peers and beginners, Egoless team structure seems most suitable for a school project. However, given school projects are low-stakes, short-lived, and small, even the other two team structures can be used for them.

[W7.7] Project Mgt: Workflows ★★

W7.7a ★★

Project Management → Revision Control → Forking Flow

Can explain forking workflow



In the *forking workflow*, the 'official' version of the software is kept in a remote repo designated as the 'main repo'. All team members fork the main repo create pull requests from their fork to the main repo.

To illustrate how the workflow goes, let’s assume Jean wants to fix a bug in the code. Here are the steps:

1. Jean creates a separate branch in her local repo and fixes the bug in that branch.
2. Jean pushes the branch to her fork.
3. Jean creates a pull request from that branch in her fork to the main repo.
4. Other members review Jean’s pull request.
5. If reviewers suggested any changes, Jean updates the PR accordingly.
6. When reviewers are satisfied with the PR, one of the members (usually the team lead or a designated 'maintainer' of the main repo) merges the PR, which brings Jean’s code to the main repo.

7. Other members, realizing there is new code in the upstream repo, sync their forks with the new upstream repo (i.e. the main repo). This is done by pulling the new code to their own local repo and pushing the updated code to their own fork.

- A detailed explanation of the Forking Workflow - From Atlassian

W7.7b ★★★

Tools → Git and GitHub → Forking Workflow

Revision Control → Forking Workflow

Can follow Forking Workflow

ℹ This activity is best done as a team. If you are learning this alone, you can simulate a team by using two different browsers to log into GitHub using two different accounts.

1. **One member: set up the team org and the team repo.**
   - Create a GitHub organization for your team. The org name is up to you. We'll refer to this organization as *team org* from now on.
   - Add a team called `developers` to your team org.
   - Add your team members to the `developers` team.
   - **Fork se-edu/samplerepo-workflow-practice to your team org**. We'll refer to this as the *team repo*.
   - Add the forked repo **to the `developers` team.** Give write access.
2. **Each team member: create PRs via own fork**
   - **Fork that repo** from your team org to your own GitHub account.
   - **Create a PR** to add a file `yourName.md` (e.g. `jonhDoe.md`) containing a brief resume of yourself (branch → commit → push → create PR)
3. **For each PR: review, update, and merge.**
   - A team member (not the PR author): Review the PR by adding comments (can be just dummy comments).
   - PR author: Update the PR by pushing more commits to it, to simulate updating the PR based on review comments.
   - Another team member: Merge the PR using the GitHub interface.
   - All members: Sync your local repo (and your fork) with upstream repo. In this case, your *upstream repo* is the repo in your team org.
4. **Create conflicting PRs.**
   - Each team member: Create a PR to add yourself under the `Team Members` section in the `README.md`.
   - One member: in the `master` branch, remove John Doe and Jane Doe from the `README.md`, commit, and push to the main repo.
5. **Merge conflicting PRs one at a time.** Before merging a PR, you'll have to resolve conflicts. Steps:
   - [Optional] A member can inform the PR author (by posting a comment) that there is a conflict in the PR.
   - PR author: Pull the `master` branch from the repo in your team org. Merge the pulled `master` branch to your PR branch. Resolve the merge conflict that crops up during the merge. Push the updated PR branch to your fork.
   - Another member or the PR author: When GitHub does not indicate a conflict anymore, you can go ahead and merge the PR.

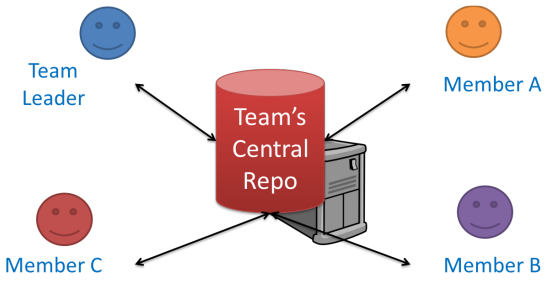
W7.7c ★★★

Project Management → Revision Control → DRCS vs CRCS

Can explain DRCS vs CRCS

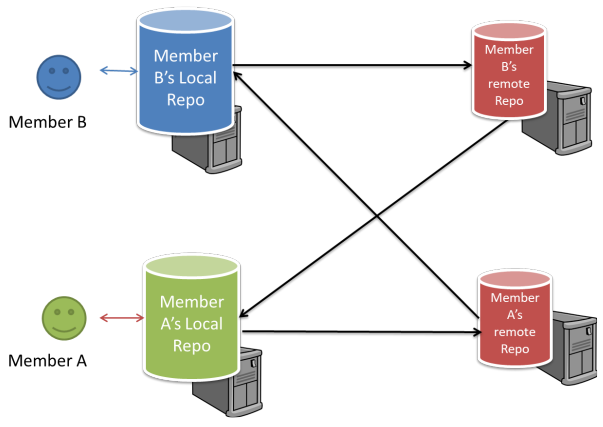
RCS can be done in two ways: the *centralized* way and the *distributed* way.

Centralized RCS (CRCS for short)uses a central remote repo that is shared by the team. Team members download ('pull') and upload ('push') changes between their own local repositories and the central repository. Older RCS tools such as CVS and SVN support only this model. Note that these older RCS do not support the notion of a local repo either. Instead, they force users to do all the versioning with the remote repo.



The centralized RCS approach without any local repos (e.g., CVS, SVN)

Distributed RCS (DRCS for short, also known as Decentralized RCS) allows multiple remote repos and pulling and pushing can be done among them in arbitrary ways. The workflow can vary differently from team to team. For example, every team member can have his/her own remote repository in addition to their own local repository, as shown in the diagram below. Git and Mercurial are some prominent RCS tools that support the distributed approach.



The decentralized RCS approach

W7.7d ★★★★: OPTIONAL

Project Management → Revision Control → Feature Branch Flow

W7.7e ★★★★: OPTIONAL

Project Management → Revision Control → Centralized Flow

Week 8 [Mar 9] - Topics

Detailed Table of Contents

[W8.1] [Revisiting] Drawing Class/Object Diagrams ★

W8.1a ★ Design → Modelling → Modelling Structure → Class Diagrams (Basics)

Can use basic-level class diagrams

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

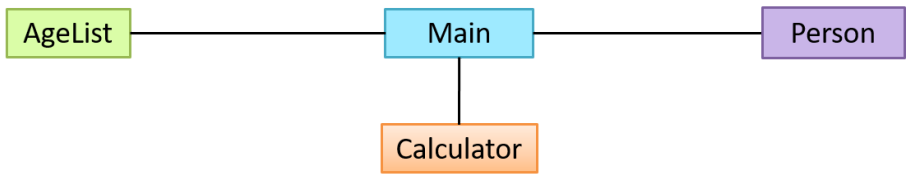
***Associations* are the main connections among the classes in a class diagram.**

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

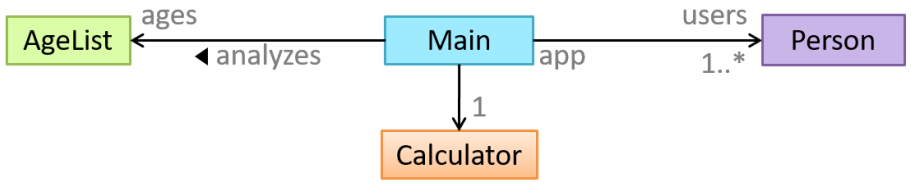
OOP Associations → Multiplicity

UML Class Diagrams → Associations → Multiplicity

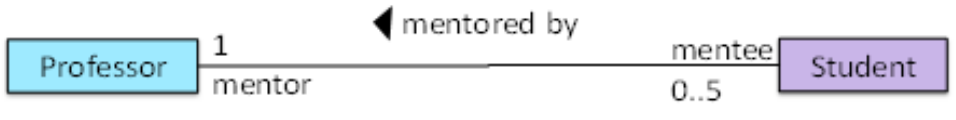
OOP Associations → Navigability

UML Class Diagrams → Associations → Navigability

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



Which association notations are shown in this diagram?

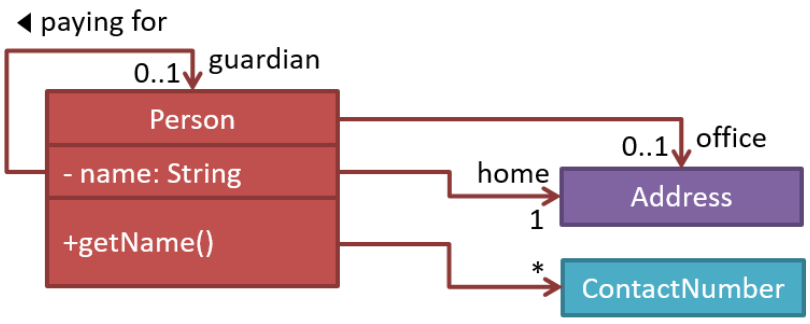


- a. association labels
- b. association roles
- c. association multiplicity
- d. class names

(a) (b) (c) (d)

Explanation: '1' is a *multiplicity*, 'mentored by' is a *label*, and 'mentor' is a *role*.

Explain the associations, navigabilities, and multiplicities in the class diagram below:



Draw a class diagram for the code below. Show the attributes, methods, associations, navigabilities, and multiplicities in the class diagram below:

```
1  class Box {
2      private Item[] parts = new Item[10];
3      private Item spareItem;
4      private Lid lid; // lid of this box
5      private Box outerBox;
6
7      public void open(){
8          //...
9      }
10 }
```

```
1  class Item {
2      public static int totalItems;
3  }
```

```
1  class Lid {
2      Box box; // the box for which this is the lid
3  }
```

W8.1b ★

Design → Modelling → Modelling Structure → Object Diagrams

Can use basic 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

This question is based on the following question from another topic:

Draw a class diagram for the code below. Show the attributes, methods, associations, navigabilities, and multiplicities in the class diagram below:

```
1  class Box {
2      private Item[] parts = new Item[10];
3      private Item spareItem;
4      private Lid lid; // lid of this box
5      private Box outerBox;
6
7      public void open(){
8          //...
9      }
10 }
```

```
1  class Item {
2      public static int totalItems;
3  }
```

```
1  class Lid {
2      Box box; // the box for which this is the lid
3  }
```

Draw an object diagram to match the code. Include objects of all three classes in your object diagram.

W8.1c ★★

Tools → UML → Object vs Class Diagrams

Can distinguish between class diagrams and object diagrams

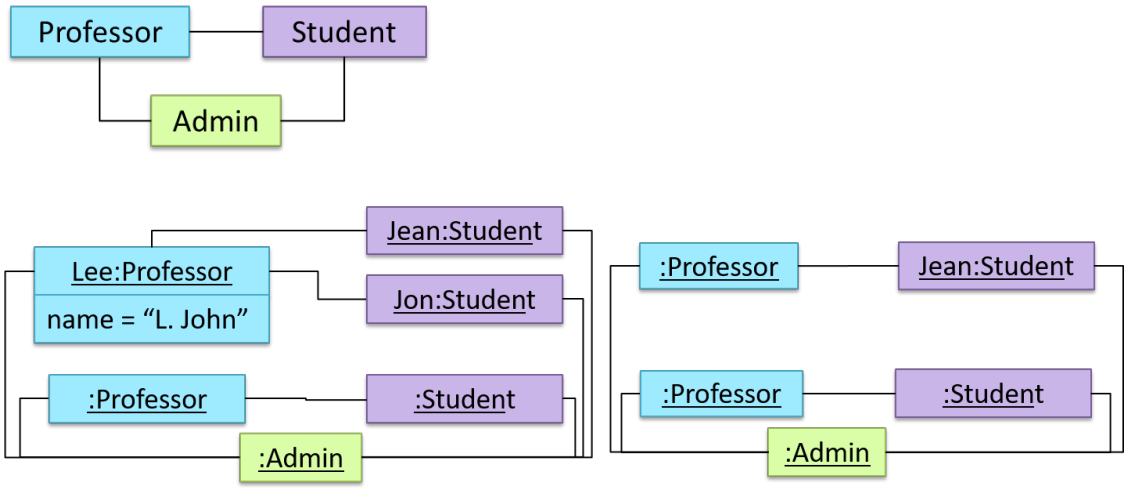
Compared to the notation for a class diagrams, object diagrams differ in the following ways:

- Shows 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

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,

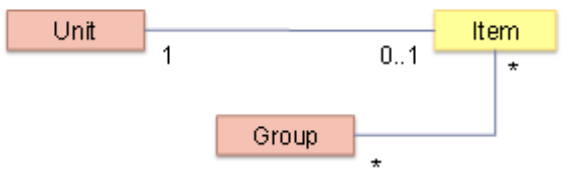
each of these object diagrams shows 'an instance of' the same class diagram.



Which of these class diagrams match the given object diagram?

(1)

(2)



- [ ] 1
- [ ] 2

(1) (2)

Explanation: Both class diagrams allow one `Unit` object to be linked to one `Item` object.

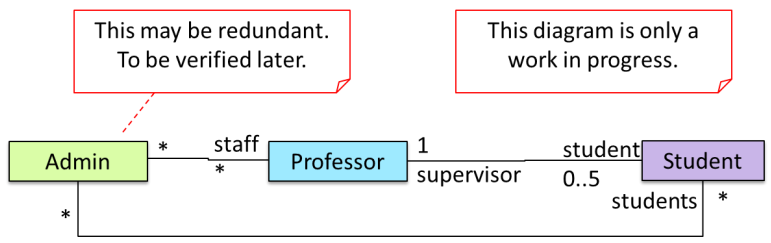
W8.1d ★★★

Tools → UML → Notes

Can use UML 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:



W8.1e ★★★★: OPTIONAL

Tools → UML → Constraints

W8.1f ★★★

Tools → UML → Class Diagrams → Associations as Attributes

Can show an association as an attribute

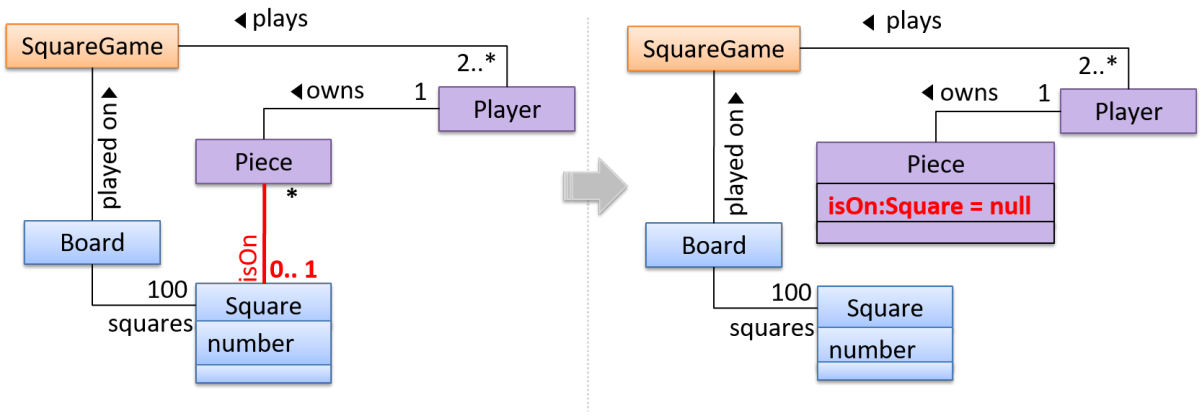
An association can be shown as an attribute instead of a line.

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

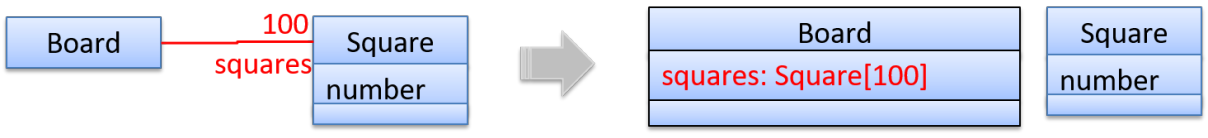
```
name: type [multiplicity] = default value
```

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 is it is easier to spot.

W8.1g ★★

Design → Modelling → Modelling Structure → Class Diagrams - Intermediate

Design → Modeling → Class Diagrams (Basic)

Can use intermediate-level class diagrams

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

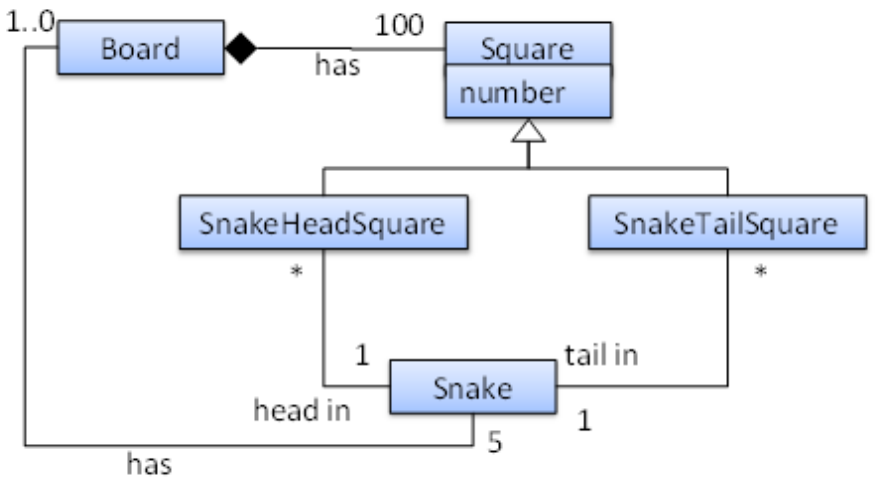
Modeling abstract classes

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

Modeling interfaces

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

Which of these statements match the class diagram?



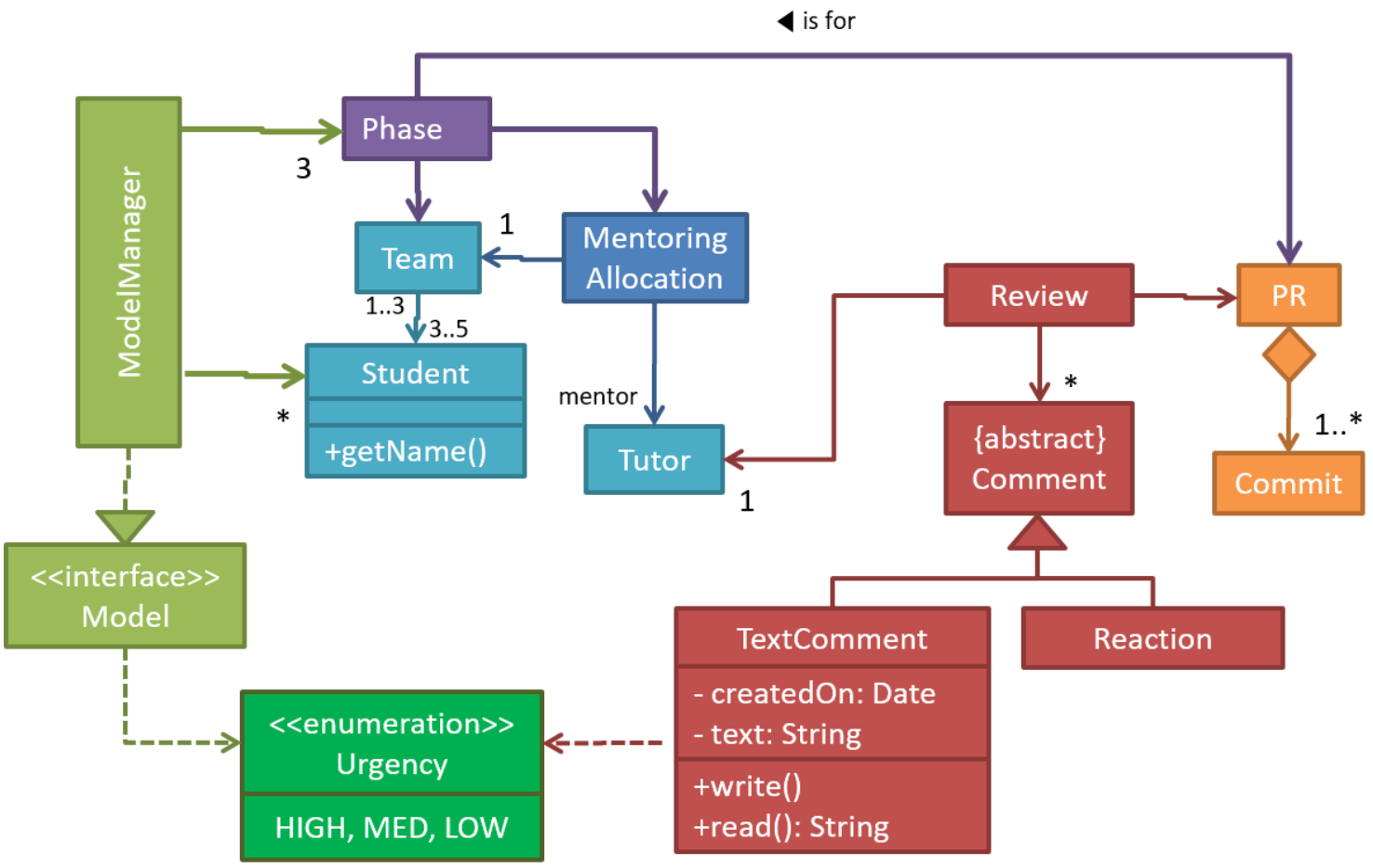
- [ ] a. A `Snake` must belong to at least one `Board` .
- [ ] b. A `SnakeHeadSquare` can contain only one `Snake` head.
- [ ] c. A `Square` can contain a `Snake` head.
- [ ] d. A `Snake` head can be in more than one `SnakeHeadSquare`
- [ ] e. The `Board` has exactly 5 `Snake` s.

~~(a)~~(b)(c)(d)(e)

Explanation:

(a) does not match because a `Snake` may or may not belong to a `Board` (multiplicity is `0..1`)
(b) matches the diagram because the multiplicity given in `1`
(c) matches the diagram because `SnakeHeadSquare` is a `Square` (due to inheritance)
(d) matches the diagram because the multiplicity given is `*`
(e) matches the diagram because the multiplicity given in `5`

Explain the meaning of various class diagram notations in the following class diagram:



Consider the code below:

```
1  public interface Billable {
2      void bill();
3  }
```

```
1  public enum Rating {
2      GOOD, OK, POOR
3  }
```

```
1  public abstract class Item
2           implements Billable {
3      public abstract void print();
4  }
```

```
1  public class Review {
2      private final Rating rating;
3
4      public Review(Rating rating) {
5          this.rating = rating;
6      }
7  }
```

```
1  public class StockItem extends Item
   {
2      private Review review;
3      private String name;
4
5      public StockItem(
6           String name, Rating rating)
   {
7
8          this.name = name;
9                  this.review  =  new
   Review(rating);
10     }
11
12     @Override
13     public void print() {
14         //...
15     }
16
17     @Override
18     public void bill() {
19         //...
20     }
21 }
```

```
1  import java.util.List;
2
3  public class Inventory {
4      private List<Item> items;
5
6      public int getItemCount(){
7          return items.size();
8      }
9
10                     public     void
   generateBill(Billable b){
11         // ...
12     }
13
14     public void add(Item s) {
15         items.add(s);
16     }
17 }
```

(a) Draw a class diagram to represent the code. Show all attributes, methods, associations, navigabilities, visibilities, known multiplicities, and association roles. Show associations as lines.
(b) Draw an object diagram to represent the situation where the inventory has one item with a name `spanner` and a review of `POOR` rating.

W8.1h ★★★

Paradigms → OOP → Associations → Association Classes

Can explain the meaning of association classes

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

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

UML Class Diagrams → Association Classes → What

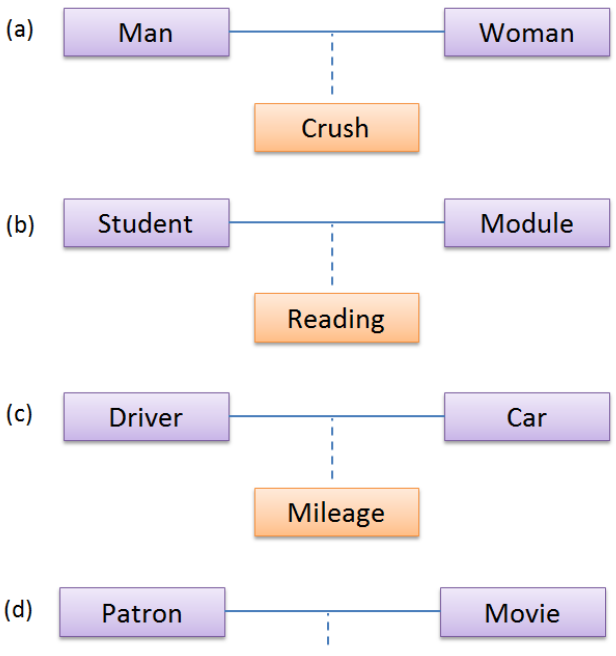
Implementing association classes

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

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

```
class Transaction{

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

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

Which one of these is the suitable as an Association Class?



'Crush' as in 'have a crush'

'Reading' means taking the module

Mileage = total distance the car has been driven

[W8.2] [Revisiting] Drawing Sequence Diagrams

★★

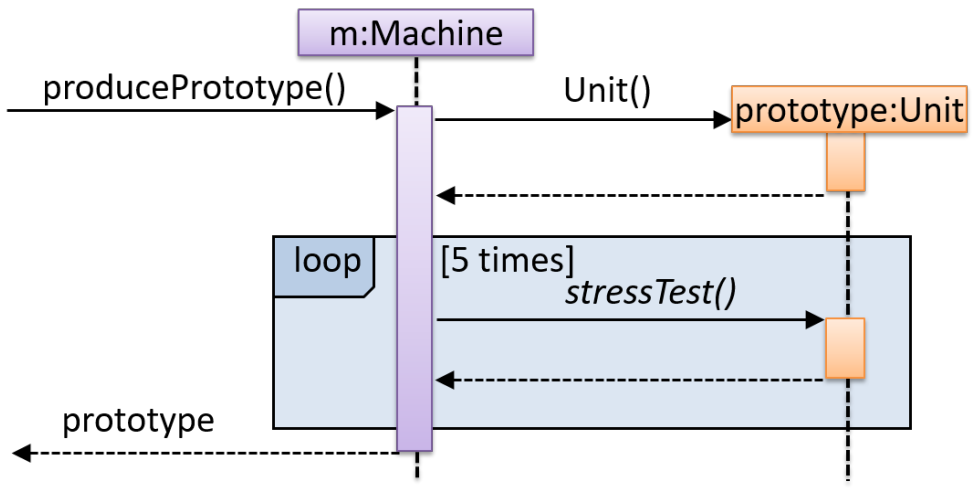
W8.2a ★★

Design → Modelling → Modelling Behaviors Sequence Diagrams - Basic

Can draw basic sequence diagrams

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

Explain in your own words the interactions illustrated by this Sequence Diagram:



Consider the code below:

```
1  class Person{
2      Tag tag;
3      String name;
4
```

```
1  class Tag{
2      Tag(String value){
3          //...
4      }
5  }
```

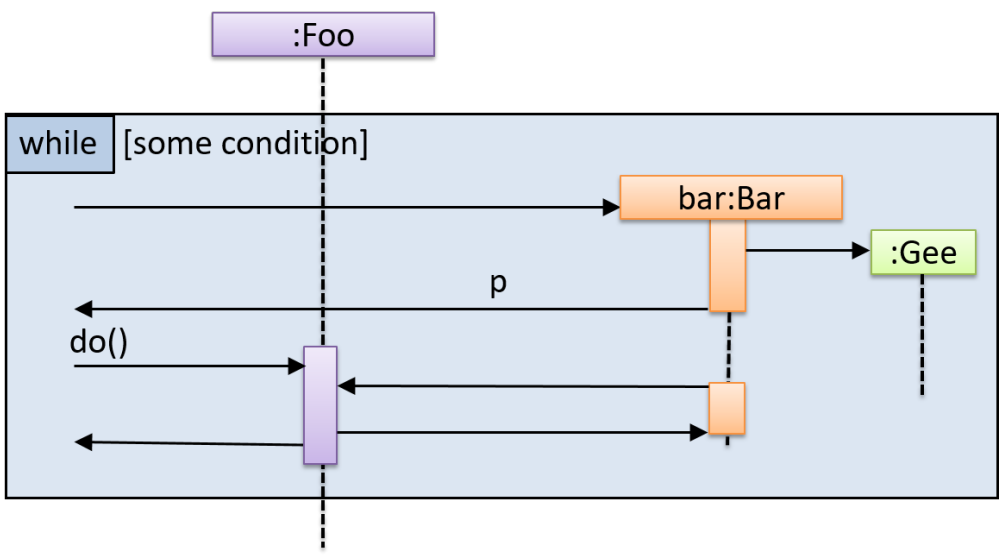
```
5           Person(String  personName,  String
tagName){
6           name = personName;
7           tag = new Tag(tagName);
8       }
9   }
```

```
6
7   class PersonList{
8         void  addPerson(Person
p){
9           //...
10      }
11  }
```

Draw a sequence diagram to illustrate the object interactions that happen in the code snippet below:

```
1  PersonList personList = new PersonList();
2  while (hasRoom){
3      Person p = new Person("Adam", "friend");
4      personList.addPerson(p);
5  }
```

Find notation mistakes in the sequence diagram below:



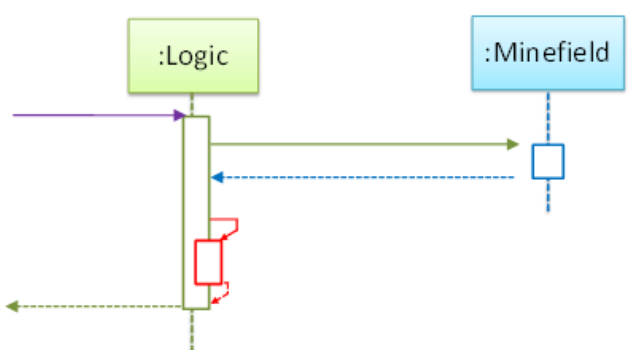
W8.2b ★★★

Design → Modelling → Modelling Behaviors Sequence Diagrams - Intermediate

Can draw intermediate-level sequence diagrams

- 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

What's going on here?

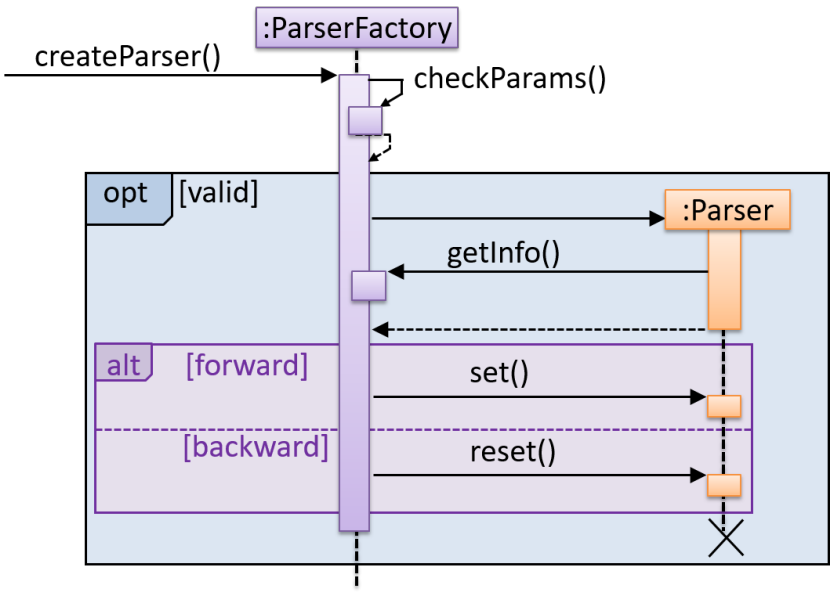


- a. `Logic` object is executing a parallel thread.
- b. `Logic` object is executing a loop.

- c. `Logic` object is creating another `Logic` instance.
- d. One of `Logic` object's methods is calling another of its methods.
- e. `Minefield` object is calling a method of `Logic` .

(d)

Explain the interactions depicted in this sequence diagram.



First, the `createParser()` method of an existing `ParserFactory` object is called. Then, ...

Draw a sequence diagram to represent this code snippet.

```
1  if (isFirstPage) {
2      new Quote().print();
3  }
```

The `Quote` class:

```
1  class Quote{
2
3      String q;
4
5      Quote(){
6          q = generate();
7      }
8
9      String generate(){
10         // ...
11     }
12
13     void print(){
14         System.out.println(q);
15     }
16
17 }
```

- Show `new Quote().print();` as two method calls.
- As the created Quote object is not assigned to a variable, it can be considered as 'deleted' soon after its `print()` method is called.

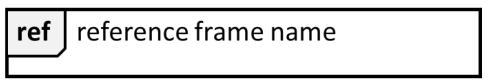
W8.2c ★★★

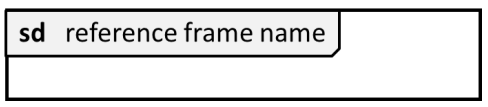
Tools → UML → Sequence Diagrams → Reference Frames

Can interpret sequence diagrams with reference frames

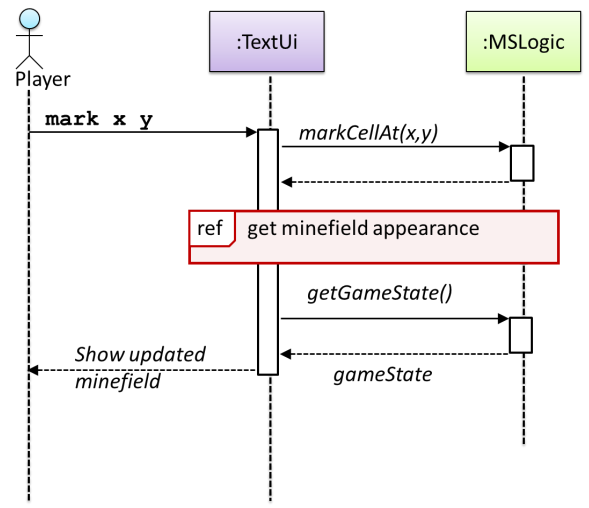
UML uses *ref frame* to allow a segment of the interaction to be omitted and shown as a separate sequence diagram. Reference frames help us to break complicated sequence diagrams into multiple parts or simply to omit details we are not interested in showing.

Notation:

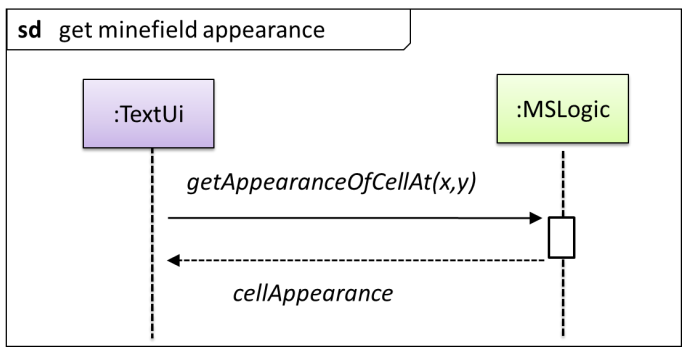




The details of the `get minefield appearance` interactions have been omitted from the diagram.



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



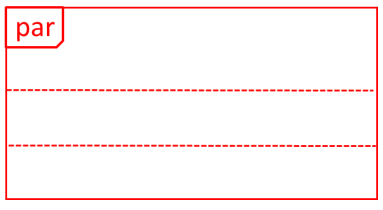
W8.2d ★★★

Tools → UML → Sequence Diagrams → Parallel Paths

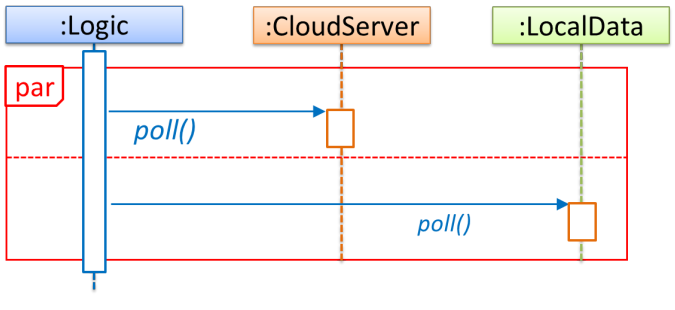
Can interpret sequence diagrams with parallel paths

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.

[W8.3] Testing: Types ★★

Integration Testing

W8.3a ★★
Quality Assurance → Testing → Integration Testing → What

🎓 Quality Assurance → Testing → Unit Testing → What →

🏆 Can explain integration testing

***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 of what the parts are supposed to do vs what the parts are actually doing.

> 📦 Suppose a class `Car` users classes `Engine` and `Wheel`. If the `Car` class assumed a `Wheel` can support 200 mph speed but the actual `Wheel` can only support 150 mph, it is the integration test that is supposed to uncover this discrepancy.

W8.3b ★★★
Quality Assurance → Testing → Integration Testing → How

🎓 Quality Assurance → Testing → Integration Testing → What →

🏆 Can use integration testing

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

> 📦 Suppose a class `Car` uses classes `Engine` and `Wheel`. Here is how you would go about doing pure integration tests:
>
> a) First, unit test `Engine` and `Wheel`.
> b) Next, unit test `Car` in isolation of `Engine` and `Wheel`, using stubs for `Engine` and `Wheel`.
> c) After that, do an integration test for `Car` using it together with the `Engine` and `Wheel` classes to ensure the `Car` integrates properly with the `Engine` and the `Wheel`.

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

> 📦 Here's how a hybrid unit+integration approach could be applied to the same example used above:
>
> (a) First, unit test `Engine` and `Wheel`.
> ~~(b) Next, unit test `Car` in isolation of `Engine` and `Wheel`, using stubs for `Engine` and `Wheel`.~~
> (c) After that, do an integration test for `Car` using it together with the `Engine` and `Wheel` classes

to ensure the `Car` integrates properly with the `Engine` and the `Wheel`. This step should include test cases that are meant to test the unit `Car` (i.e. test cases used in the step (b) of the example above) as well as test cases that are meant to test the integration of `Car` with `Wheel` and `Engine` (i.e. pure integration test cases used of the step (c) in the example above).

Note that you no longer need stubs for `Engine` and `Wheel`. The downside is that `Car` is never tested in isolation of its dependencies. Given that its dependencies are already unit tested, the risk of bugs in `Engine` and `Wheel` affecting the testing of `Car` can be considered minimal.

System Testing

W8.3c ★★

Quality Assurance → Testing → System Testing → What

Can explain system testing

***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' having pushed beyond its limits.

Suppose the SUT is a browser 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: load a web page containing more than 5000 characters.
* Expected behavior: abort the loading of the page and show 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.

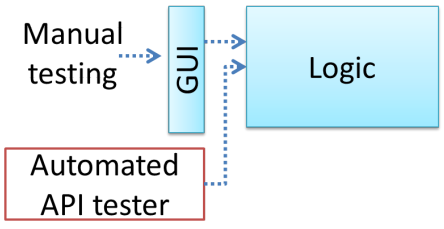
W8.3d ★★★

Quality Assurance → Testing → Test Automation → Automated Testing of GUIs

Can explain automated GUI testing

If a software product has a GUI component, all product-level testing (i.e. the types of testing mentioned above) need to be done using the GUI. However, **testing the GUI is much harder than testing the CLI (command line interface) or API**, for the following reasons:

- Most GUIs can support a large number of different operations, many of which can be performed in any arbitrary order.
- GUI operations are more difficult to automate than API testing. Reliably automating GUI operations and automatically verifying whether the GUI behaves as expected is harder than calling an operation and comparing its return value with an expected value. Therefore, automated regression testing of GUIs is rather difficult.
- The appearance of a GUI (and sometimes even behavior) can be different across platforms and even environments. For example, a GUI can behave differently based on whether it is minimized or maximized, in focus or out of focus, and in a high resolution display or a low resolution display.



Moving as much logic as possible out of the GUI can make the GUI testing easier. That way, you can bypass the GUI to test the rest of the system using automated API testing. While this still requires the GUI to be tested, the number of such test cases can be reduced as most of the system has been tested using automated API testing.

There are testing tools that can automate GUI testing.

Some tools used for automated GUI testing:

- **TestFx** can do automated testing of JavaFX GUIs
- **VisualStudio** supports 'record replay' type of GUI test automation.
- **Selenium** can be used to automate testing of Web application UIs

  demo video of automated testing of a Web application

GUI testing is usually easier than API testing because it doesn't require any extra coding.

- True
- False

False

Acceptance Testing

W8.3e ★★

Quality Assurance → Testing → Acceptance Testing → What

Can explain acceptance testing

***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.

W8.3f ★★★

Quality Assurance → Testing → Acceptance Testing → Acceptance vs System Testing

Can explain the differences between system testing and acceptance 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 vs 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 solve those problems (e.g. explain the email search feature)
specifies the interface available for intended end-users	could contain additional APIs not available for end-users (for the use of developers/testers)

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.

Choose the correct statements about system testing and acceptance testing.

- ☐ a. Both system testing and acceptance testing typically involve the whole system.
- ☐ b. System testing is typically more extensive than acceptance testing.
- ☐ c. System testing can include testing for non-functional qualities.
- ☐ d. Acceptance testing typically has more user involvement than system testing.

- [ ] e. In smaller projects, the developers may do system testing as well, in addition to developer testing.
- [ ] f. If system testing is adequately done, we need not do acceptance testing.

(a)(b)(c)(d)(e)~~(f)~~

Explanation:

(b) is correct because system testing can aim to cover all specified behaviors and can even go beyond the system specification. Therefore, system testing is typically more extensive than acceptance testing.

(f) is incorrect because it is possible for a system to pass system tests but fail acceptance tests.

Alpha/Beta Testing

W8.3g ★★★

Quality Assurance → Testing → Alpha/Beta Testing → What

Can explain alpha and beta testing

***Alpha testing* is performed by the users, under controlled conditions set by the software development team.**

***Beta testing* is performed by a selected subset of target users of the system in their natural work setting.**

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.

Exploratory vs Scripted Testing

W8.3h ★★★

Quality Assurance → Testing → Exploratory and Scripted Testing → What

Can explain exploratory testing and scripted testing

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; we need to look at them soon. But before that, let us 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%, we 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*.

Scripted testing requires tests to be written in a scripting language; Manual testing is called exploratory testing.

- True
- False

False

Explanation: “Scripted” means test cases are predetermined. They need not be an executable script. However, exploratory testing is usually manual.

Which testing technique is better?

- a. error guessing
- b. bug hunting
- c. attack-based testing
- d. reactive testing
- e. These are different names used to describe exploratory testing.

(e)

Explain the concept of exploratory testing using Minesweeper as an example.

When we test the Minesweeper by simply playing it in various ways, especially trying out those that are likely to be buggy, that would be exploratory testing.

write your answer here...

W8.3i ★★★

Quality Assurance → Testing → Exploratory and Scripted Testing → When

Can explain the choice between exploratory testing and scripted testing

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]

Scripted testing is better than exploratory testing.

- ○ True
- ○ False

False

Explanation: Each has pros and cons. Relying on only one is not recommended. A combination is better.

[W8.4] Testing: Intermediate Concepts ★★

Dependency Injection

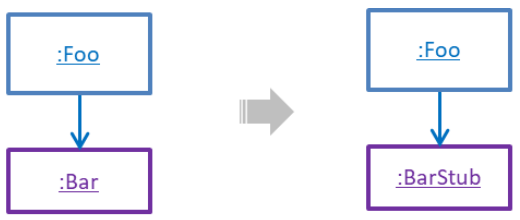
W8.4a ★★★

Quality Assurance → Testing → Dependency Injection → What

Can explain dependency injection

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

A `Foo` object normally depends on a `Bar` object, but we can inject a `BarStub` object so that the `Foo` object no longer depends on a `Bar` object. Now we can test the `Foo` object in isolation from the `Bar` object.



Normal dependency After dependency injection

W8.4b ★★★★: OPTIONAL

Quality Assurance → Testing → Dependency Injection → How

Testability

W8.4c ★★★

Quality Assurance → Testing → Introduction → Testability

Can explain testability

***Testability* is an indication of how easy it is to test an SUT.** As testability depends a lot on the design and implementation. You should try to increase the testability when you design and implement a software. The higher the testability, the easier it is to achieve a better quality software.

Test Coverage

W8.4d ★★

Quality Assurance → Testing → Test Coverage → What

Can explain test coverage

***Test coverage* is a metric used to measure the extent to which testing exercises the code** i.e., how much of the code is 'covered' by the tests.

Here are some examples of different coverage criteria:

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

> `if(x > 2 && x < 44)` is considered one decision point but two conditions.
>
> For 100% branch or decision coverage, two test cases are required:
>
> - `(x > 2 && x < 44) == true` : [e.g. `x == 4`]
> - `(x > 2 && x < 44) == false` : [e.g. `x == 100`]
>
> For 100% condition coverage, three test cases are required
>
> - `(x > 2) == true` , `(x < 44) == true` : [e.g. `x == 4`]
> - `(x < 44) == false` : [e.g. `x == 100`]
> - `(x > 2) == false` : [e.g. `x == 0`]

- **Path coverage** measures coverage in terms of possible paths through a given part of the code executed. 100% path coverage means all possible paths have been executed. A commonly used notation for path analysis is called the *Control Flow Graph (CFG)*.
- **Entry/exit coverage** measures coverage in terms of possible *calls to* and *exits* from the operations in the SUT.

Which of these gives us the highest intensity of testing?

- a. 100% statement coverage
- b. 100% path coverage
- c. 100% branch coverage
- d. 100% condition coverage

(b)

Explanation: 100% path coverage implies all possible execution paths through the SUT have been tested. This is essentially ‘exhaustive testing’. While this is very hard to achieve for a non-trivial SUT, it technically gives us the highest intensity of testing. If all tests pass at 100% path coverage, the SUT code can be considered ‘bug free’. However, note that path coverage does not include paths that are missing from the code altogether because the programmer left them out by mistake.

W8.4e ★★★

Quality Assurance → Testing → Test Coverage → How

Can explain how test coverage works

Measuring coverage is often done using *coverage analysis tools*. Most IDEs have inbuilt support for measuring test coverage, or at least have plugins that can measure test coverage.

Coverage analysis can be useful in improving the quality of testing e.g., if a set of test cases does not achieve 100% branch coverage, more test cases can be added to cover missed branches.

Measuring code coverage in Intellij IDEA

TDD

W8.4f ★★★★: OPTIONAL

Quality Assurance → Testing → Test-Driven Development → What

This site was built with MarkBind 2.14.1 at Tue, 21 Apr 2020, 6:16:45 UTC

Week 9 [Mar 16] - Topics

Detailed Table of Contents

[W9.1] OO Domain Models ★★★

W9.1a ★★★

Design → Modelling → Modelling Structure → Object Oriented Domain Models

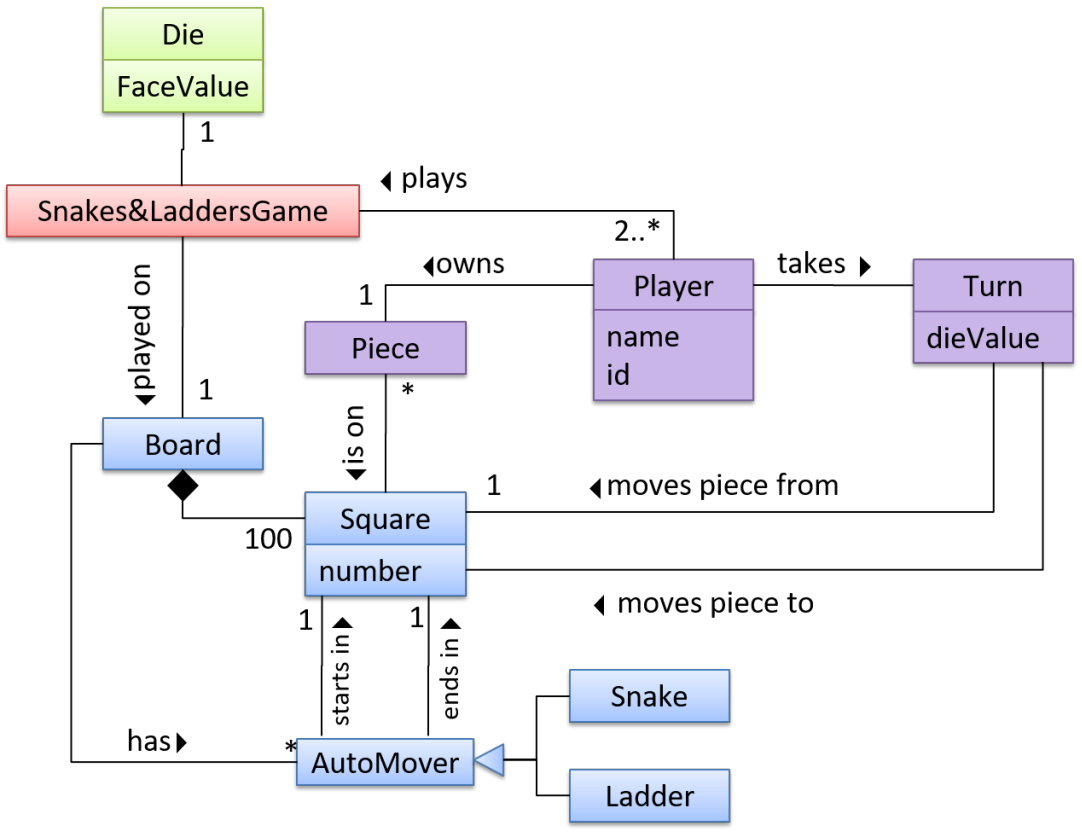
Can explain object oriented domain models

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

Class diagrams can also be used to model objects in the problem domain (i.e. to model how objects actually interact in the real world, before emulating them in the solution). **Class diagrams are used to model the problem domain are called *conceptual class diagrams* or *OO domain models (OODMs)*.**

OO domain model of a snakes and ladders game is given below.

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



The above OO domain model omits the ladder class for simplicity. It can be included in a similar fashion to the Snake class.

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

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

OODM notation is similar to class diagram notation but omit methods and navigability.

This diagram is,

- a. A class diagram.
- b. An object diagram.
- c. An OO domain model, also known as a conceptual class diagram.
- d. Can be either a class diagram or an OO domain model.

(a)

Explanation: The diagram shows navigability which is not shown in an OO domain model. Hence, it has to be a class diagram.

What is the main difference between a class diagram and and an OO domain model?

- a. One is about the problem domain while the other is about the solution domain.
- b. One has more classes than the other.
- c. One shows more details than the other.
- d. One is a UML diagram, while the other is not a UML diagram.

(a)

Explanation: Both are UML diagrams, and use the class diagram notation. While it is true that often a class diagram may have more classes and more details, the main difference is that the OO domain model describes the problem domain while the class diagram describes the solution.

[W9.2] **Activity Diagrams** ★★★

W9.2a ★★★

Design → Modelling → Modelling Behaviors **Activity Diagrams - Basic**

Can use basic-level activity diagrams

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

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

A software that automates the work of an insurance company needs to take into account the workflow of processing an insurance claim.

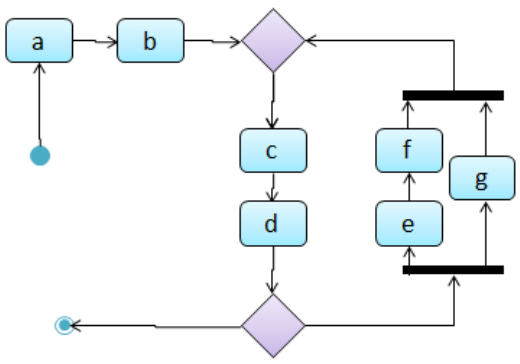
The algorithm of a price of code represents the workflow (i.e. the execution flow) of the code.

UML Activity Diagrams → Introduction → What

- UML Activity Diagrams → Basic Notation → Linear Paths
- UML Activity Diagrams → Basic Notation → Alternate Paths
- UML Activity Diagrams → Basic Notation → Parellel Paths

Which of these sequence of actions is not allowed by the given activity diagram?

- [ ] i. start a b c d end
- [ ] ii. start a b c d e f g c d end
- [ ] iii. start a b c d e g f c d end
- [ ] iv. start a b c d g c d end



(iv)

Explanation: `-e-f-` and `-g-` are parallel paths. Both paths should complete before the execution reaches `c` again.

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

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

Example Activity Diagram

W9.2b ★★★

Design → Modelling → Modelling Behaviors Activity Diagrams - Intermediate

Can use intermediate-level activity diagrams

- UML Activity Diagrams → Intermediate Notation → Rakes
- UML Activity Diagrams → Intermedidate Notation → Swim Lanes

[W9.3] Design principles ★★

W9.3a : ★★★

Principles → Single Responsibility Principle

Can explain 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

- An explanation of the SRP from www.oodesign.com
- Another explanation (more detailed) by Patkos Csaba
- A book chapter on SRP - A book chapter on SRP, written by the father of the principle itself Robert C Martin.

W9.3b : ★★★

Principles → Open-Closed Principle

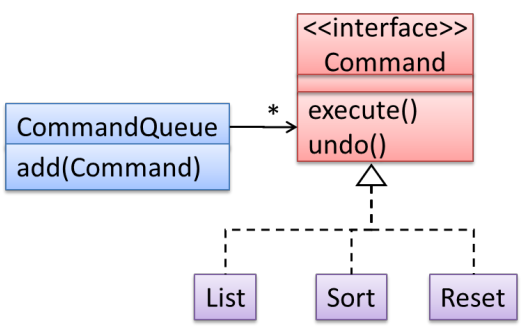
Can explain open-closed principle (OCP)

The Open-Close Principle aims to make a code entity easy to adapt and reuse without needing to modify the code entity itself.

> **Open-Closed Principle (OCP)**: A module should be *open* for extension but *closed* for modification. That is, modules should be written so that they can be extended, without requiring them to be modified. -- proposed by Bertrand Meyer

In object-oriented programming, OCP can be achieved in various ways. This often requires separating the *specification (i.e. interface)* of a module from its *implementation*.

> In the design given below, the behavior of the `CommandQueue` class can be altered by adding more concrete `Command` subclasses. For example, by including a `Delete` class alongside `List`, `Sort`, and `Reset`, the `CommandQueue` can now perform delete commands without modifying its code at all. That is, its behavior was extended without having to modify its code. Hence, it was open to extensions, but closed to modification.



The behavior of a Java generic class can be altered by passing it a different class as a parameter. In the code below, the `ArrayList` class behaves as a container of `Students` in one instance and as a container of `Admin` objects in the other instance, without having to change its code. That is, the behavior of the `ArrayList` class is extended without modifying its code.

```
1 ArrayList students = new ArrayList<Student>();
2 ArrayList admins = new ArrayList<Admin>();
```

Which of these is closest to the meaning of the open-closed principle?

- a. We should be able to change a software module's behavior without modifying its code.
- b. A software module should remain open to modification as long as possible.
- c. A software module should be open to modification and closed to extension.
- d. Open source software rocks. Closed source software sucks.

(a)

Explanation: Please refer the handout for the definition of OCP.

W9.3c ★★

Principles → Separation of Concerns Principle

Can explain 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 `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 on 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.

"Only the GUI class should interact with the user. The GUI class should only concern itself with user interactions". This statement follows from,

- a. A software design should promote separation of concerns in a design.
- b. A software design should increase cohesion of its components.

c. A software design should follow single responsibility principle.

(a)(b)(c)

Explanation: By making 'user interaction' GUI class' sole responsibility, we increase its cohesion. This is also in line with separation of concerns (i.e., we separated the concern of user interaction) and single responsibility principle (GUI class has only one responsibility).

W9.3d : ★★★

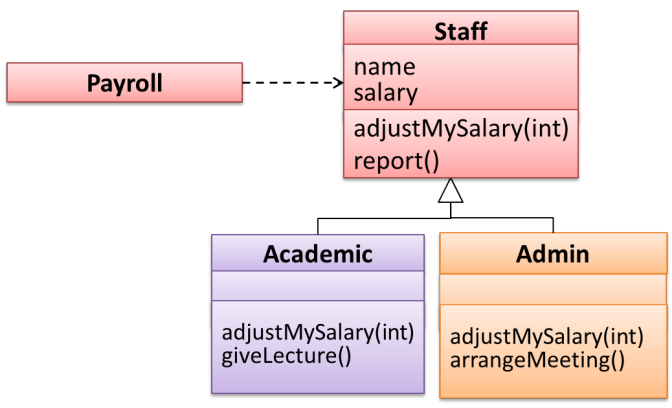
Principles → Liskov Substitution Principle

Can explain Liskov Substitution Principle

Liskov Substitution Principle (LSP): Derived classes must be substitutable for their base classes. -- proposed by Barbara Liskov

LSP sounds same as substitutability but it goes beyond substitutability; **LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.** As we know, Java has language support for substitutability. However, if LSP is not followed, substituting a subclass object for a superclass object can break the functionality of the code.

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



Now consider the following:

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

In the above scenario,

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

Another example

If a subclass imposes more restrictive conditions than its parent class, it violates Liskov Substitution Principle.

True

False

True.

Explanation: If the subclass is more restrictive than the parent class, code that worked with the parent class may not work with the child class. Hence, the substitutability does not exist and LSP has been violated.

W9.3e ★★★

Principles → Law of Demeter

Can explain the Law of Demeter

Law of Demeter (LoD):

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

Also known as

- Don't talk to strangers.
- Principle of least knowledge

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

- The object `O` itself
- Objects passed as parameters of `m`
- Objects created/instantiated in `m` (directly or indirectly)
- Objects from the direct association of `O`

The following code fragment violates LoD due to the reason: while `b` is a 'friend' of `foo` (because it receives it as a parameter), `g` is a 'friend of a friend' (which should be considered a 'stranger'), and `g.doSomething()` is analogous to 'talking to a stranger'.

```
1 void foo(Bar b) {
2     Goo g = b.getGoo();
3     g.doSomething();
4 }
```

LoD aims to prevent objects navigating internal structures of other objects.

An analogy for LoD can be drawn from Facebook. If Facebook followed LoD, you would not be allowed to see posts of friends of friends, unless they are your friends as well. If Jake is your friend and Adam is Jake's friend, you should not be allowed to see Adam's posts unless Adam is a friend of yours as well.

Explain the Law of Demeter using code examples. You are to make up your own code examples. Take Minesweeper as the basis for your code examples.

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

```
setMinefield(Minefiled mf):void
```

Consider the following that can happen inside this operation.

- `mf.init();` : this does not violate LoD since LoD allows calling operations of parameters received.
- `mf.getCell(1,3).clear();` : //this violates LoD because `Logic` is handling `Cell` objects deep inside `Minefield` . Instead, it should be `mf.clearCellAt(1,3);`
- `timer.start();` : //this does not violate LoD because `timer` appears to be an internal component (i.e. a variable) of `Logic` itself.
- `Cell c = new Cell(); c.init();` : // this does not violate LoD because `c` was created inside the operation.

write your answer here...

This violates Law of Demeter.

```
1 void foo(Bar b) {
2     Goo g =  new Goo();
3     g.doSomething();
4 }
```

- ○ True
- ○ False

False

Explanation: The line `g.doSomething()` does not violate LoD because it is OK to invoke methods of objects created within a method.

Pick the odd one out.

- ☐ a. Law of Demeter.
- ☐ b. Don’t add people to a late project.
- ☐ c. Don’t talk to strangers.
- ☐ d. Principle of least knowledge.
- ☐ e. Coupling.

(b)

Explanation: Law of Demeter, which aims to reduce coupling, is also known as ‘Don’t talk to strangers’ and ‘Principle of least knowledge’.

W9.3f ★★★★: OPTIONAL

Principles → Interface Segregation Principle

W9.3g ★★★★: OPTIONAL

Principles → Dependency Inversion Principle

W9.3h ★★★

Principles → SOLID Principles

Can explain SOLID Principles

The five OOP principles given below are known as *SOLID Principles* (an acronym made up of the first letter of each principle):

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

W9.3i ★★★★: OPTIONAL

Principles → YAGNI Principle

W9.3j ★★★★: OPTIONAL

Principles → DRY Principle

W9.3k ★★★★: OPTIONAL

Principles → Brooks' Law

[W9.4] Conceptualizing a Design ★★

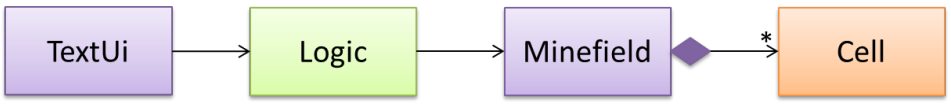
W9.4a ★★

Design → Modeling → Modeling a Solution → Introduction

Can explain how modelling can be used before implementation

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

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



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

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

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

W9.4b ★★★

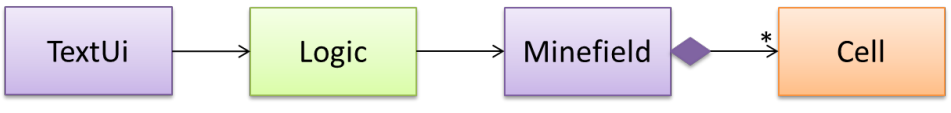
Design → Modeling → Modeling a Solution → Basic

Design → Modeling → Modeling Behaviors → Sequence Diagrams → Basic

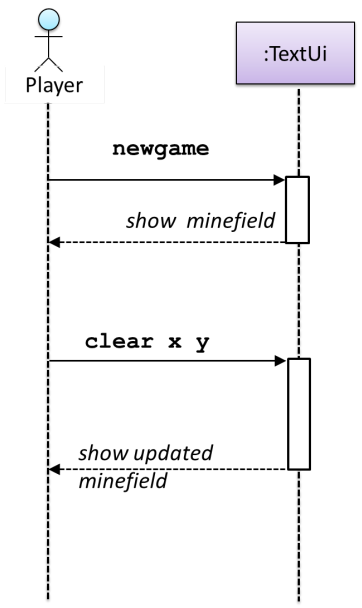
Design → Modeling → Modeling Structures → Class Diagrams → Basic

Can use simple class diagrams and sequence diagrams to model an OO solution

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

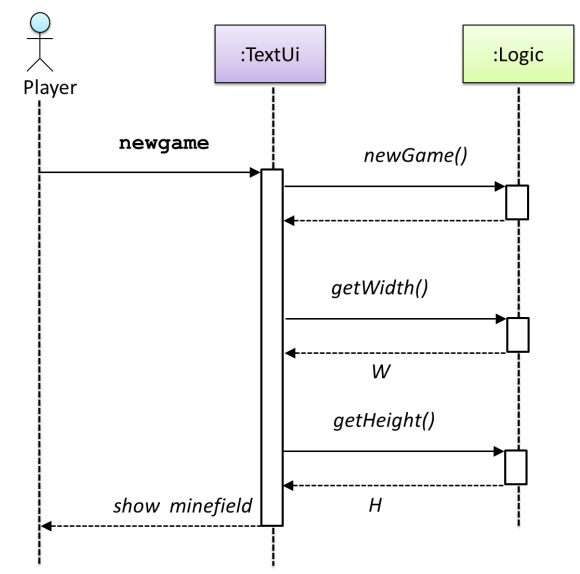


Let us start by modelling a sample interaction between the person playing the game and the `TextUi` object.



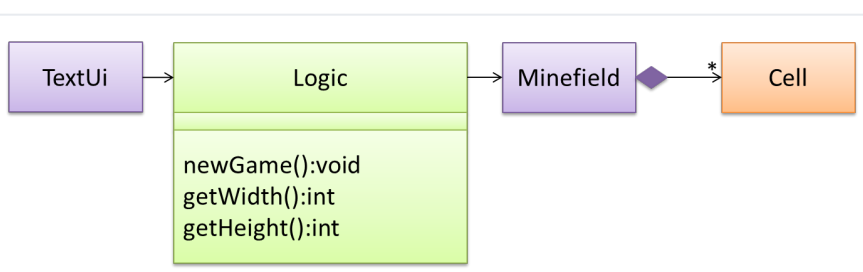
`newgame` and `clear x y` represent commands typed by the `Player` on the `TextUi`.

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

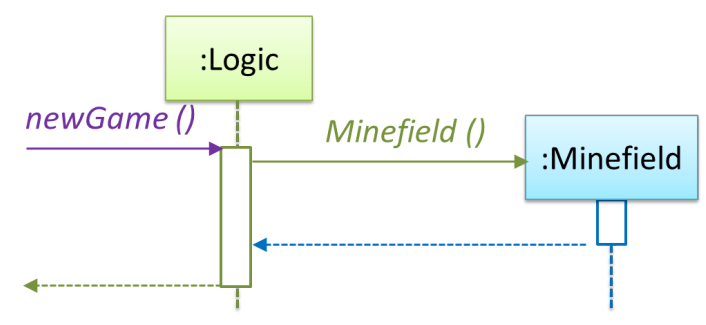


`W` = Width of the minefield; `H` = Height of the minefield

The above diagram assumes that `W` and `H` are the only information `TextUi` requires to display the minefield to the `Player`. Note that there could be other ways of doing this.
The `Logic` methods we conceptualized in our modelling so far are:

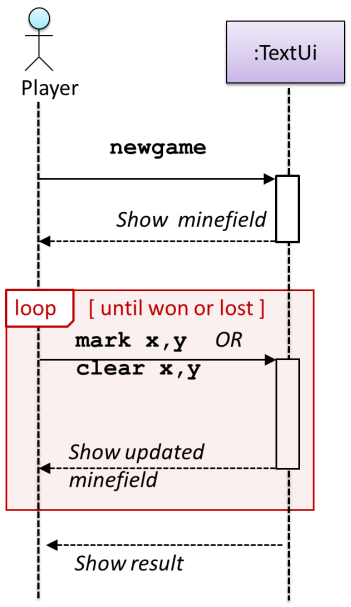


Now, let us look at what other objects and interactions are needed to support the `newGame()` operation. It is likely that a new `Minefield` object is created when the `newGame()` method is called.



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

Given below are the interactions between the player and the Text UI for the whole game.



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

Defining the architecture-level APIs for a small Tic-Tac-Toe game:

Game of Tic-Tac-Toe: A worked example of designing architecture an...

W9.4c ★★★★: OPTIONAL

Design → Modeling → Modeling a Solution → Intermediate

[W9.5] SDLC Process Models ★★

W9.5a ★★

Project Management → SDLC Process Models → Introduction → What

Can explain SDLC process models

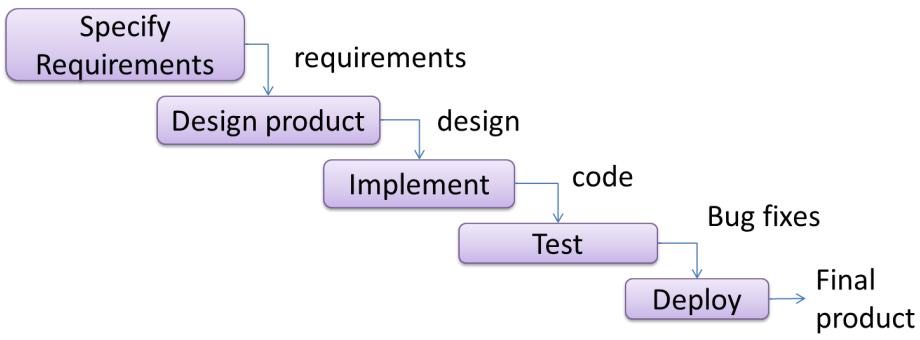
Software development goes through different stages such as *requirements, analysis, design, implementation* and *testing*. These stages are collectively known as the *software development life cycle* (SDLC). There are several approaches, known as *software development life cycle models* (also called *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.

W9.5b ★★

Project Management → SDLC Process Models → Introduction → Sequential Models

Can explain sequential process 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 requirement 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 that 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 a relative ease.

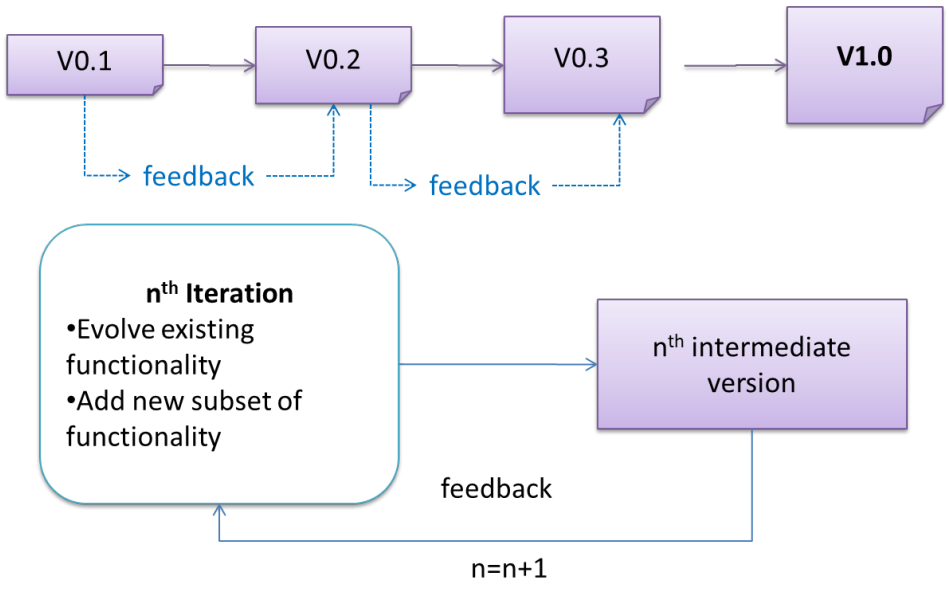
The major problem with this model is that requirements of a real-world project are rarely well-understood at the beginning and keep changing over time. One reason for this is that users are generally not aware of how a software application can be used without prior experience in using a similar application.

W9.5c ★★

Project Management → SDLC Process Models → Introduction → Iterative Models

Can explain iterative process 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 requirement 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 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.
- **depth-first**: an iteration focuses on fleshing out only some components.

Most project use a mixture of breadth-first and depth-first iterations. Hence, the common phrase 'an iterative and incremental process'.

W9.5d ★★★

Project Management → SDLC Process Models → Introduction → Agile Models

Can explain agile process 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).

> We are uncovering better ways of developing software by doing it and helping others do it.
>
> Through this work we have come to value:
>
> - **Individuals and interactions** over processes and tools
> - **Working software** over comprehensive documentation
> - **Customer collaboration** over contract negotiation
> - **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we 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.
- 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.

Choose the correct statements about agile processes.

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

(a)(b)(c)(d)

W9.5e ★★★

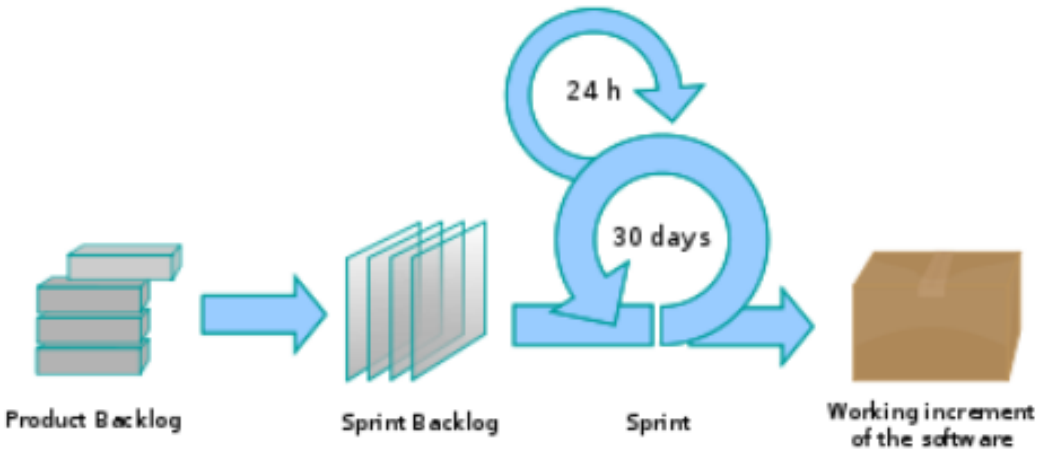
Project Management → SDLC Process Models → Scrum

🏆 Can explain scrum

This description of Scrum was adapted from Wikipedia [retrieved on 18/10/2011], emphasis added:

> **Scrum is a process skeleton that contains sets of practices and predefined roles. The main roles in Scrum are:**
>
> - **The Scrum Master**, who maintains the processes (typically in lieu of a project manager)
> - **The Product Owner**, who represents the stakeholders and the business
> - **The Team**, a cross-functional group who do the actual analysis, design, implementation, testing, etc.
>
> **A Scrum project is divided into iterations called Sprints.** A sprint is the basic unit of development in Scrum. Sprints tend to last between one week and one month, and are a timeboxed (i.e. restricted to a specific duration) effort of a constant length.
>
> **Each sprint is preceded by a planning meeting**, where the tasks for the sprint are identified and an estimated commitment for the sprint goal is made, and followed by a review or retrospective meeting, where the progress is reviewed and lessons for the next sprint are identified.
>
> **During each sprint, the team creates a potentially deliverable product increment** (for example, working and tested software). The set of features that go into a sprint come from the product backlog, which is a prioritized set of high level requirements of work to be done. Which backlog items go into the sprint is determined during the sprint planning meeting. During this meeting, the Product Owner informs the team of the items in the product backlog that he or she wants completed. The team then determines how much of this they can commit to complete during the next sprint, and records this in the sprint backlog. During a sprint, no one is allowed to change the sprint backlog, which means that the requirements are frozen for that sprint. Development is timeboxed such that the sprint must end on time; if requirements are not completed for any reason they are left out and returned to the product backlog. After a sprint is completed, the team demonstrates the use of the software.
>
> **Scrum enables the creation of self-organizing teams by encouraging co-location of all team members**, and verbal communication between all team members and disciplines in the project.
>
> **A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need** (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner.

As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.



***Daily Scrum* is another key scrum practice**. The description below was adapted from https://www.mountaingoatsoftware.com (emphasis added):

> **In Scrum, on each day of a sprint, the team holds a daily scrum meeting called the "daily scrum."** Meetings are typically held in the same location and at the same time each day. Ideally, a daily scrum meeting is held in the morning, as it helps set the context for the coming day's work. These scrum meetings are strictly time-boxed to 15 minutes. This keeps the discussion brisk but relevant.
>
> ...
>
> **During the daily scrum, each team member answers the following three questions:**
>
> - What did you do yesterday?
> - What will you do today?
> - Are there any impediments in your way?
>
> ...
>
> The daily scrum meeting is not used as a problem-solving or issue resolution meeting. **Issues that are raised are taken offline and usually dealt with by the relevant subgroup immediately after the meeting.**

▶ 🎥 Intro to Scrum in Under 10 Minutes

▼ W9.5f ★★★

Project Management → SDLC Process Models → XP

🏆 Can explain XP

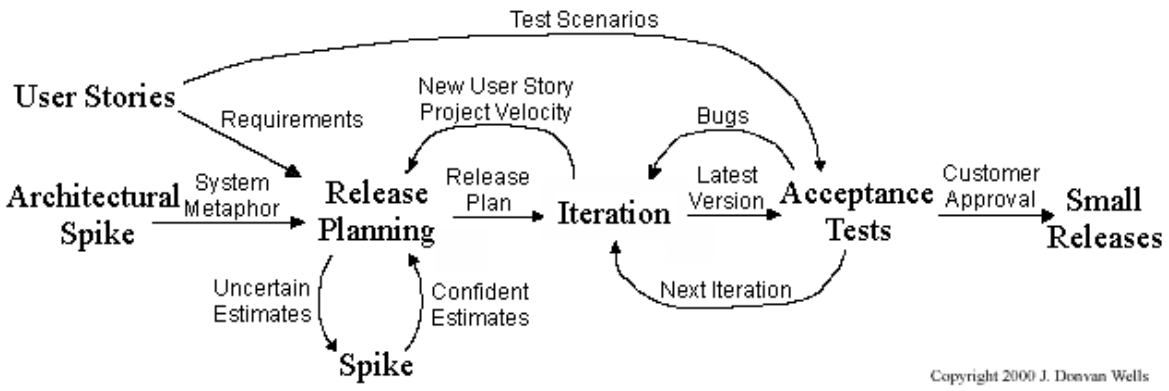
The following description was adapted from the XP home page, emphasis added:

> **Extreme Programming (XP) stresses customer satisfaction.** Instead of delivering everything you could possibly want on some date far in the future, this process delivers the software you need as you need it.
>
> **XP aims to empower developers to confidently respond to changing customer requirements**, even late in the life cycle.

XP emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. XP implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

XP aims to improve a software project in five essential ways: communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. Every small success deepens their respect for the unique contributions of each and every team member. With this foundation, Extreme Programmers are able to courageously respond to changing requirements and technology.

XP has a set of simple rules. XP is a lot like a jig saw puzzle with many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. This flow chart shows how Extreme Programming's rules work together.



***Pair programming, CRC cards, project velocity*, and *standup meetings* are some interesting topics related to XP.** Refer to extremeprogramming.org to find out more about XP.

W9.5g ★★★★: OPTIONAL

Project Management → SDLC Process Models → Unified Process

W9.5h ★★★★: OPTIONAL

Project Management → SDLC Process Models → CMMI

W9.5i ★★★

Project Management → SDLC Process Models → Recap

🏆 Can explain process models at a higher level

This section has some exercise that cover multiple topics related to SDLC process models.

Discuss how sequential approach and the iterative approach can affect the following aspects of a project.

a) Quality of the final product.

b) Risk of overshooting the deadline.

c) Total project cost.

d) Customer satisfaction.

e) Monitoring the project progress.

f) Suitability for a school project

a) Quality of the final product:

- Iterative: Frequent reworking can deteriorate the design. Frequent refactoring should be used to prevent this. Frequent customer feedback can help to improve the quality (i.e. quality as seen by the customer).
- Sequential: Final quality depends on the quality of each phase. Any quality problem in any phase could result in a low quality product.

b) Risk of overshooting the deadline.

- Iterative: Less risk. If the last iteration got delayed, we can always deliver the previous version. However, this does not guarantee that all features promised at the beginning will be delivered on the deadline.
- Sequential: High risk. Any delay in any phase can result in overshooting the deadline with nothing to deliver.

c) Total project cost.

- Iterative: We can always stop before the project budget is exceeded. However, this does not guarantee that all features promised at the beginning will be delivered under the estimated cost. (The sequential model requires us to carry on even if the budget is exceeded because there is no intermediate version to fall back on).

Iterative reworking of existing artifacts could add to the cost. However, this is “cheaper” than finding at the end that we built the wrong product.

d) Customer satisfaction

- Iterative: Customer gets many opportunities to guide the product in the direction he wants. Customer gets to change requirements even in the middle of the product. Both these can increase the probability of customer satisfaction.
- Sequential: Customer satisfaction is guaranteed only if the product was delivered as promised and if the initial requirements proved to be accurate. However, the customer is not required to do the extra work of giving frequent feedback during the project.

e) Monitoring project progress

- Iterative: Hard to measure progress against a plan, as the plan itself keeps changing.
- Sequential: Easier to measure progress against the plan, although this does not ensure eventual success.

f) Suitability for a school project:

Reasons to use iterative:

- Requirements are not fixed.
- Overshooting the deadline is not an option.
- Gives a chance to learn lessons from one iteration and apply them in the next.

Sequential:

- Can save time because we minimize rework.

write your answer here...

Find out more about the following three topics and give at least three arguments for and three arguments against each.

(a) Agile processes

(b) Pair programming

(c) Test-driven development

(a) Arguments in favor of agile processes:

- More focus on customer satisfaction.
- Less chance of building the wrong product (because of frequent customer feedback).
- Less resource wasted on bureaucracy, over-documenting, contract negotiations.

Arguments against agile processes (not necessarily true):

- It is ‘just hacking’. Not very systematic. No discipline.
- It is hard to know in advance the exact final product.
- It does not give enough attention to documentation.
- Lack of management control (gives too much freedom to developers)

(b) Arguments in favor of pair programming:

- It could produce better quality code.
- It is good to have more than one person know about any piece of code.
- It is a way to learn from each other.
- It can be used to train new programmers.
- Better discipline and better time management (e.g. less likely to play Farmville while working).
- Better morale due to more interactions with co-workers.

Arguments against pair programming:

- Increase in total man hours required
- Personality clashes between pair-members
- Workspaces need to be adapted to suit two developers working at one computer.
- If pairs are rotated, one needs to know more parts of the system than in solo programming

(c) Arguments in favor of TDD:

- Testing will not be neglected due to time pressure (because it is done first).
- Forces the developer to think about what the component should be before jumping into implementing it.
- Optimizes programmer effort (i.e. if all tests pass, there is no need to add any more functionality).
- Forces us to automate all tests.

Arguments against TDD (not necessarily true):

- Since tests can be seen as ‘executable specifications’, programmers tend to neglect others forms of documentation.
- Promotes ‘trial-and-error’ coding instead of making programmers think through their algorithms (i.e. ‘just keep hacking until all tests pass’).
- Gives a false sense of security. (what if you forgot to test certain scenarios?)

Not intuitive. Some programmer might resist adopting TDD.

write your answer here...

The sequential model and the waterfall model are the two most basic process models.

- True
- False

False

Explanation: The sequential model and the waterfall model are the same thing. The second basic model is the iterative model.

Choose the correct statements about the sequential and iterative process models.

- a. The sequential model organizes the project based on activities.
- b. The iterative and incremental model organizes the project based on functionality.
- c. The iterative model can be breadth-first or depth-first.
- d. The iterative model is always better than the sequential model.
- e. Compared to the sequential model, the iterative model is better at adapting to changing requirements.

(a)(b)(c)~~(d)~~(e)

Explanation: Both models have pros and cons. There is no definitive ‘better’ choice between the two. However, the iterative model works better in typical software projects than a purely sequential approach.

In general, which has a higher risk of overshooting a deadline?

- a. The iterative process.
- b. The sequential process.
- c. The risk is similar.

(b)

Explanation: An iterative process can meet a deadline better than a sequential process. If the last iteration got delayed, we can always deliver the previous version. However, this does not guarantee that all features promised at the beginning will be delivered on the deadline.

[W9.6] Writing Developer Documents ★★

Type of Developer Docs

W9.6a ★★ Implementation → Documentation → Introduction → What

🏆 Can explain the two types of developer docs

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

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

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

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

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

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

There is a secret that needs to be understood in order to write good software documentation: there isn't one thing called documentation, there are four. They are: tutorials, how-to guides, explanation and technical reference. They represent four different purposes or functions, and require four different approaches to their creation. Understanding the implications of this will help improve most software documentation - often immensely. ...

TUTORIALS A tutorial: • is learning-oriented • allows the newcomer to get started • is a lesson Analogy: teaching a small child how to cook	**HOW-TO GUIDES** A how-to guide: • is goal-oriented • shows how to solve a specific pr • is a series of steps Analogy: a recipe in a cookery book
EXPLANATION An explanation: • is understanding-oriented • explains • provides background and context Analogy: an article on culinary social history	**REFERENCE** A reference guide: • is information-oriented • describes the machinery • is accurate and complete Analogy: a reference encyclopedia art

Software documentation (applies to both user-facing and developer-facing) is best kept in a text format, for the ease of version tracking. **A writer friendly source format is also desirable** as non-programmers (e.g., technical writers) may need to author/edit such documents. As a result, formats such as Markdown, Asciidoc, and PlantUML are often used for software documentation.

Choose correct statements about API documentation.

- ☐ a. They are useful for both developers who use the API and developers who maintain the API implementation.
- ☐ b. There are tools that can generate API documents from code comments.
- ☐ c. API documentation may contain code examples.

All

Guideline: Aim for Comprehensibility

W9.6b ★★

Implementation → Documentation → Guidelines → Aim for Comprehensibility → What

🏆 **Can explain the need for comprehensibility in documents**

Technical documents exist to help others understand technical details. Therefore, **it is not enough for the documentation to be accurate and comprehensive, it should also be comprehensible too**.

W9.6c ★★★

Implementation → Documentation → Guidelines → Aim for Comprehensibility → How

Can write reasonably comprehensible developer documents

Here are some tips on writing effective documentation.

- **Use plenty of diagrams**: It is not enough to explain something in words; complement it with visual illustrations (e.g. a UML diagram).
- **Use plenty of examples**: When explaining algorithms, show a running example to illustrate each step of the algorithm, in parallel to worded explanations.
- **Use simple and direct explanations**: Convoluted explanations and fancy words will annoy readers. Avoid long sentences.
- **Get rid of statements that do not add value**: For example, 'We made sure our system works perfectly' (who didn't?), 'Component X has its own responsibilities' (of course it has!).
- **It is not a good idea to have separate sections for each type of artifact**, such as 'use cases', 'sequence diagrams', 'activity diagrams', etc. Such a structure, coupled with the indiscriminate inclusion of diagrams without justifying their need, indicates a failure to understand the purpose of documentation. Include diagrams when they are needed to explain something. If you want to provide additional diagrams for completeness' sake, include them in the appendix as a reference.

It is recommended for developer documents,

- ☐ a. to have separate sections for each type of diagrams such as class diagrams, sequence diagrams, use case diagrams etc.
- ☐ b. to give a high priority to comprehension too, not stop at comprehensiveness only.

~~(a)~~(b)

Explanation:

(a) Use diagrams when they help to understand the text descriptions. Text and diagrams should be used in tandem. Having separate sections for each diagram type is a sign of generating diagrams for the sake of having them.

(b) Both are important, but lengthy, complete, accurate yet hard to understand documents are not that useful.

Guideline: Describe Top-Down

W9.6d ★★

Implementation → Documentation → Guidelines → Describe Top-Down → What

Can distinguish between top-down and bottom up documentation

When writing project documents, a top-down breadth-first explanation is easier to understand than a bottom-up one.

W9.6e ★★★

Implementation → Documentation → Guidelines → Describe Top-Down → Why

Can explain the advantages of top-down documentation

The main advantage of the top-down approach is that the document is structured like an upside down tree (root at the top) and **the reader can travel down a path she is interested in until she reaches the component she is interested to learn in-depth**, without having to read the entire document or understand the whole system.

W9.6f ★★

Implementation → Documentation → Guidelines → Describe Top-Down → How

Can write documentation in a top-down manner

To explain a system called `SystemFoo` with two sub-systems, `FrontEnd` and `BackEnd`, start by describing the system at the highest level of abstraction, and progressively drill down to lower level details. An outline for such a description is given below.

[First, explain what the system is, in a black-box fashion (no internal details, only the external view).]

> `SystemFoo` is a ....

[Next, explain the high-level architecture of `SystemFoo`, referring to its major components only.]

> `SystemFoo` consists of two major components: `FrontEnd` and `BackEnd`.
>
> The job of `FrontEnd` is to ... while the job of `BackEnd` is to ...
>
> And this is how `FrontEnd` and `BackEnd` work together ...

[Now we can drill down to `FrontEnd`'s details.]

> `FrontEnd` consists of three major components: `A`, `B`, `C`
>
> `A`'s job is to ...
> `B`'s job is to...
> `C`'s job is to...
>
> And this is how the three components work together ...

[At this point, further drill down the internal workings of each component. A reader who is not interested in knowing nitty-gritty details can skip ahead to the section on `BackEnd`.]

> In-depth description of `A`
>
> In-depth description of `B`
>
> ...

[At this point drill down details of the `BackEnd`.]

> ...

Guideline: Minimal but Sufficient

W9.6g ★★

Implementation → Documentation → Guidelines → Minimal but Sufficient → What

🏆 Can explain documentation should be minimal yet sufficient

Aim for 'just enough' developer documentation.

- Writing and maintaining developer documents is an overhead. You should try to minimize that overhead.
- If the readers are developers who will eventually read the code, the documentation should complement the code and should provide only just enough guidance to get started.

W9.6h ★★★

Implementation → Documentation → Guidelines → Minimal but Sufficient → How

🏆 Can write minimal yet sufficient documentation

Anything that is already clear in the code need not be described in words. Instead, **focus on providing higher level information that is not readily visible in the code or comments**.

Refrain from duplicating chunks or text. When describing several similar algorithms/designs/APIs, etc., do not simply duplicate large chunks of text. Instead, **describe the similarity in one place and emphasize only the differences in other places.** It is very annoying to see pages and pages of similar text without any indication as to how they differ from each other.

Week 10 [Mar 23] - Topics

Detailed Table of Contents

[W10.1] Design Patterns ★★

Introduction

W10.1a ★★

Design → Design Patterns → Introduction → What

Can explain design patterns

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

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

Some examples of recurring design problems:

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

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

Which one of these describes the ‘software design patterns’ concept best?

- a. Designs that appear repetitively in software.
- b. Elegant solutions to recurring problems in software design.
- c. Architectural styles used in applications.
- d. Some good design techniques proposed by the Gang of Four

(b)

W10.1b ★★

Design → Design Patterns → Introduction → Format

Can explain design patterns format

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

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

When we describe a pattern, we must also specify anti-patterns.

- True
- False

False.

Explanation: Anti-patterns are related to patterns, but they are not a 'must have' component of a pattern description.

Singleton pattern

W10.1c ★★

Design → Design Patterns → Singleton → What

Can explain the Singleton design pattern

Context

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

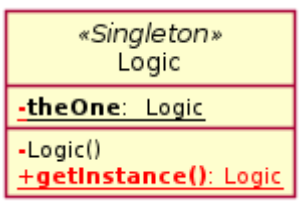
Problem

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

Solution

Make the constructor of the singleton class `private`, because a `public` constructor will allow others to instantiate the class at will. Provide a `public` class-level method to access the *single instance*.

Example:



We use the Singleton pattern when

- a. we want an a class with a private constructor.
- b. we want a single class to hold all functionality of the system.
- c. we want a class with no more than one instance.
- d. we want to hide internal structure of a component from its clients.

(c)

W10.1d ★★

Design → Design Patterns → Singleton → Implementation

Can apply the Singleton design pattern

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

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

    private Logic() {
        ...
    }

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

Notes:

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

If `Logic` was not a Singleton class, an object is created like this:

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

But now, the `Logic` object needs to be accessed like this:

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

W10.1e ★★★

Design → Design Patterns → Singleton → Evaluation

Can decide when to apply Singleton design pattern

Pros:

- easy to apply
- effective in achieving its goal with minimal extra work
- provides an easy way to access the singleton object from anywhere in the code base

Cons:

- The singleton object acts like a global variable that increases coupling across the code base.
- In testing, it is difficult to replace Singleton objects with stubs (static methods cannot be overridden)
- In testing, singleton objects carry data from one test to another even when we want each test to be independent of the others.

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

Facade pattern

W10.1f ★★★

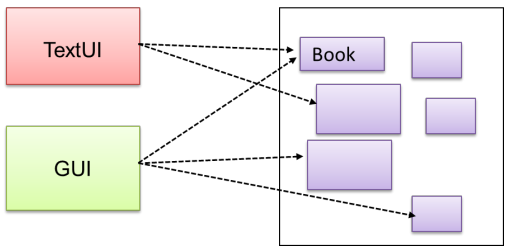
Design → Design Patterns → Facade Pattern → What

🏆 Can explain the Facade design pattern

Context

Components need to access functionality deep inside other components.

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



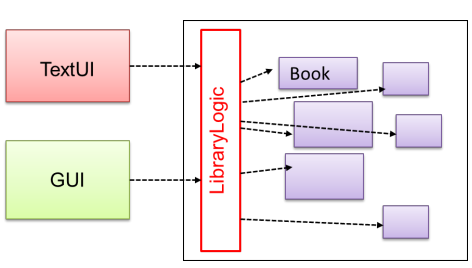
Problem

Access to the component should be allowed without exposing its internal details. e.g. the `UI` component should access the functionality of the `Logic` component without knowing that it contained a `Book` class within it.

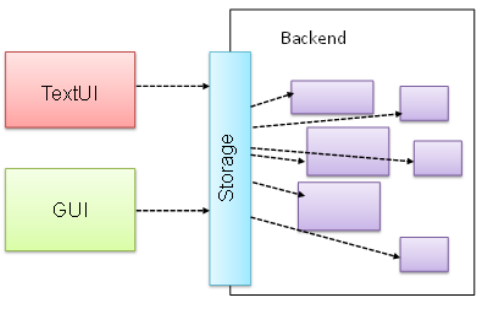
Solution

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

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



Does the design below likely to use the Facade pattern?



- True
- False

True.

Facade is clearly visible (Storage is the <> class).

Command pattern

W10.1g ★★★

Design → Design Patterns → Command Pattern → What

🏆 Can explain the Command design pattern

Context

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

Problem

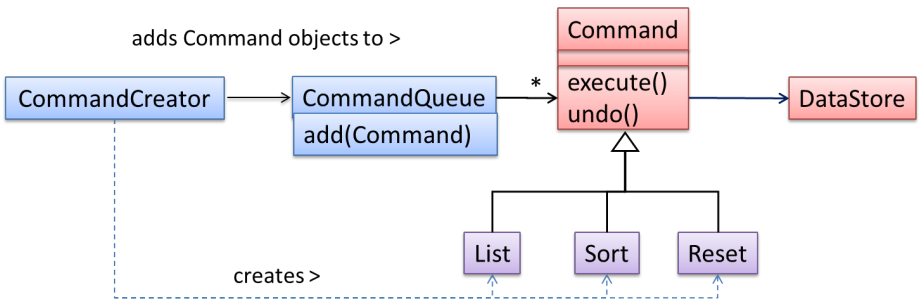
It is preferable that some part of the code executes these commands without having to know each command type. e.g., there can be a `CommandQueue` object that is responsible for queuing commands and executing them without knowledge of what each command does.

Solution

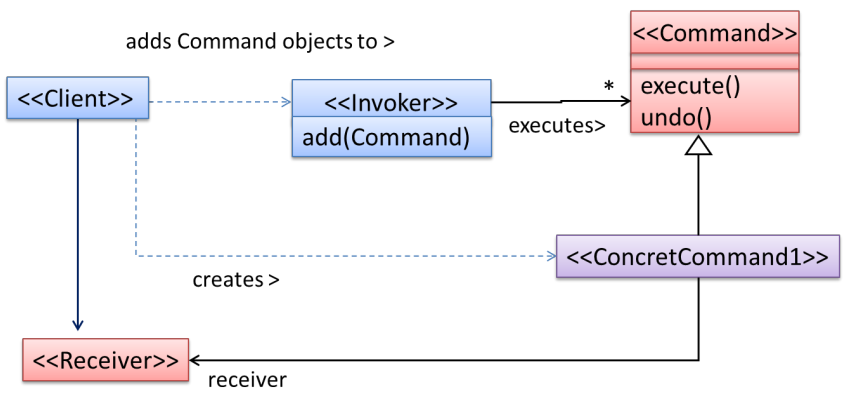
The essential element of this pattern is to have a general `<<Command>>` object that can be passed around, stored, executed, etc without knowing the type of command (i.e. via polymorphism).

Let us examine an example application of the pattern first:

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



The general form of the solution is as follows.



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

Abstraction Occurrence pattern

W10.1h ★★★★: OPTIONAL

Design → Design Patterns → Abstraction Occurrence Pattern → What

[W10.2] Defensive Programming ★★

W10.2a ★★

Implementation → Error Handling → Defensive Programming → What

Can explain defensive programming

A defensive programmer codes under the assumption "if we leave room for things to go wrong, they *will* go wrong". Therefore, a defensive programmer proactively tries to eliminate any room for things to go wrong.

Consider a `MainApp#getConfig()` a method that returns a `Config` object containing configuration data. A typical implementation is given below:

```
class MainApp{
    Config config;

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

If the returned Config object is not meant to be modified, a defensive programmer might use a more *defensive* implementation given below. This is more defensive because even if the returned `Config` object is modified (although it is not meant to be) it will not affect the `config` object inside the `MainApp` object.

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

W10.2b ★★★

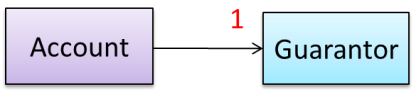
Implementation → Error Handling → Defensive Programming → Enforcing Compulsory Associations

Implementation → Error Handling → Defensive Programming → What

Can use defensive coding to enforce compulsory associations

Consider two classes, `Account` and `Guarantor`, with an association as shown in the following diagram:

Example:



Here, the association is compulsory i.e. an `Account` object should always be linked to a `Guarantor`. One way to implement this is to simply use a reference variable, like this:

```
1  class Account {
2      Guarantor guarantor;
3
4      void setGuarantor(Guarantor g) {
5          guarantor = g;
6      }
7  }
```

However, what if someone else used the `Account` class like this?

```
1  Account a = new Account();
2  a.setGuarantor(null);
```

This results in an `Account` without a `Guarantor`! In a real banking system, this could have serious consequences! The code here did not try to prevent such a thing from happening. We can make the code more defensive by proactively enforcing the multiplicity constraint, like this:

```
1  class Account {
2      private Guarantor guarantor;
3
4      public Account(Guarantor g){
5          if (g == null) {
6                          stopSystemWithMessage("multiplicity  violated.  Null
   Guarantor");
7          }
8          guarantor = g;
9      }
10     public void setGuarantor (Guarantor g){
11         if (g == null) {
12                         stopSystemWithMessage("multiplicity  violated.  Null
   Guarantor");
13         }
14         guarantor = g;
15     }
16     …
17 }
```

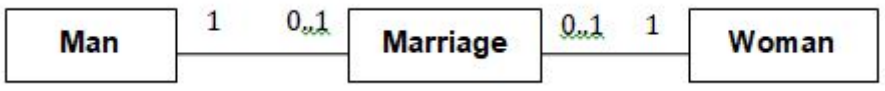
For the `Manager` class shown below, write an `addAccount()` method that

- restricts the maximum number of Accounts to 8
- avoids adding duplicate Accounts

```java
import java.util.*;

public class Manager {
    private ArrayList< Account > theAccounts ;

    public void addAccount(Account acc) throws Exception {
        if (theAccounts.size( ) == 8){
            throw new Exception ("adding more than 8 accounts");
        }

        if (!theAccounts.contains(acc)) {
            theAccounts.add(acc);
        }
    }

    public void removeAccount(Account acc) {
        theAccounts.remove(acc);
    }
}
```

write your answer here...

Implement the classes defensively with appropriate references and operations to establish the association among the classes. Follow the defensive coding approach. Let the `Marriage` class handle setting/removal of reference.



```java
public class Marriage {
    private Man husband = null;
    private Woman wife = null;

    // extra information like date etc can be added

    public Marriage(Man m, Woman w) throws Exception {
        if (m == null || w == null) {
            throw new Exception("no man/woman");
        }
        if (m.isMarried() || w.isMarried()) {
            throw new Exception("already married");
        }
        husband = m;
        m.enterMarriage(this);
        wife = w;
        w.enterMarriage(this);
    }

    public Man getHusband() throws Exception {
        if(husband == null) {
            throw new Exception("error state");
        } else {
            return husband;
        }
    }

    public Woman getWife() throws Exception {
        if(wife == null) {
            throw new Exception("error state");
        } else {
            return wife;
        }
    }

    // removal of both ends of 'Marriage'
    public void divorce() throws Exception {
        if (husband==null || wife==null) {
            throw new Exception("no marriage");
        }
        husband.removeFromMarriage(this);
        husband = null;
        wife.removeFromMarriage(this);
        wife = null;
    }
}
```

write your answer here...

W10.2c ★★★★: OPTIONAL

Implementation → Error Handling → Defensive Programming → Enforcing 1-to-1 Associations

W10.2d ★★★★: OPTIONAL

Implementation → Error Handling → Defensive Programming → Enforcing Referential Integrity

W10.2e ★★★

Implementation → Error Handling → Defensive Programming → When

Implementation → Error Handling → Defensive Programming → What

🏆 **Can explain when to use defensive programming**

It is not necessary to be 100% defensive all the time. While defensive code may be less prone to be misused or abused, such code can also be more complicated and slower to run.

The suitable degree of defensiveness depends on many factors such as:

- How critical is the system?
- Will the code be used by programmers other than the author?
- The level of programming language support for defensive programming
- The overhead of being defensive

Defensive programming,

- ☐ a. can make the program slower.
- ☐ b. can make the code longer.
- ☐ c. can make the code more complex.
- ☐ d. can make the code less susceptible to misuse.
- ☐ e. can require extra effort.

(a)(b)(c)(d)(e)

Explanation: Defensive programming requires a more checks, possibly making the code longer, more complex, and possibly slower. Use it only when benefits outweigh costs, which is often.

W10.2f ★★★★: OPTIONAL

Implementation → Error Handling → Design by Contract → Design by Contract

[W10.3] Test Cases: Intro ★★

W10.3a ★★★

Quality Assurance → Test Case Design → Introduction → What

🏆 **Can explain the need for deliberate test case design**

Except for trivial SUTs, exhaustive testing is not practical because such testing often requires a massive/infinite number of test cases.

📦 Consider the test cases for adding a string object to a collection:

- Add an item to an empty collection.
- Add an item when there is one item in the collection.
- Add an item when there are 2, 3, ... n items in the collection.
- Add an item that has an English, a French, a Spanish, ... word.
- Add an item that is the same as an existing item.
- Add an item immediately after adding another item.
- Add an item immediately after system startup.
- ...

Exhaustive testing of this operation can take many more test cases.

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> --Edsger Dijkstra

Every test case adds to the cost of testing. In some systems, a single test case can cost thousands of dollars e.g. on-field testing of flight-control software. Therefore, **test cases need to be designed to make the best use of testing resources.** In particular:

- **Testing should be *effective*** i.e., it finds a high percentage of existing bugs e.g., a set of test cases that finds 60 defects is more effective than a set that finds only 30 defects in the same system.
- **Testing should be *efficient*** i.e., it has a high rate of success (bugs found/test cases) a set of 20 test cases that finds 8 defects is more efficient than another set of 40 test cases that finds the same 8 defects.

For testing to be E&E, each new test we add should be targeting a potential fault that is not already targeted by existing test cases. There are test case design techniques that can help us improve E&E of testing.

Given below is the sample output from a text-based program `TriangleDetector` ithat determines whether the three input numbers make up the three sides of a valid triangle. List test cases you would use to test this software. Two sample test cases are given below.

```
C:\> java TriangleDetector
Enter side 1: 34
Enter side 2: 34
Enter side 3: 32
Can this be a triangle?:  Yes
Enter side 1:
```

Sample test cases,

```
34,34,34: Yes
0, any valid, any valid: No
```

In addition to obvious test cases such as

- sum of two sides == third,
- sum of two sides < third ...

We may also devise some interesting test cases such as the ones depicted below.

Note that their applicability depends on the context in which the software is operating.

- Non-integer number, negative numbers, `0` , numbers formatted differently (e.g. `13F`), very large numbers (e.g. `MAX_INT`), numbers with many decimal places, empty string, ...
- Check many triangles one after the other (will the system run out of memory?)
- `Backspace` , `tab` , `CTRL+C` , ...
- Introduce a long delay between entering data (will the program be affected by, say the screensaver?), minimize and restore window during the operation, hibernate the system in the middle of a calculation, start with invalid inputs (the system may perform error handling differently for the very first test case), …
- Test on different locale.

The main point to note is how difficult it is to test exhaustively, even on a trivial system.

write your answer here...

Explain the why exhaustive testing is not practical using the example of testing `newGame()` operation in the `Logic` class of a Minesweeper game.

Consider this sequence of test cases:

- Test case 1. Start Minesweeper. Activate `newGame()` and see if it works.
- Test case 2. Start Minesweeper. Activate `newGame()` . Activate `newGame()` again and see if it works.
- Test case 3. Start Minesweeper. Activate `newGame()` three times consecutively and see if it works.
- …
- Test case 267. Start Minesweeper. Activate `newGame()` 267 times consecutively and see if it works.

Well, you get the idea. Exhaustive testing of `newGame()` is not practical.

write your answer here...

Improving efficiency and effectiveness of test case design can,

- [ ] a. improve the quality of the SUT.
- [ ] b. save money.
- [ ] c. save time spent on test execution.
- [ ] d. save effort on writing and maintaining tests.
- [ ] e. minimize redundant test cases.
- [ ] f. forces us to understand the SUT better.

(a)(b)(c)(d)(e)(f)

W10.3b ★★

Quality Assurance → Test Case Design → Introduction → Positive vs Negative Test Cases

Can explain positive and negative test cases

A *positive test case* is when the test is designed to produce an expected/valid behavior. A negative test case is designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message.

Consider testing of the method `print(Integer i)` which prints the value of `i`.

- A positive test case: `i == new Integer(50)`
- A negative test case: `i == null;`

W10.3c ★★

Quality Assurance → Test Case Design → Introduction → Black Box vs Glass Box

Can explain black box and glass box test case design

Test case design can be of three types, based on how much of SUT internal details are considered when designing test cases:

- ***Black-box* (aka *specification-based or responsibility-based*) approach**: test cases are designed exclusively based on the SUT's specified external behavior.
- ***White-box* (aka *glass-box or structured or implementation-based*) approach**: test cases are designed based on what is known about the SUT's implementation, i.e. the code.
- ***Gray-box* approach**: test case design uses *some* important information about the implementation. For example, if the implementation of a sort operation uses different algorithms to sort lists shorter than 1000 items and lists longer than 1000 items, more meaningful test cases can then be added to verify the correctness of both algorithms.

Black-box and white-box testing

W10.3d ★★★

Quality Assurance → Test Case Design → Testing Based on Use Cases

Can explain test case design for use case based testing

Use cases can be used for system testing and acceptance testing. For example, the main success scenario can be one test case while each variation (due to extensions) can form another test case. However, note that use cases do not specify the exact data entered into the system. Instead, it might say something like `user enters his personal data into the system`. Therefore, the tester has to choose data by considering equivalence partitions and boundary values. The combinations of these could result in one use case producing many test cases.

To increase E&E of testing, high-priority use cases are given more attention. For example, a scripted approach can be used to test high priority test cases, while an exploratory approach is used to test other areas of concern that could emerge during testing.

[W10.4] Test Cases: Equivalence Partitioning ★★

W10.4a ★★

Quality Assurance → Test Case Design → Equivalence Partitions → What

Can explain equivalence partitions

Consider the testing of the following operation.

`isValidMonth(m)` : returns `true` if `m` (and `int`) is in the range [1..12]

It is inefficient and impractical to test this method for all integer values `[-MIN_INT to MAX_INT]`. Fortunately, there is no need to test all possible input values. For example, if the input value `233` failed to produce the correct result, the input `234` is likely to fail too; there is no need to test both.

In general, **most SUTs do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways.** That means a range of inputs is treated the same way inside the SUT. ***Equivalence partitioning (EP)*** **is a test case design technique that uses the above observation to improve the E&E of testing.**

Equivalence partition (aka equivalence class): A group of test inputs that are likely to be processed by the SUT in the same way.

By dividing possible inputs into equivalence partitions we can,

- **avoid testing too many inputs from one partition.** Testing too many inputs from the same partition is unlikely to find new bugs. This increases the efficiency of testing by reducing redundant test cases.

- **ensure all partitions are tested.** Missing partitions can result in bugs going unnoticed. This increases the effectiveness of testing by increasing the chance of finding bugs.

W10.4b ★★

Quality Assurance → Test Case Design → Equivalence Partitions → Basic

Can apply EP for pure functions

Equivalence partitions (EPs) are usually derived from the specifications of the SUT.

> These could be EPs for the isValidMonth example:
>
> - [MIN_INT ... 0] : **below** the range that produces `true` (produces `false`)
> - [1 … 12] : the range that produces `true`
> - [13 … MAX_INT] : **above** the range that produces `true` (produces `false`)

When the SUT has multiple inputs, you should identify EPs for each input.

> Consider the method `duplicate(String s, int n): String` which returns a `String` that contains `s` repeated `n` times.
>
> Example EPs for `s` :
>
> - zero-length strings
> - string containing whitespaces
> - ...
>
> Example EPs for `n` :
>
> - `0`
> - negative values
> - ...

An EP may not have adjacent values.

> Consider the method `isPrime(int i): boolean` that returns true if `i` is a prime number.
>
> EPs for `i` :
>
> - prime numbers
> - non-prime numbers

Some inputs have only a small number of possible values and a potentially unique behavior for each value. In those cases we have to consider each value as a partition by itself.

> Consider the method `showStatusMessage(GameStatus s): String` that returns a unique `String` for each of the possible value of s (`GameStatus` is an `enum`). In this case, each possible value for `s` will have to be considered as a partition.

Note that the EP technique is merely a heuristic and not an exact science, especially when applied manually (as opposed to using an automated program analysis tool to derive EPs). The partitions derived depend on how one 'speculates' the SUT to behave internally. Applying EP under a glass-box or gray-box approach can yield more precise partitions.

> Consider the method EPs given above for the `isValidMonth` . A different tester might use these EPs instead:
>
> - [1 … 12] : the range that produces `true`
> - [all other integers] : the range that produces `false`

Some more examples:

Specification	Equivalence partitions
`isValidFlag(String s): boolean` Returns `true` if s is one of [`"F"` , `"T"` , `"D"`]. The comparison is case-sensitive.	[`"F"`] [`"T"`] [`"D"`] [`"f"` , `"t"` , `"d"`] [any other string][null]
`squareRoot(String s): int` Pre-conditions: s represents a positive integer Returns the square root of `s` if the square root is an integer; returns `0` otherwise.	[`s` is not a valid number] [`s` is a negative integer] [`s` has an integer square root] [`s` does not have an integer square root]

Consider this SUT:

```
isValidName (String s): boolean
```

Description: returns true if `s` is not `null` and not longer than 50 characters.

A. Which one of these is least likely to be an equivalence partition for the parameter s of the isValidName method given below?

- a. null.
- b. strings having more than 50 characters.
- c. strings having 50 or fewer characters.
- d. strings consisting of numbers instead of letters.

B. If you had to choose 3 test cases from the 4 given below, which one will you leave out based on the EP technique?

- a. A string that is 50 characters long
- b. A string that is 51 characters long
- c. A string that is 40 characters long
- d. null

A. (d)

Explanation: The description does not mention anything about the content of the string. Therefore, the method is unlikely to behave differently for strings consisting of numbers.

B. (a) or (c)

Explanation: both belong to the same EP

W10.4c ★★★

Quality Assurance → Test Case Design → Equivalence Partitions → [illegible]

Can apply EP for OOP methods

When deciding EPs of OOP methods, we need to identify EPs of all data participants that can potentially influence the behaviour of the method, such as,

- the target object of the method call
- input parameters of the method call
- other data/objects accessed by the method such as global variables. This category may not be applicable if using the black box approach (because the test case designer using the black box approach will not know how the method is implemented)

Consider this method in the `DataStack` class: `push(Object o): boolean`

- Adds o to the top of the stack if the stack is not full.
- returns `true` if the push operation was a success.
- throws
  - `MutabilityException` if the global flag `FREEZE==true` .
  - `InvalidValueException` if o is null.

EPs:

- `DataStack` object: [full] [not full]
- `o` : [null] [not null]
- `FREEZE` : [true][false]

Consider a simple Minesweeper app. What are the EPs for the `newGame()` method of the `Logic` component?

As `newGame()` does not have any parameters, the only obvious participant is the `Logic` object itself.

Note that if the glass-box or the grey-box approach is used, other associated objects that are involved in the method might also be included as participants. For example, `Minefield` object can be considered as another participant of the `newGame()` method. Here, the black-box approach is assumed.

Next, let us identify equivalence partitions for each participant. Will the `newGame()` method behave differently for different `Logic` objects? If yes, how will it differ? In this case, yes, it might behave differently based on the game state. Therefore, the equivalence partitions are:

- `PRE_GAME` : before the game starts, minefield does not exist yet
- `READY` : a new minefield has been created and waiting for player's first move
- `IN_PLAY` : the current minefield is already in use
- `WON` , `LOST` : let us assume the `newGame` behaves the same way for these two values

Consider the `Logic` component of the Minesweeper application. What are the EPs for the `markCellAt(int x, int y)` method?. The partitions in **bold** represent valid inputs.

- `Logic` : PRE_GAME, **READY**, **IN_PLAY**, WON, LOST
- `x` : [MIN_INT..-1] **[0..(W-1)]** [W..MAX_INT] (we assume a minefield size of WxH)
- `y` : [MIN_INT..-1] **[0..(H-1)]** [H..MAX_INT]
- `Cell` at `(x,y)` : **HIDDEN**, MARKED, CLEARED

[W10.5] Test Cases: Boundary Value Analysis

★★

W10.5a ★★

Quality Assurance → Test Case Design → Boundary Value Analysis → What

Can explain boundary value analysis

***Boundary Value Analysis (BVA)* is test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.** This is not surprising, as the end points of the boundary are often used in branching instructions etc. where the programmer can make mistakes.

`markCellAt(int x, int y)` operation could contain code such as if `(x > 0 && x <= (W-1))` which involves boundaries of x's equivalence partitions.

BVA suggests that when picking test inputs from an equivalence partition, values near boundaries (i.e. boundary values) are more likely to find bugs.

Boundary values are sometimes called *corner cases*.

Boundary value analysis recommends testing *only* values that reside on the equivalence class boundary.

- ○ True
- ○ False

False

Explanation: It does not recommend testing *only* those values *on* the boundary. It merely suggests that values on and around a boundary are more likely to cause errors.

W10.5b ★★★

Quality Assurance → Test Case Design → Boundary Value Analysis → How

Can apply boundary value analysis

Typically, we choose three values around the boundary to test: one value from the boundary, one value just below the boundary, and one value just above the boundary. The number of values to pick depends on other factors, such as the cost of each test case.

Some examples:

Equivalence partition	Some possible boundary values
[1-12]	0,1,2, 11,12,13

[MIN_INT, 0] (MIN_INT is the minimum possible integer value allowed by the environment)	MIN_INT, MIN_INT+1, -1, 0 , 1
[any non-null String]	Empty String, a String of maximum possible length
[prime numbers] [“F”] [“A”, “D”, “X”]	No specific boundary No specific boundary No specific boundary
[non-empty Stack] (we assume a fixed size stack)	Stack with: one element, two elements, no empty spaces, only one empty space

Week 11 [Mar 30] - Topics

Detailed Table of Contents

[W11.1] More Design Patterns ★★

W11.1a ★★
Design → Design Patterns → MVC Pattern → What

Can explain the Model View Controller (MVC) design pattern

Context

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

Problem

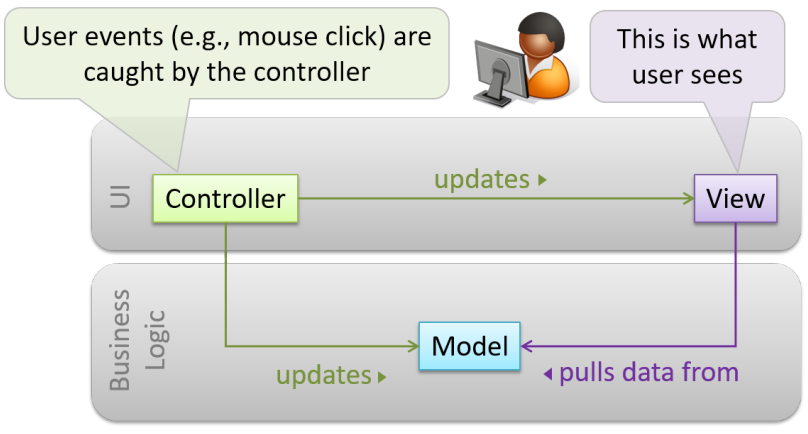
The high coupling that can result from the interlinked nature of the features described above.

Solution

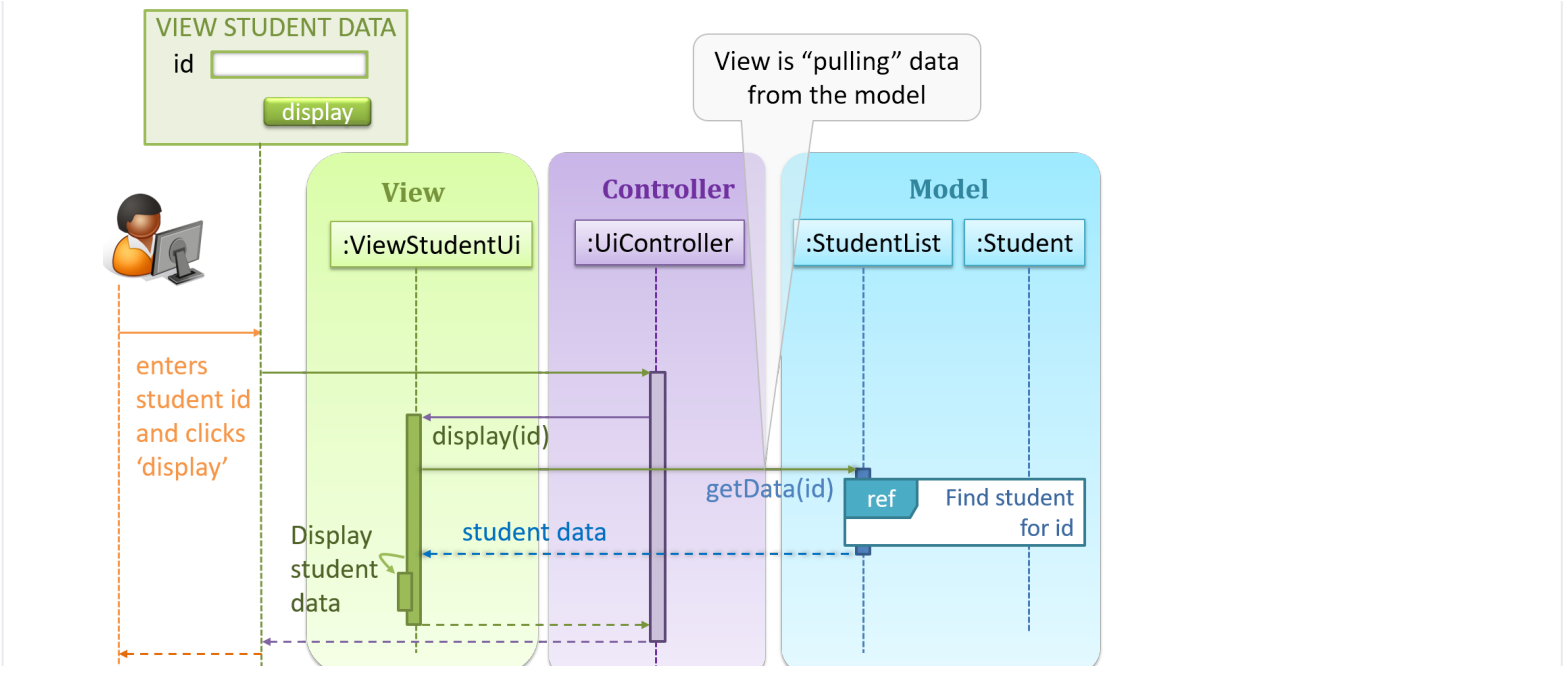
Decouple data, presentation, and control logic of an application by separating them into three different components: *Model*, *View* and *Controller*.

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

The relationship between the components can be observed in the diagram below. Typically, the UI is the combination of view and controller.



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



W11.1b ★★★

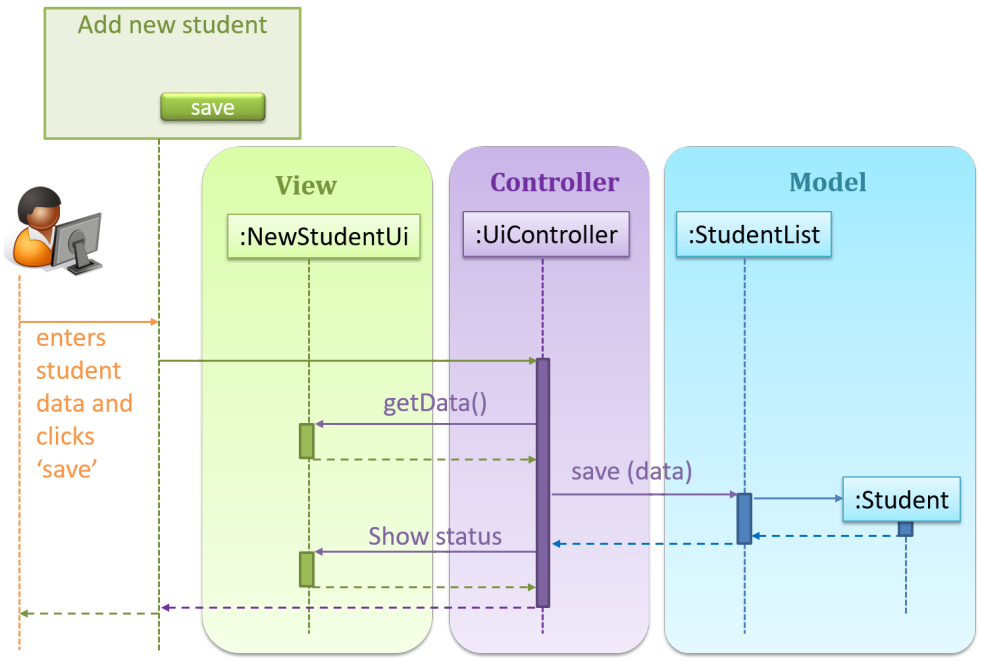
Design → Design Patterns → Observer Pattern → What

Can explain the Observer design pattern

Context

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

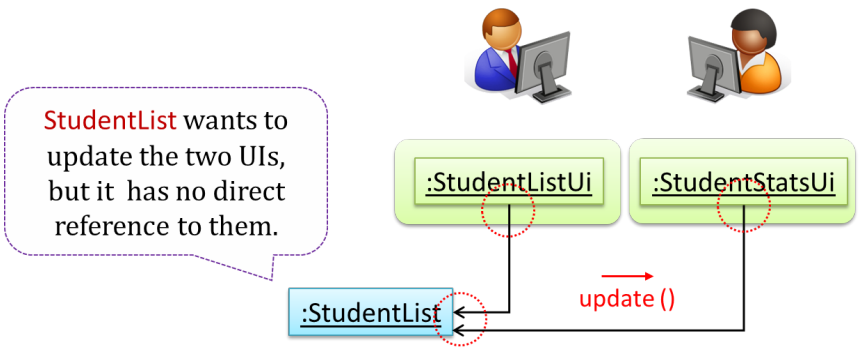
Consider this scenario from the a student management system where the user is adding a new student to the system.



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

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

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



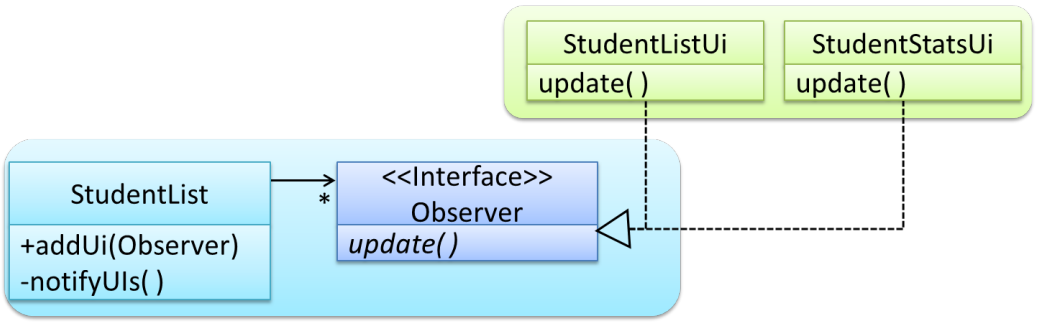
However, the `StudentList` object has no knowledge about `StudentListUi` and `StudentStatsUi` (note the direction of the navigability) and has no way to inform those objects. This is an example of the type of problem addressed by the Observer pattern.

Problem

The ‘observed’ object does not want to be coupled to objects that are ‘observing’ it.

Solution

Force the communication through an interface known to both parties.



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

During the initialization of the system,

1. First, create the relevant objects.

```
StudentList studentList = new StudentList();
StudentListUi listUi = new StudentListUi();
StudentStatusUi statusUi = new StudentStatsUi();
```

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

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

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

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

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

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

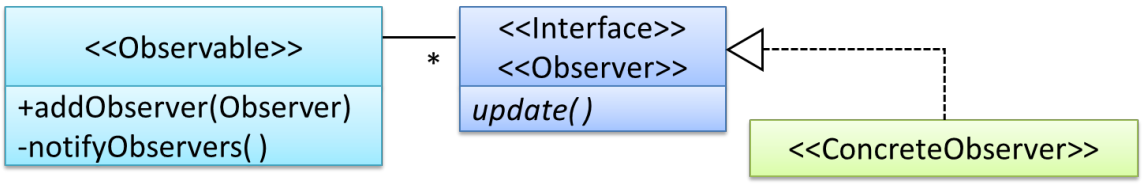
```
//StudentList class
public void notifyUIs() {
    //for each observer in the list
    for(Observer o: observerList){
        o.update();
    }
}
```

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

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

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

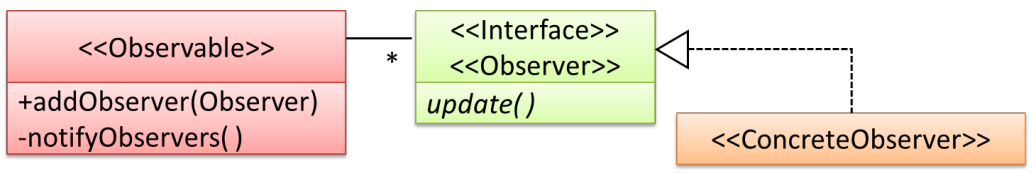
Here is the generic description of the observer pattern:



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

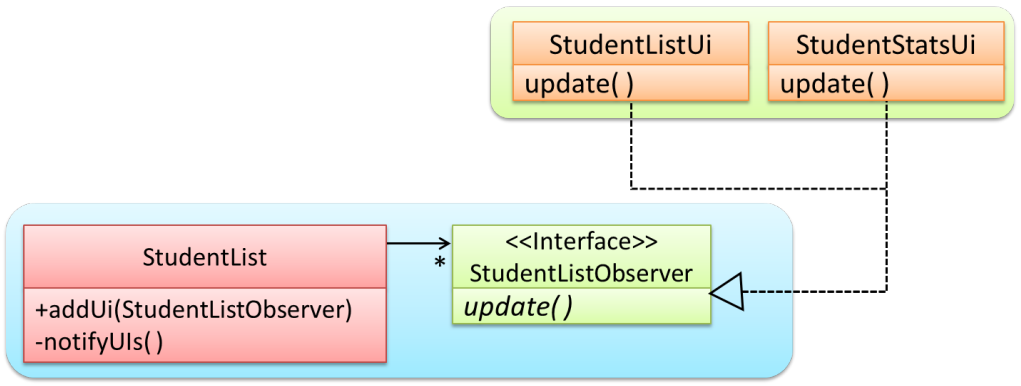
> In a GUI application, how is the Controller notified when the “save” button is clicked? UI frameworks such as JavaFX has inbuilt support for the Observer pattern.

Explain how polymorphism is used in the Observer pattern.



With respect to the general form of the Observer pattern given above, when the Observable object invokes the `notifyObservers()` method, it is treating all `ConcreteObserver` objects as a general type called `Observer` and calling the `update()` method of each of them. However, the `update()` method of each `ConcreteObserver` could potentially show different behavior based on its actual type. That is, `update()` method shows polymorphic behavior.

In the example given below, the `notifyUIs` operation can result in `StudentListUi` and `StudentStatsUi` changing their views in two different ways.



write your answer here...

The Observer pattern can be used when we want one object to initiate an activity in another object without having a direct dependency from the first object to the second object.

- True
- False

True

Explanation: Yes. For example, when applying the Observer pattern to an MVC structure, Views can get notified and update themselves about a change to the Model without the Model having to depend on the Views.

W11.1c ★★★★: OPTIONAL

Design → Design Patterns → Other Design Patterns

W11.1d ★★★★: OPTIONAL

Design → Design Patterns → Combining Design Patterns

W11.1e ★★★★: OPTIONAL

Design → Design Patterns → Using Design Patterns

W11.1f ★★★★: OPTIONAL

Design → Design Patterns → Design Patterns vs Design Principles

W11.1g ★★★★: OPTIONAL

Design → Design Patterns → Other Types of Patterns

[W11.2] Architectural Styles ★

W11.2a ★★★

Design → Architecture → Styles → What

🎓 Design → Architecture → Introduction → What

🏆 Can explain architectural styles

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

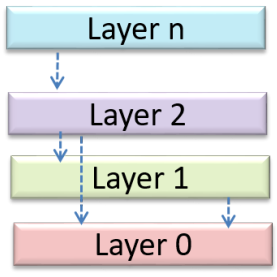
📦 n-tier style, client-server style, event-driven style, transaction processing style, service-oriented style, pipes-and-filters style, message-driven style, broker style, ...

W11.2b ★★

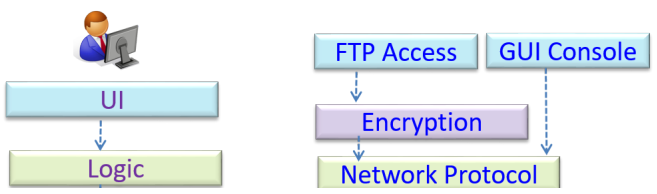
Design → Architecture → Styles → n-Tier Style → What

🏆 Can identify n-tier architectural style

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



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

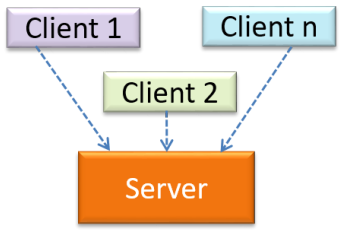


W11.2c ★★★

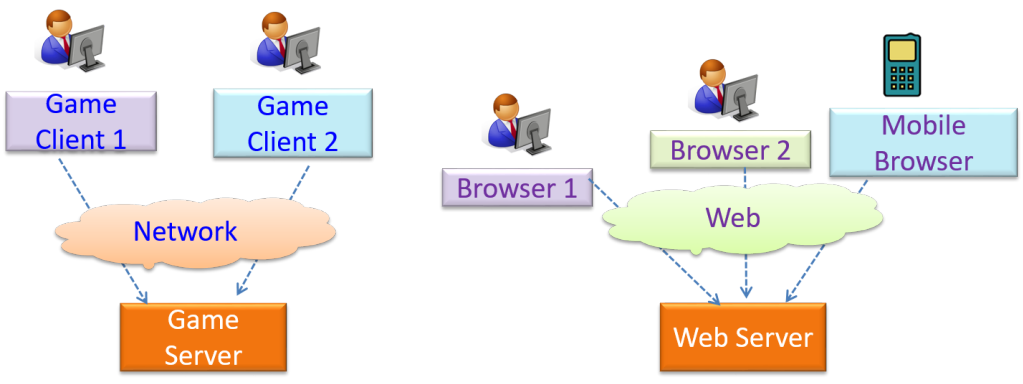
Design → Architecture → Styles → Client-Server Style → What

Can identify the client-server architectural style

The *client-server* style has at least one component playing the role of a server and at least one client component accessing the services of the server. This is an architectural style used often in distributed applications.



The online game and the Web application below uses the client-server style.

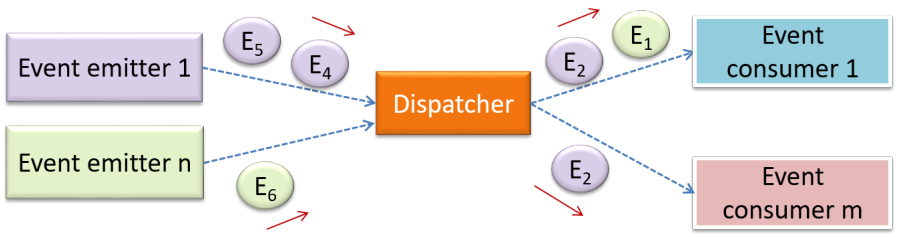


W11.2d ★★

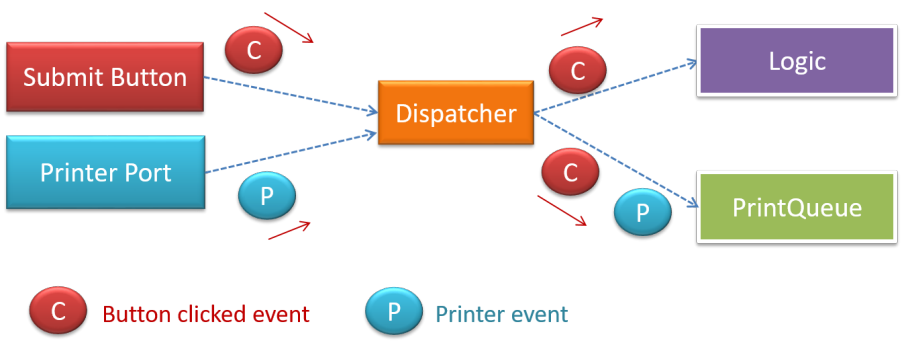
Design → Architecture → Styles → Event-Driven Style → What

Can identify event-driven architectural style

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



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

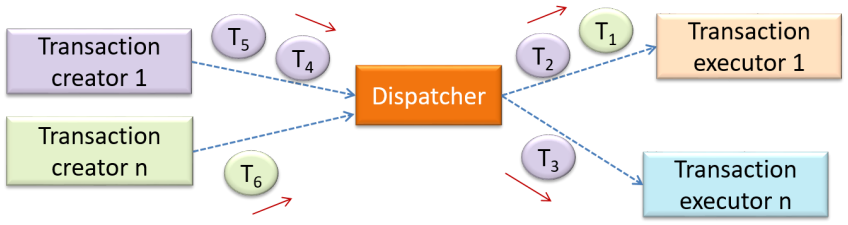


W11.2e ★★★

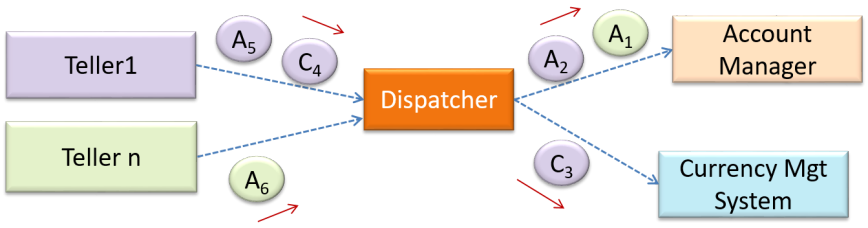
Design → Architecture → Styles → Transaction Processing Style → What

Can identify transaction processing architectural style

The *transaction processing* style divides the workload of the system down to a number of *transactions* which are then given to a *dispatcher* that controls the execution of each transaction. Task queuing, ordering, undo etc. are handled by the dispatcher.



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



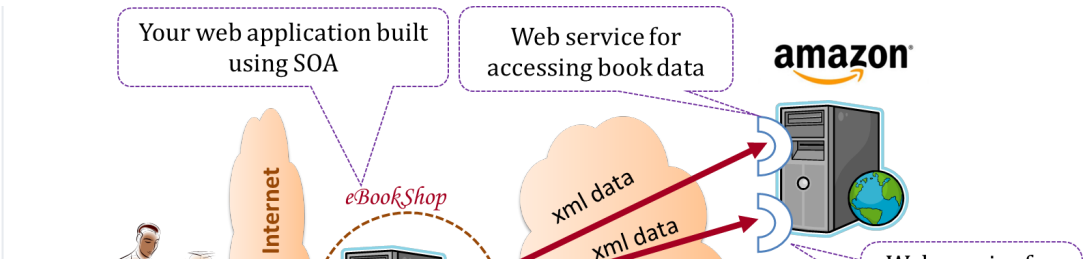
W11.2f ★★★

Design → Architecture → Styles → Service-Oriented Style → What

Can identify service-oriented architectural style

The *service-oriented architecture* (SOA) style builds applications by combining functionalities packaged as *programmatically accessible services*. SOA aims to achieve interoperability between distributed services, which may not even be implemented using the same programming language. A common way to implement SOA is through the use of *XML web services* where the web is used as the medium for the services to interact, and XML is used as the language of communication between service providers and service users.

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



W11.2g ★★★★: OPTIONAL

Design → Architecture → Styles → More Styles

W11.2h ★★★

Design → Architecture → Styles → Using Styles

Can explain how architectural styles are combined

Most applications use a mix of these architectural styles.

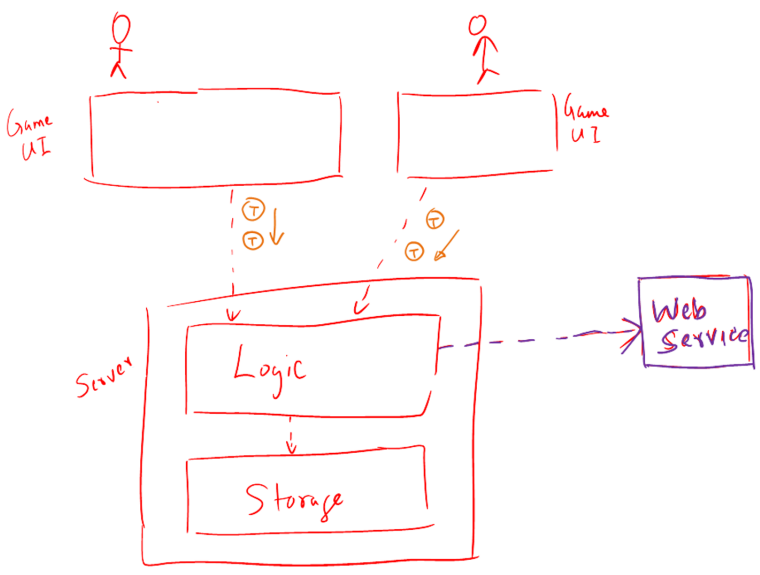
An application can use a client-server architecture where the server component comprises several layers, i.e. it uses the n-Tier architecture.

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

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

1. Client-server
2. Transaction-processing
3. SOA (Service Oriented Architecture)
4. multi-layer (n-tier)

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



write your answer here...

W11.2i ★

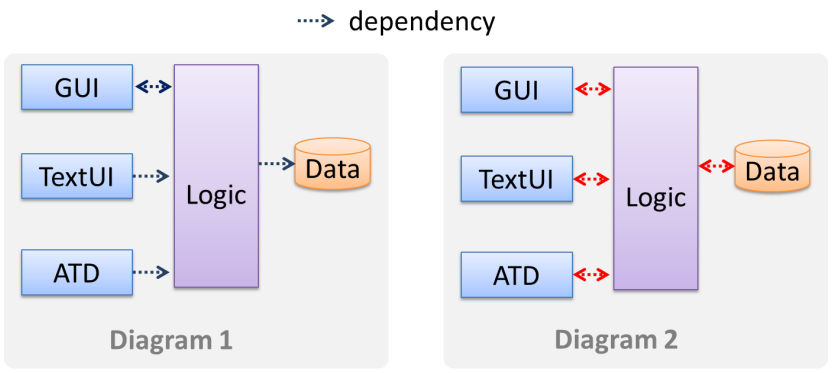
Design → Architecture → Architecture Diagrams → Drawing

Can draw an architecture diagram

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

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

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



[W11.3] Test Cases: Combining Multiple Inputs ★★

W11.3a ★★

Quality Assurance → Test Case Design → Combining Test Inputs → Why

Can explain the need for strategies to combine test inputs

An SUT can take multiple inputs. You can select values for each input (using equivalence partitioning, boundary value analysis, or some other technique).

an SUT that takes multiple inputs and some values chosen as values for each input:

- Method to test: `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test:

Input	valid values to test	invalid values to test
participation	0, 1, 19, 20	21, 22
projectGrade	A, B, C, D, F	
isAbsent	true, false	
examScore	0, 1, 69, 70,	71, 72

Testing all possible combinations is effective but not efficient. If you test all possible combinations for the above example, you need to test 6x5x2x6=360 cases. Doing so has a higher chance of discovering bugs (i.e. effective) but the number of test cases can be too high (i.e. not efficient). Therefore, **we need smarter ways to combine test inputs that are both effective and efficient.**

W11.3b ★★

Quality Assurance → Test Case Design → Combining Test Inputs → Test Input Combination Strategies

Can explain some basic test input combination strategies

Given below are some basic strategies for generating a set of test cases by combining multiple test input combination strategies.

Let's assume the SUT has the following three inputs and you have selected the given values for testing:

SUT: `foo(p1 char, p2 int, p3 boolean)`

Values to test:

Input	Values
p1	a, b, c
p2	1, 2, 3
p3	T, F

The *all combinations* strategy generates test cases for each unique combination of test inputs.

the strategy generates 3x3x2=18 test cases

Test Case	p1	p2	p3
1	a	1	T
2	a	1	F
3	a	2	T
...	...	...	...
18	c	3	F

The *at least once* strategy includes each test input at least once.

this strategy generates 3 test cases.

Test Case	p1	p2	p3
1	a	1	T
2	b	2	F

Test Case	p1	p2	p3
3	c	3	VV/IV

VV/IV = Any Valid Value / Any Invalid Value

The *all pairs* strategy creates test cases so that for any given pair of inputs, all combinations between them are tested. It is based on the observations that a bug is rarely the result of more than two interacting factors. The resulting number of test cases is lower than the *all combinations* strategy, but higher than the *at least once* approach.

📦 this strategy generates 9 test cases:

▶ see steps

Test Case	p1	p2	p3
1	a	1	T
2	a	2	T
3	a	3	F
4	b	1	F
5	b	2	T
6	b	3	F
7	c	1	T
8	c	2	F
9	c	3	T

A variation of this strategy is to test all pairs of inputs but only for inputs that could influence each other.

📦 Testing all pairs between p1 and p3 only while ensuring all p2 values are tested at least once:

Test Case	p1	p2	p3
1	a	1	T
2	a	2	F
3	b	3	T
4	b	VV/IV	F
5	c	VV/IV	T
6	c	VV/IV	F

The *random* strategy generates test cases using one of the other strategies and then pick a subset randomly (presumably because the original set of test cases is too big).

There are other strategies that can be used too.

W11.3c ★★★

Quality Assurance → Test Case Design → Combining Test Inputs → Heuristic: Each Valid Input at Least Once in a Positive Test Case

🏆 Can apply heuristic 'each valid input at least once in a positive test case'

Consider the following scenario.

SUT: `printLabel(fruitName String, unitPrice int)`

Selected values for `fruitName` (invalid values are underlined):

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

Selected values for `unitPrice` :

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

Suppose these are the test cases being considered.

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

It looks like the test cases were created using the *at least once* strategy. After running these tests can we confirm that square-format label printing is done correctly?

- Answer: No.
- Reason: `Cherry` -- the only input that can produce a square-format label -- is in a negative test case which produces an error message instead of a label. If there is a bug in the code that prints labels in square-format, these tests cases will not trigger that bug.

In this case a useful heuristic to apply is **each valid input must appear at least once in a positive test case**. `Cherry` is a valid test input and we must ensure that it appears at least once in a positive test case. Here are the updated test cases after applying that heuristic.

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

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

W11.3d ★★★

Quality Assurance → Test Case Design → Combining Test Inputs → Heuristic: No More Than One Invalid Input In A Test Case

🎓 Quality Assurance → Test Case Design → Combining Test Inputs → Heuristic: Each Valid Input at Least Once in a Positive Test Case

🏆 Can apply heuristic ‘no more than one invalid input in a test case’

Consider the test cases designed in [Heuristic: each valid input at least once in a positive test case].

After running these test cases can you be sure that the error message “invalid price” is shown for negative prices?

- Answer: No.
- Reason: `-1` -- the only input that is a negative price -– is in a test case that produces the error message “invalid fruit”.

In this case a useful heuristic to apply is **no more than one invalid input in a test case**. After applying that, we get the following test cases.

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

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

Applying the heuristics covered so far, we can determine the precise number of test cases required to test any given SUT effectively.

- True
- False

False

Explanation: These heuristics are, well, heuristics only. They will help you to make better decisions about test case design. However, they are speculative in nature (especially, when testing in black-box fashion) and cannot give you precise number of test cases.

W11.3e ★★★

Quality Assurance → Test Case Design → Combining Test Inputs → Mix

Can apply multiple test input combination techniques together

Consider the calculateGrade scenario given below:

- SUT : `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test: invalid values are underlined
  - participation: 0, 1, 19, 20, 21, 22
  - projectGrade: A, B, C, D, F
  - isAbsent: true, false
  - examScore: 0, 1, 69, 70, 71, 72

To get the first cut of test cases, let's apply the *at least once* strategy.

Test cases for calculateGrade V1

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV/IV	69	...
4	20	D	VV/IV	70	...
5	21	F	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

VV/IV = Any Valid or Invalid Value, Err Msg = Error Message

Next, let's apply the *each valid input at least once in a positive test case* heuristic. Test case 5 has a valid value for `projectGrade=F` that doesn't appear in any other positive test case. Let's replace test case 5 with 5.1 and 5.2 to rectify that.

Test cases for calculateGrade V2

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...

Case No.	participation	projectGrade	isAbsent	examScore	Expected
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV/IV	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

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

Next, we apply the *no more than one invalid input in a test case* heuristic. Test cases 5.2 and 6 don't follow that heuristic. Let's rectify the situation as follows:

Test cases for calculateGrade V3

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV	VV	VV	Err Msg
5.3	22	VV	VV	VV	Err Msg
6.1	VV	VV	VV	71	Err Msg
6.2	VV	VV	VV	72	Err Msg

Next, let us assume that there is a dependency between the inputs `examScore` and `isAbsent` such that an absent student can only have `examScore=0`. To cater for the hidden invalid case arising from this, we can add a new test case where `isAbsent=true` and `examScore!=0`. In addition, test cases 3-6.2 should have `isAbsent=false` so that the input remains valid.

Test cases for calculateGrade V4

Case No.	participation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	false	69	...
4	20	D	false	70	...
5.1	VV	F	false	VV	...
5.2	21	VV	false	VV	Err Msg

Case No.	participation	projectGrade	isAbsent	examScore	Expected
5.3	22	VV	false	VV	Err Msg
6.1	VV	VV	false	71	Err Msg
6.2	VV	VV	false	72	Err Msg
7	VV	VV	true	!=0	Err Msg

Which of these contradict the heuristics recommended when creating test cases with multiple inputs?

- a. All invalid test inputs must be tested together.
- b. It is ok to combine valid values for different inputs.
- c. No more than one invalid test input should be in a given test case.
- d. Each valid test input should appear at least once in a test case that doesn't have any invalid inputs.

a

Explanation: If you test all invalid test inputs together, you will not know if each one of the invalid inputs are handled correctly by the SUT. This is because most SUTs return an error message upon encountering the first invalid input.

Apply heuristics for combining multiple test inputs to improve the E&E of the following test cases, assuming all 6 values in the table need to be tested. underlines indicate invalid values. Point out where the heuristics are contradicted and how to improve the test cases.

SUT: `consume(food, drink)`

Test case	food	drink
TC1	bread	water
TC2	rice	lava

[W11.4] Other QA Techniques ★★

W11.4a ★★

Quality Assurance → Quality Assurance → Introduction → What

Can explain software quality assurance

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*.

W11.4b ★★★

Quality Assurance → Quality Assurance → Introduction → Validation vs Verification

Can explain validation and verification

Quality Assurance = Validation + Verification

QA involves checking two aspects:

1. Validation: are we *building the right system* i.e., are the requirements correct?
2. Verification: are we *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 both are done, instead of limiting to verification (i.e., remember that the requirements can be wrong too).

Choose the correct statements about validation and verification.

- ☐ a. Validation: Are we building the right product?, Verification: Are we building the product right?
- ☐ b. It is very important to clearly distinguish between validation and verification.
- ☐ c. The important thing about validation and verification is to remember to pay adequate attention to both.
- ☐ d. Developer-testing is more about verification than validation.
- ☐ e. QA covers both validation and verification.
- ☐ f. A system crash is more likely to be a verification failure than a validation failure.

(a)~~(b)~~(c)(d)(e)(f)

Explanation:

Whether something belongs under validation or verification is not that important. What is more important is that we do both.

Developer testing is more about bugs in code, rather than bugs in the requirements.

In QA, system testing is more about verification (does the system follow the specification?) and acceptance testings is more about validation (does the system solve the user's problem?).

A system crash is more likely to be a bug in the code, not in the requirements.

W11.4c ★★★

Quality Assurance → Quality Assurance → Formal Verification → What

Can explain formal verification

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

▶ An introduction to Formal Methods

Advantages:

- **Formal verification can be used to prove the absence of errors**. In contrast, testing can only prove the presence of error, 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 cannot prove the absence of errors. It can only prove the presence of errors. However, formal methods can prove the absence of errors.

- ○ True
- ○ False

True

Explanation: While using formal methods is more expensive than testing, it indeed can prove the correctness of a piece of software conclusively, in certain contexts. Getting such proof via testing requires exhaustive testing, which is not practical to do in most cases.

[W11.5] **Reuse** ★

APIs

W11.5a ★★

Implementation → Reuse → Introduction → What

🏆 Can explain software reuse

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.

W11.5b ★★★

Implementation → Reuse → Introduction → When

🏆 Can explain the costs and benefits of reuse

While you may be tempted to use many libraries/frameworks/platform 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, or/and 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.

One of your teammates is proposing to use a recently-released “cool” UI framework for your class project. List the pros and cons of this idea.

Pros

- The potential to create a much better product by reusing the framework.
- Learning a new framework is good for the future job prospects.

Cons

- Learning curve may be steep.
- May not be stable (it was recently released).
- May not allow us to do exactly what we want. While frameworks allow customization, such customization can be limited.
- Performance penalties.
- Might interfere with learning objectives of the module.

Note that having more cons does not mean we should not use this framework. Further investigation is required before we can make a final decision.

write your answer here...

Libraries

W11.5c ★

Implementation → Reuse → Libraries → What

Can explain libraries

A library is a collection of modular code that is general and can be used by other programs.

Java classes you get with the JDK (such as `String`, `ArrayList`, `HashMap`, etc.) are library classes that are provided in the default Java distribution.

Natty is a Java library that can be used for parsing strings that represent dates e.g. `The 31st of April in the year 2008`

W11.5d ★★

Implementation → Reuse → Libraries → How

Can make use of a library

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 functionality.

W11.5e ★★

Implementation → Reuse → APIs → What

Can explain APIs

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 GitHub server accepts and the corresponding responses. We can write a program that interacts with GitHub through that API.

When developing large systems, if you define the API of each components early, the development team can develop the components in parallel because the future behavior of the other components are now more predictable.

Choose the correct statements

- [ ] a. A software component can have an API.
- [ ] b. Any method of a class is part of its API.
- [ ] c. Private methods of a class are not part of its API.
- [ ] d. The API forms the contract between the component developer and the component user.
- [ ] e. Sequence diagrams can be used to show how components interact with each other via APIs.

(a) (c) (d) (e)

Explanation: (b) is incorrect because private methods cannot be a part of the API

Defining component APIs early is useful for developing components in parallel.

- ( ) True
- ( ) False

True

Explanation: Yes, once we know the precise behavior expected of each component, we can start developing them in parallel.

Frameworks

W11.5f ★★

Implementation → Reuse → Frameworks → What

Can explain frameworks

The overall structure and execution flow of a specific category of software systems can be very similar. The similarity is an opportunity to reuse at a high scale.

> Running example:
>
> IDEs for different programming languages are similar in how they support editing code, organizing project files, debugging, etc.

A *software framework* is a reusable implementation of a software (or part thereof) providing *generic* functionality that can be selectively customized to produce a *specific* application.

> Running example:
>
> Eclipse is an IDE framework that can be used to create IDEs for different programming languages.

Some frameworks provide a complete implementation of a *default* behavior which makes them immediately usable.

📦 Running example:

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

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

📦 Running example:

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 components or an aspect.

📦 JavaFx a framework for creating Java GUIs. TkInter is a GUI framework for Python.

📦 More examples of frameworks

- Frameworks for Web-based applications: Drupal(PHP), Django(Python), Ruby on Rails (Ruby), Spring (Java)
- Frameworks for testing: JUnit (Java), unittest (Python), Jest (Java Script)

W11.5g ★★★

Implementation → Reuse → Frameworks → Frameworks vs Libraries

🏆 Can differentiate between frameworks and 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.

Choose correct statements about software frameworks.

- [ ] a. They follow the hollywood principle, otherwise known as 'inversion of control'
- [ ] b. They come with full or partial implementation.
- [ ] c. They are more concrete than patterns or principles.
- [ ] d. They are often configurable.
- [ ] e. They are reuse mechanisms.
- [ ] f. They are similar to reusable libraries but bigger.

(a)(b)(c)(d)(e)~~(f)~~

Explanation: While both libraries and frameworks are reuse mechanisms, and both more concrete than principles and patterns, libraries differ from frameworks in some key ways. One of them is the 'inversion of control' used by frameworks but not libraries. Furthermore, frameworks do not have to be bigger than libraries all the time.

Which one of these are frameworks ?

- [ ] a. JUnit

- ☐ b. Eclipse
- ☐ c. Drupal
- ☐ d. Ruby on Rails

(a)(b)(c)(d)

Explanation: These are frameworks.

Platforms

W11.5h ★★★

Implementation → Reuse → Platforms → What

🏆 Can explain platforms

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

📦 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 apps.

📦 **Two well-known examples of platforms are JavaEE and .NET**, both of which sit above Operating systems layer, and are used to develop enterprise applications. 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 the JavaEE applications is the JVM (Java Virtual Machine) that can run on different Operating Systems.
- .NET is a similar platform and a framework. Its runtime is called CLR (Common Language Runtime) and usually used on Windows machines.

[W11.6] Cloud Computing ★★★★: OPTIONAL

W11.6a ★★★★: OPTIONAL

Implementation → Reuse → Cloud Computing → What

W11.6b ★★★★: OPTIONAL

Implementation → Reuse → Cloud Computing → Iaas, PaaS, and SaaS

[W11.7] Other UML Models ★★★★: OPTIONAL

W11.7a ★★★★: OPTIONAL
Design → Modelling → Modelling Structure → Deployment Diagrams

W11.7b ★★★★: OPTIONAL
Design → Modelling → Modelling Structure → Component Diagrams

W11.7c ★★★★: OPTIONAL
Design → Modelling → Modelling Structure → Package Diagrams

W11.7d ★★★★: OPTIONAL
Design → Modelling → Modelling Structure → Composite Structure Diagrams

W11.7e ★★★★: OPTIONAL
Design → Modelling → Modelling Behaviors Timing Diagrams

W11.7f ★★★★: OPTIONAL
Design → Modelling → Modelling Behaviors Interaction Overview Diagrams

W11.7g ★★★★: OPTIONAL
Design → Modelling → Modelling Behaviors Communication Diagrams

W11.7h ★★★★: OPTIONAL
Design → Modelling → Modelling Behaviors State Machine Diagrams