# LO7 - Design software using object-oriented best practices

7.1 - Describe the Single Responsibility Principle

7.2 - Describe the principle of separation of concerns

7.3 - Describe the principle of "tell, don't ask"

7.4 - Describe the dependency inversion principle

7.5 - Describe common design patterns

7.6 - Apply design patterns to simplify software construction

7.7 - Produce a class design using object-oriented best practices

Object-oriented design and development is an immensely complex endeavor. It is often said that, in software development, change is the only thing you can count on. Specifications are rarely well defined, and in many cases the end result of a software project will differ in many ways from the initial intention. Because of this, software design must take into account the need to simplify future change. Over time, software development practitioners have come up with and documented a wide variety of principles, patterns and practices of object-oriented development which can make it easier to build flexible, change-friendly designs.

Probably the most widely known and used groups of these principles are known as the "[SOLID](http://www.objectmentor.com/omSolutions/oops_what.html)" principles of object-oriented design. These principles are:

**S** - (SRP) Single Responsibility Principle. Each class in a system design should have only one responsibility; only one reason to change. If two different changes in requirements cause changes to the same class, it is likely that that class has multiple responsibilities. If changing the database used to store a "Person" object requires you to change the class which represents one on screen, that class likely has too many responsibilities.

**O** - (OCP) Open/Closed Principle. Classes and interfaces should be open for extension (inheritance), but closed for modification. Rather than adding functionality to a completed class, extend (add new code) or refactor it into another class. 22Helps with adherence to the SRP. Rather than adding a "roofIsUp" attribute to "Car", create "Convertible", extending "Car" to add the necessary functionality.

**L** - (LSP) Liskov Substitution Principle. Discussed in an earlier learning outcome. Any object in a system should be able to be replaced by any of its subclasses without affecting the correctness of the program. In the earlier example, replacing a Person object with a Student object should not require changes to the code using those objects.

**I** - (ISP) Interface Segregation Principle. Closely related to the SRP. No client object should be forced to depend on methods it does not use. Essentially, this principle promotes the idea that many small, single purpose interfaces make for a more flexible design than large interfaces, as classes will be less likely to have to implement a method it does not need to have. A classic example is a "File" interface, which contains method definitions for both reading from and writing to a file. Any object implementing this interface would be required to implement both, even if it was using read-only storage. Using the ISP, we would create two interfaces; one for ReadableFiles, the other for WritableFiles.

**D** - (DIP) Dependency Inversion Principle. High level (highly abstract) classes should not depend on low level classes (concrete classes); both should depend on highly abstract classes or interfaces. Rather than directly making use of low level, concrete implementations, we should instead make use of interfaces which describe their behaviour. Low level classes which implement those interfaces can be provided and swapped out as required. This principle is one of the hardest to grasp, but is also one of the most important.

Some of the principles presented here are not part of the "SOLID" design principles, but are more general principles of software development. Mastery of these principles will take a great deal of time and practice; this learning outcome is intended to provide an overview.

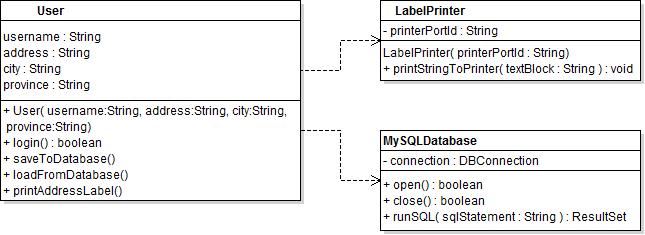
Another useful tool in the creation of object-oriented software is the idea of the *software design pattern*. As your career as a software developer progresses, you will begin to recognize common themes and patterns in the problems you are being asked to solve. It is true that most problems or situations in software development have been solved before; design patterns attempt to form these solutions into general, reusable frameworks which can be adapted to other languages and situations. While software design patterns are an immensely complex topic, a few basic, commonly encountered design patterns will be discussed here.

## 7.1 - Describe the single responsibility principle

The Single Responsibility Principle is the first of the "SOLID" design principles described by [Robert C. Martin](https://sites.google.com/site/unclebobconsultingllc/). The Single Responsibility Principle states that every class should have a single responsibility, and that the responsibility should be entirely contained by that class. It has also been defined as "There should never be more than one reason for a class to change."[[1]](#footnote-2)

The responsibilities of a class are usually directly related to the requirements stated by a customer for that given business object. For example, developers may have discovered through the requirements gathering process that the users of a system must be able to log in, set their username and password, send messages to other users, print address labels for themselves and must be persisted in the database. In a design which does not adhere to the single responsibility principle, a developer might include all of these functions into a single "User" class. While this has the advantage of collecting all user-related functionality into one place, the class will likely be very large and be making use of many other classes. A class which contains many loosely related functions or responsibilities is said to lack cohesion; concise and cohesive designs should always be something that developers strive to achieve.

Below is an abbreviated class diagram for the "User" class described above. As you can see, even with only a few methods it is bound to several other classes. (09-naïve\_user)



One way to identify a class with too many responsibilities is to think about whether it would be easy to re-use in another project. Classes which are easy to re-use generally have few dependencies, or other classes which they require in order to function. In this case, we are showing two mostly unrelated dependencies on the User class: LabelPrinter and MySQLDatabase. These classes represent two vastly different sets of functionality, and are very low level classes. LabelPrinter appears to directly interact with some sort of hardware printer port, while MySQLDatabase seems to be built for the single purpose of running SQL statements against a MySQL database server. Because the User depends on these classes, and because the classes are so specific, it would be difficult to re-use the User class in another project.

Besides directly thinking about the responsibilities or features themselves, a good exercise to perform when checking to see if a class has too many responsibilities is to think about situations under which a class would have to be changed. If the client changed their stated requirements around what database engine the product should use, would the User class need to change? As an exercise, examine the class diagram from above and try to come up with at least 5 requirements changes the client could make which would make it necessary to change the code of the User class.

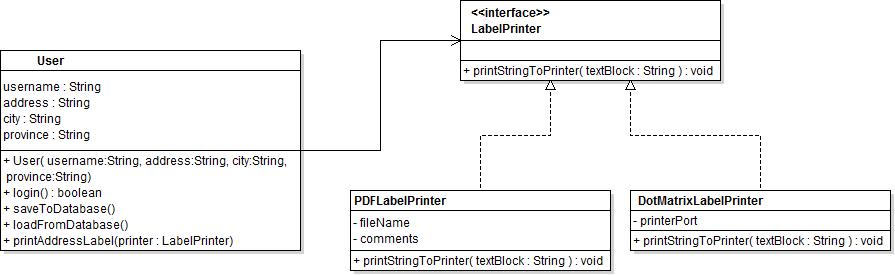
At the moment, the User class has far too many reasons to change.

* Client changes required database server software, requiring changes to User to change SQL statements generated and the class name of the database connector class
* Database Management team changes field names for username or password, requiring changes to all methods which use the database
* Security team requests that a different password hashing algorithm be used, requiring changes to login checks and user save SQL
* Client wants to print PDF labels rather than simple text labels, requiring flags to choose the printer and additional code in the "printAddressLabel" method
* The client wants to be able to choose which port the LabelPrinter is connected to

These are just a few of the situations which might require code changes in the class design shown above, though there are many others. These three classes are far too tightly coupled together. The User class as it exists now depends on the exact concrete implementations of the MySQLDatabase and hardware LabelPrinter classes, meaning that any change affecting those classes will require changes to the User class itself.

Correcting violations of the SRP generally involves making use of Abstraction; rather than having a class like User depend directly on a low level concrete class, define an interface which describes the action that must be performed and have the User class rely on it instead. If the underlying implementation of that action needs to change, a new implementation of that interface could be created and swapped in to replace the old one with no other code changes.

In this case, the User needs to be able to send a string of text to some object and have it printed in some way. With the current implementation, the User class' "printAddressLabel" method would be creating a LabelPrinter, setting its port number, creating a string of text to describe itself and then calling the "printStringToPrinter" method to have the printer output a page. Instead, the class design could be changed as follows: (picture - 09\_print\_interface\_user­)



Rather than depending directly on the actual label printer, we define an interface that represents what User requires of a label printer, then create different implementations of that interface for different situations. Depending on the required output, other classes can provide User with a different actual implementation of LabelPrinter, but the code within User stays exactly the same. While this approach does not remove all of the printing-related reasons for the User class to change (changing the string to be printed would still require changes in User), it is a good incremental step towards compliance with the SRP. This method of having classes depend on abstractions rather than concrete implementations is an excellent way to reduce the complexity of your code, and to make change simpler and less painful.

Being able to identify classes which are violating the Single Responsibility Principle is a very valuable skill, even if at this point you are unable to correct the problem. In the final learning step of this outcome, we will work though a similar problem case in order to correct violations of the SRP.

Section Links:

Classical example of an object which displays itself, and how to refactor

<http://www.codeproject.com/Articles/611593/SOLID-Principles-Single-Respons>

The Single Responsibility Principle chapter from Agile Programming by Robert Martin

<http://objectmentor.com/resources/articles/srp.pdf>

<http://java.dzone.com/articles/single-responsibility>

<http://en.wikipedia.org/wiki/Single_responsibility_principle>

<http://code.tutsplus.com/tutorials/solid-part-1-the-single-responsibility-principle--net-36074>

## 7.2 - Describe the principle of separation of concerns

The principle of separation of concerns, while not one of the SOLID principles, is an important concept. It is very closely related to the Single Responsibility Principle, but rather than always dealing with individual classes, separation of concerns deals with more high level concepts. SoC encourages the separation of a program into distinct sections or components, each dealing with a specific set of information. It encourages a modular design, where changes to one part of a program do not affect other parts.

SoC is implemented in much the same way as the Single Responsibility Principle: by defining set interfaces which describe the behaviour of a class or set of classes, and having components depend on those interfaces rather than the specific, low level details of how those classes are implemented. Good use of encapsulation (information hiding) aids in creating this separation between the modules in a system, as it prevents other functional areas of the program from having access to the details of how other areas are implemented. Having well defined interfaces between the modular parts of a system will generally lead to a more maintainable and easy to modify system, as changes to one segment of the program will have less effect on other parts.

One of the most prevalent examples of Separation of Concerns is the idea of the three-tier architecture. A three-tier architecture is a method of application design where an application is separated into three distinct segments. Each segment communicates only with the directly adjacent segment, which helps to encourage separation of concerns.

1. **Presentation**. Commonly considered the "top" tier, this portion contains the user interface and user interaction portions of the application.
   1. Responsibilities include presenting information from logic layer to the user, receiving and translating user input into commands for the logic layer, and informing the user of ongoing operations at lower layers.
   2. Generally specific to the client. Different implementations of the presentation layer will be created to cater to different clients, but all will use the same set of logic and data access layers.
2. **Business logic**. This "middle" tier is often considered the glue of the application.
   1. Responsibilities include processing commands from the presentation layer, applying business rules, logic and validation. Commands the data tier to persist or retrieve data, and the presentation layer to display output.
   2. Generally the most portable of the three layers, as it is specific to the application and business rules, rather than any underlying platform.
3. **Data tier**. Concerned with storing and retrieving data. Provides an abstract interface to data storage and retrieval, making data access from the business logic layer independent of how the data is actually stored.
   1. Having a well designed interface between the data and business logic layer make it easier to replace the data-storage mechanism. For example, switching from storing data in flat files to using a relational database.
   2. The interface between the business logic and data tiers usually takes the form of "domain objects", or classes which represent the concepts the application makes use of. For example, the data tier of a well-designed calendaring application might accept and produce lists of "Appointment" objects, rather than accepting SQL statements and producing a cursor or result set.

Using these three tiers can be an excellent way to frame discussion while designing a piece of software. Going through each use case, then breaking it apart into different activities at each layer can be very beneficial in developing a modular software design. We will use this technique to break down a small set of software requirements into a basic class design in a later learning step.

<http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips>

<http://en.wikipedia.org/wiki/Separation_of_concerns>

<http://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/>

## 7.3 - Describe the principle of "tell, don't ask"

The principle of "Tell, don't ask" encourages developers to write code where objects tell each-other what to do, rather than asking them questions about their state and making decisions or calculations outside of those objects. Software in which each class has a well-defined responsibility will often naturally follow this principle; rather than thinking about how each class is implemented, developers will think of what actions each class will need to perform.

Tell, don't ask is directly related to the concept of encapsulation, or information hiding. Rather than providing access to their internal implementation, classes provide a set of actions they can perform, but hide how those actions are internally implemented. Make decisions which affect an object inside of that object, rather than outside.

Imagine an application where different types of users were to see a different banner message at the top of each screen. A common implementation of this sort of behaviour would be as follows:

public class LoginScreen  
{

private User currentUser;

//…

public void displayLoginMessage(TextView messageView)

{

if(currentUser.is\_administrator)

{

messageView.setText(User.admin\_message);

}

else if(currentUser.is\_standard\_user)

{

messageView.setText(User.standard\_user\_message);

}

else if(currentUser.is\_guest)

{

messageView.setText(User.guest\_message);

}

}  
}

What problems can you see with this implementation? Primarily, it is not using encapsulation. While this implementation may work fine, what happens if the same message is to be shown on every screen, rather than just the login screen? The developer would have to copy the "displayLoginMessage" method to every one of the other screens. If a new user level were added, the developer would have to modify every one of those implementations, leaving themselves vulnerable to mistakes. A much more effective method would be as follows:

public class LoginScreen  
{

private User currentUser;

//…

public void displayLoginMessage(TextView messageView)

{

messageView.setText(currentUser.loginMessage());

}  
}

In this version, the logic behind what message to display for each user type is encapsulated within the User class. It exists in one place, and rather than asking the *currentUser* object what type it contains and making decisions external to the User class, the LoginScreen tells the User class to provide a login message.

An equally common sight in poorly designed code is "train-wreck" method chaining, where a method reaches deep into a class hierarchy to retrieve or set some value. For example, a developer working on an ordering system might need to apply a discount to all items of an order with a price over $35. A very mild "train-wreck" might look like this:

public class ShoppingCart

{

//...

private void applyDiscount(Order currentOrder)

{

for(int i=0; i < currentOrder.items.length; i++)

{

if(currentOrder.items[i].price > 35)

{

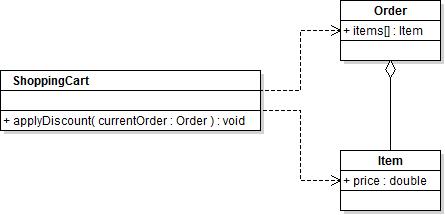
currentOrder.items[i].price \*= 0.95;

}

}

}

}

The problem here is one of depth of knowledge. The ShoppingCart class must know that the Order class contains an attribute named items, that that array is a collection of Item objects, each of which has a numerical attribute named price, which represents the final price of the item. Accessing attributes within attributes within attributes not only makes any change difficult, it makes the methods very difficult to test, since every class along the chain needs to be present in order to write a test. This also means that the ShoppingCart class is dependant (bound to) both the Order and Item classes, as it uses attributes of both. Note that this dependency would still exist if the three classes were using "getters and setters" for their properties rather than simply having public properties; the getters and setters may provide some layer of security, but are still providing information about the internal implementation of each class.

Instead, the Order object should have the ability to apply a discount to its own items, given a set of conditions. Rather than the ShoppingCart object knowing that Orders contain Items which have prices, it would only know that it can tell an order to apply a discount, without needing to know how that action is actually implemented.

public class ShoppingCart

{

//...

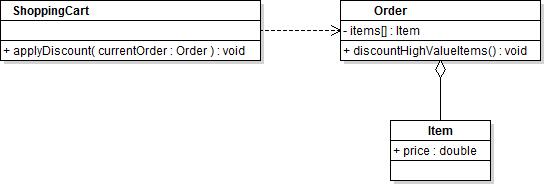
private void applyDiscount(Order currentOrder)

{

currentOrder.discountHighValueItems();

}

}



The knowledge of how the discount is performed and how each class is implemented is now hidden. The ShoppingCart class can now command the Order object to discount its high valued items, leaving the details of how to do so to the Order class itself. The fact that Order objects contain Items is now hidden from the ShoppingCart, and so the dependency between ShoppingCart and Item no longer exists.

While this example used only a three layer deep hierarchy of objects, it is quite common to have a hierarchy that is many layers deeper. In such systems, use of the "tell, don't ask" principle is a good way to keep class diagrams and associations from becoming spaghetti-like and unmanageable.

The [Law of Demeter](http://c2.com/cgi/wiki?LawOfDemeter) is a concept closely related to the "tell, don't ask" principle.

<http://pragprog.com/articles/tell-dont-ask>

<http://martinfowler.com/bliki/TellDontAsk.html>

## 7.4 - Describe the dependency inversion principle

The Dependency Inversion Principle attempts to decouple the behaviour required of a system from how it is actually implemented. It states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions
2. Abstractions should not depend on details. Details should depend on abstractions.[[2]](#footnote-3)

Most object-oriented software consists of high level, more abstract classes which model business logic and the processes of the system, with a series of low level, concrete classes which implement system-specific details. Essentially, the Dependency Inversion Principle states that a class should not rely on classes or objects that are more concrete (low level) than itself, but rather should depend on interfaces which describe the required behaviour. The actual implementation of those interfaces should be separate from the definition of the behaviour. This is often the most difficult of the SOLID principles to apply in practice, but it is also one of the most useful when applied correctly.

What is a dependency?

* Any other class which a class makes use of by name, either directly or indirectly
* Third-party frameworks or libraries
* Database systems
* System resources
* Time

In classical software design, software was typically constructed of a set of high level concepts or requirements, which were broken down into smaller and smaller sections and procedures. Program flow was typically modeled as a set of finely defined steps, each done in sequence, with each step depending on the one before it. Each step was very sensitive to changes made to how earlier steps were actually implemented; changes at an early step could require many more changes in later steps**. Dependency inversion attempts to reduce the overhead of change by distinguishing between *what* something does and *how* it does it.**

It is common to see newcomers to software development tend to combine both the user interface and functionality aspects of their software within the same class or even method. For example, a developer might wish to have a request to a web API triggered when a user clicks the save button on a product editing screen within their application, and so would place the code to create, send and handle the results of that web request within the event handler for that button. This type of code makes change difficult - if the web api were to change or be replaced, they would need to directly change the code in their event handler to allow it to fit with the changed API. If, instead, they were to create an interface defining a method named something like "saveProduct(Product updatedProduct)", they could have their UI event handler make use of the methods defined by that interface instead. The developer could create implementations of that interface to work with different versions of the web api, without having to make changes to their UI code. This separation of behaviour from implementation is essential when creating large software projects, as it can help to make change much easier to manage.

### Further reading:

<http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

<http://martinfowler.com/articles/dipInTheWild.html>

<http://code.tutsplus.com/tutorials/solid-part-4-the-dependency-inversion-principle--net-36872>

## 7.5 - Describe common design patterns

Design Patterns are sets of class designs and interactions used to solve common problems encountered in software development. While the systems and products created by software developers are often very unique, the individual activities that take place within the software are often problems that have been encountered and solved by generations of previous software developers. Design patterns are intended to be flexible - able to be adapted to suit a wide range of programming languages, frameworks and problem areas.

Design patterns are often separated into three categories:

* **Creational**. These patterns describe common ways in which objects or sets of objects can be instantiated and connected together. They seek to hide the details of how classes are actually instantiated in order to reduce the complexity of the parts of the software which use those classes.
  + Singleton
* **Behavioural**. These patterns describe solutions to common sets of object interactions. They seek to create structure around how different objects communicate with each-other, in order to make those interactions more flexible and modular.
  + Iterator
  + Repository
* **Structural**. These patterns help to create simpler and more modular inheritance and object relational structures.
  + Adapter

In this section, we will explore several common design patterns. This is not a comprehensive list of design patterns - thousands exist and are in common usage. The book [*Design Patterns: Elements of Reusable Object Oriented Software*](https://en.wikipedia.org/wiki/Design_Patterns) (more commonly known as the "Gang of Four") is considered by many to be the most important work in existence when it comes to describing software design patterns. While design patterns were in widespread use before the Gang of Four book was published, it was one of the first to describe a set of common behaviours, patterns and guidelines for their use, and is still relevant to this day.

### Singleton

The [Singleton](https://sourcemaking.com/design_patterns/singleton) pattern is used to ensure that there will only be one instance of a given object at any given time in a software system. It is also occasionally used to defer the creation of an object which may take some time to instantiate to when it is actually going to be used. The singleton pattern is very simple to implement in most languages, though care should be taken to ensure that the "only one at a time" behaviour is actually needed.

For example:

public class Logger

{

private static Logger uniqueInstance;

private Logger()

{

}

public static synchronized Logger getInstance()

{

if (uniqueInstance == null)

{

uniqueInstance = new Logger();

}

return uniqueInstance;

}

public void log(String message)

{

//Save message to log file

}

}

This class implements a logging facility using the Singleton pattern. It's likely that a logging system would need to open a log file, write messages to it, then possibly close the log file afterwards - it would be inefficient to create and destroy an instance of the logger every time it was needed. While a developer could simply create one single logger instance and pass it in to every object which needed to make use of it, it would be considerably simpler to use a class such as the one shown. Logger has a private constructor - because of this, only methods within the Logger class can instantiate the class. Logger exposes a single static method, *getInstance()*, which, on the first call, will create and save an instance of the Logger class. Future calls to *getInstance()* will return the existing Logger instance rather than creating a new one.

### Repository

The [Repository](https://msdn.microsoft.com/en-us/library/ff649690.aspx) design pattern seeks to help developers create more modular code when they are tasked with fetching data from some remote data store, such as a web API, database or file structure. Rather than falling into the common trap of directly instantiating a database connection, fetching data and converting it to domain objects directly where those objects are to be used, developers using the repository pattern would create an interface with methods describing the action of fetching or saving a collection of those domain objects. Code which needs to interact with the data store will now use an implementation of that repository object, rather than fetching the data directly. This allows for a more flexible system, since changing the source of the data from a database to a web api would now only require writing a new implementation of the repository class, rather than having to change the code which uses the objects it provides.

As an example, think of an application which needs to fetch a list of users from a database and display them on screen. During the analysis phase, the designers may have sketched out the flow of events as:

1. Connect to the database
2. Fetch a set of user records
3. For each record in the set, add the first and last names to the displayed list

public class UserListScreen

{

private ListView userList;

//...

public void displayUsers()

{

//...UI setup

MySQLDatabase dbConnection = new MySQLDatabase("username", "password", "databasename");

String userSQL = "SELECT firstName, lastName from Users";

Cursor userCursor = dbConnection.execSQL(userSQL);

if(userCursor.moveToFirst())

{

do{

String name = cursor.getString(0) + " " + cursor.getString(1);

userList.items.add(name);

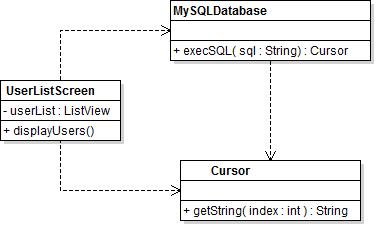
} while(userCursor.moveToNext());

}

}

}

(picture: 09\_dip\_poor)



This example class is an example of a class with *hidden* dependencies, or dependencies which are not declared in the signature of the class. It uses the *MySQLDatabase* class to fetch a set of records from the database, but the MySQLDatabase class is not listed as an attribute of the class or as part of a method signature. It also depends on the Cursor class, as well as on the structure of the "Users" database table having fields named "firstName" and "lastName". Classes such as this one would make change to the system as a whole rather difficult; switching to new database server software would require changes to each method and class which makes use of the MySQLDatabase class. This class is also an example of poor separation of concerns, as in addition to fetching user information from a database, the displayUsers() method is also constructing the database connection itself. Separating object construction from the use of those objects is also one of the benefits of dependency inversion.

The DIP states that high level modules, such as UserListScreen, should not depend on low level modules. In this case, UserListScreen is depending directly on the MySQLDatabase class, which is a very concrete, low level class. If developers needed to change to a new database system or to use test data, they would have a very difficult time doing so with the existing system. What UserListScreen actually needs to do is somehow obtain a list of users, then display that list of users on screen. Instead of depending directly on the MySQLDatabase class, DIP states that we should depend on an interface which describes what activities need to be performed, and have a concrete implementation which actually performs the work.

public interface UserRepository

{

public ArrayList<User> allUsers();

//...

}

public class User

{

private String firstName;

private String lastName;

public User(firstName, lastName)

{

this.firstName = firstName;

this.lastName = lastName;

}

//...

}

public class UserListScreen

{

private ListView userList;

private UserRepository userRepo;

//...

public UserListScreen(UserRepository repository)

{

//…

this.userRepo = repository;

}

public void displayUsers()

{

//...UI setup

ArrayList<User> users = userRepo.allUsers();

userList.addAll(users);

}

}

Instead of using the database directly, we abstract out the idea of "some object which can fetch a list of Users", creating a *UserRepository* interface to describe a class with that ability. For the moment, this class describes a single action, "allUsers", which returns an ArrayList of *User* objects. Each User object captures the information describing a single User. Also, the UserListScreen class now explicitly states its dependencies; it states that it requires an instance of some class which implements the "UserRepository" interface, and thus provides an "allUsers()" method which returns an ArrayList of *Users*.

What has this gained us? Since *UserListScreen* no longer depends on any specific implementation of fetching *User* objects, we are free to create or replace the concrete implementation of *UserRepository* to fit our needs and passing it in to *UserListScreen* on construction. To fetch Users from a *MySQLDatabase*, we could create an implementation of *UserRepository* like this one:

public class MySQLUserRepository implements UserRepository

{

private MySQLDatabase database;

//...

public ArrayList<User> allUsers()

{

ArrayList<User> users = new ArrayList<User>();

String userSQL = "SELECT firstName, lastName from Users";

Cursor userCursor = dbConnection.execSQL(userSQL);

if(userCursor.moveToFirst())

{

do{

String firstName = userCursor.getString(0);

String lastName = userCursor.getString(1);

users.add(new User(firstName, lastName));

} while(userCursor.moveToNext());

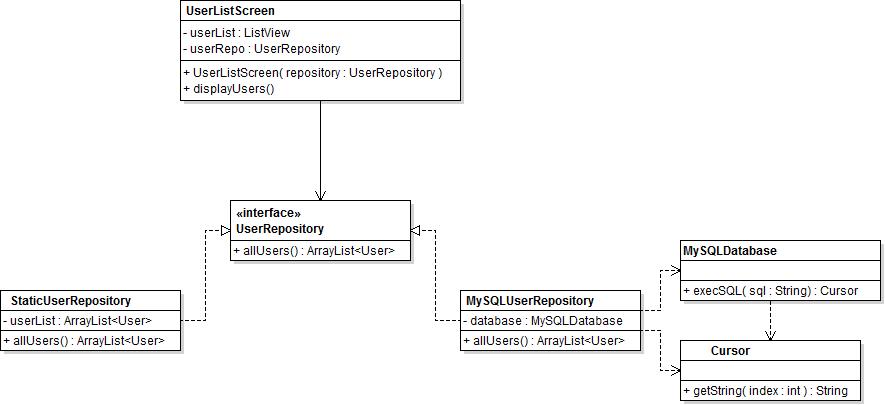
}

}

}

Since *MySQLUserRepository* implements (is-a) *UserRepository*, it can be passed in to the constructor of *UserListScreen*. If, for example, the field names of the Users table change to "first\_name" and "last\_name", only the *MySQLUserRepository* class would have a reason to change; the classes which use it (*UserListScreen*) are insulated from those changes by the *UserRepository* abstraction. If developers did not yet have the database server set up and running, they could create a *StaticUserRepository* implementation, which returned a hard-coded list of User objects, and pass it in as an argument to the *UserListScreen* constructor instead. With the use of dependency inversion and abstraction, *UserListScreen* is said to be insulated from change.

In the class diagram for this more abstract system, we can see that *UserListScreen* depends only on the *UserRepository* interface, rather than directly on the concrete implementations of that interface. (picture 09\_dip\_better)



### Iterator

A very common action in any software system is to have to inspect and interact with each of the items in some sort of collection, such as an array, Stack or Queue. While there are many different ways of performing this sort of task, a standard way of performing this sort of operation is implemented in most programming languages called an *Iterator*.

The [Iterator](https://sourcemaking.com/design_patterns/iterator) behavioural design pattern describes an object which can be used to move through a collection of objects of the same type, regardless of what type of collection object is being used. In Java, collection classes (such as arrays, ArrayList, Stack and Queue) all implement the *Iterable* interface. This interface forces those collection classes to implement a method named *iterator()*, which returns an object which implements the *Iterator* interface. This instance of an Iterator subclass has methods which can be used to traverse through the elements contained within the collection it was created by; since every Iterable collection returns a subclass of Iterator, the same code can be used to iterate over the items contained within many different types of collections.

For example, read through the code below:

//Create a collection and add items to it

ArrayList<String> names = new ArrayList<String>();

names.add("Dan");

names.add("Albert");

names.add("Enid");

names.add("Bob");

names.add("Dave");

names.add("Fred");

//Move through the items in the collection one at a time,

// printing each out to the console.

Iterator<String> iterator = names.iterator();

while(iterator.hasNext())

{

String nextName = iterator.next();

System.out.print(nextName + "\n");

}

In this example, a collection of names is created and several items are added to the collection. In this case, the collection is an ArrayList of String objects. Next, an Iterator of Strings is created, which is then used to move through the collection one item at a time until there are no more items to inspect. While the same behaviour could have been implemented using a simple for loop with an index, not every collection can be directly indexed into. Notice that the second block of code contains no reference to the fact that *names* is an ArrayList - it simply uses the Iterator provided by it to move through the collection. In fact, the type of the collection could be changed to a Stack, Queue, Vector or any other generic collection, and the code in the second block would still work exactly as it is.

The Iterator pattern simply describes a set of common behaviours to be used when moving through a collection, and is generic enough to be implemented by almost any class containing or generating a sequence or collection of values. Code making use of the iterator does not need to have any knowledge of how the collection is internally implemented, making it much easier to write modular code.

### Adapter

The [Adapter](https://sourcemaking.com/design_patterns/adapter) pattern is a structural design pattern which allows existing, otherwise incompatible objects to be used together. Developers often use the adapter pattern without realizing it, as classes using the pattern often don't contain the name "Adapter", but still perform the same types of actions.

A simple example of the Adapter pattern can be found in the Java standard library, in the [*Collection.toArray()*](http://docs.oracle.com/javase/6/docs/api/java/util/Collection.html#toArray()) method. In Java, arrays are a special type of object which can be indexed into using the square bracket notation ( ie names[0] to access the first item in the names array ). Arrays are widely used within the Java standard library as parameters to many different methods. Because arrays are of a fixed size and must be recreated in order to be expanded or contracted in size, developers often use on of Java's collection classes instead, such as ArrayList or Vector. These classes offer much greater flexibility than arrays, but cannot be simply used in place of an array - collection classes and arrays have incompatible interfaces. The *toArray()* method acts as a bridge between the two. Every class inheriting from Collection will have an implementation of the *toArray()* method, which will create and return a simple Java array containing the same items as the collection the method was called on.

### Observer Pattern

The [Observer](https://sourcemaking.com/design_patterns/observer) pattern is an excellent fit for this situation. The Observer pattern describes a way to structure communication between objects when multiple objects wish to be notified of a change to some "source" object. The observer pattern is very widely used - almost every graphical user interface toolkit (such as Swing or JavaFX) uses the observer pattern for event handling, such as button clicks. In the observer pattern, an interface is created with methods to define the notification being sent to the observing objects. The source object is given a collection of objects which implement this interface, and calls the appropriate method defined by that interface when an event occurs.

Diagram

Description automatically generatedDiagram

Description automatically generated

In this case, our Student objects are the observers; they wish to be notified by the system when a new Assignment is posted. While we could simply have an "*assignmentPosted"* method on the Student class, with a collection of students to be notified by code in SubmissionHandler when an assignment is submitted, we gain flexibility by following the Observer pattern. If we create an interface to describe the behaviour of an object which is interested in an assignment being posted, then have SubmissionHandler depend on that interface rather than directly on Student, we gain the ability to have classes other than Students be notified of submitted assignments without having to modify SubmissionHandler. Any class which implements the AssignmentListener interface can now register itself with SubmissionHandler, and be notified by having its *assignmentPosted* method called.

## 

## 

## 7.6 - Apply design patterns to simplify software construction

Hunting through your software for places to use design patterns is generally considered to be bad practice - trying to force yourself to use design patterns wherever possible is likely to result in code that is difficult to understand and modify. Once you have experience in software development, and have become familiar with some common design patterns, you are likely to begin to find yourself using them frequently in your projects without even realizing it. In the example in this learning step, you will discuss and choose a design pattern which could be used to simplify the construction of a small portion of a software system.

### CST University Notification

You have been asked to implement a portion of a student assignment calendar system at CST University. Students at CST University want to know when their instructor has posted anew assignment, but currently have to check their calendar a few times per day to see if a new assignment is present. Rather than having to constantly check for new assignments, students want to be able to be notified when a new assignment is posted so that they can start working on it as soon as possible.

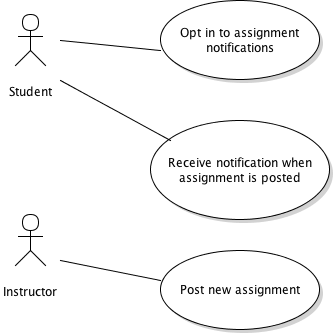
A simple, widely used design pattern exists which could be used to assist in designing the code you could possibly write while implementing this feature. In order to help you discover the candidate pattern, take a moment to brainstorm and write down a set of actors for this system. It may be useful to create a simple use-case diagram for the system. Once you have created your list and/or diagram, proceed to the next page.

### Discover Actors and Classes

Remember the process for discovering actors from an earlier learning outcome. Search the requirements for nouns to find candidate actors. These actor names will often become classes in the final system. Search for verbs to find the actions those actors perform, which will often become methods on the classes you create. In this example, we can easily discover a set of possible actors:

1. Student
   1. wants to be notified when a new assignment is posted to the system
   2. wants to be able to sign up for notifications
2. Instructor
   1. wants to post new assignments to the system

Below is a use-case diagram to describe the same set of behaviours.



Notice that some behaviour was included here that is likely not part of the feature you are adding: the ability to post a new assignment. That particular feature is described in the requirements as already in existence - we are adding additional functionality to it. In this case, the system as currently implemented is likely to have at least four classes:

1. Student - notified of new assignment, requests to be notified of new assignments
2. Instructor - submits new assignments
3. Assignment - information about this is sent to interested parties
4. SubmissionHandler - takes in assignments submitted by Instructors for processing

The common theme in this example is that there will be some collection of objects (one Student instance per student in the class) who must be notified when some change happens in the system, in this case the act of an assignment being submitted. This sort of workflow is extremely common in software projects. With this knowledge in mind, take some time to find a design pattern which describes this type of behaviour; an excellent source would be the SourceMaking behavioural patterns list, found [here](https://sourcemaking.com/design_patterns/behavioral_patterns). Since the pattern describes an interaction and communication between objects, you are most likely to find the pattern you are looking for in the set of Behavioural patterns. Once you have found a pattern that fits, proceed to the next page.

## 

## 

# 7.7 - Produce a class design using object-oriented best practices

## You have been tasked with creating a basic financial logging system. Read through the System Requirements below.

## System Requirements

For now, the system needs only to consist of an Account, which contains the name of the owner and an initial balance along with a set of transactions which have taken place on that account. Each transaction consists of the date on which the transaction was posted, along with the amount of the transaction. Transactions can be either Debit transactions, which represent a withdrawal of money, or a Credit transaction, which represents a deposit of money. The account must be able to have new transactions added to it, as well as the ability to calculate the current balance. The current balance is defined as the opening balance, plus the amount of each credit transaction, minus the amount of each debit transaction. The opening balance will never be changed. Instead, the current balance is recalculated each time it is requested.

In the above example, our client has set out very clear requirements. The client has also given their developers an excellent set of domain specific nouns which can easily be turned into class names.

Search through the system description above and complete steps 1, 2, and 3 below.

* + - 1. Pick out a list of nouns which are likely candidates for conversion into classes in the system. List the nouns below:

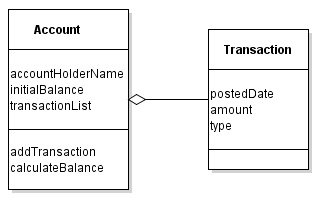
* + - 1. For each class, include the set of attributes that class will need to contain, as well as any behaviours required of that class. List the attributes and behaviours below:  
           
           
           
         3. Once you have created this list, format it into a basic analysis class diagram for the system. Save the diagram in a program of your choice (ie. Visio). *Remember that analysis class diagrams are intended to be basic descriptions of the system, so you can omit any attribute types if they are obvious by the name of the attribute*.

## Initial design

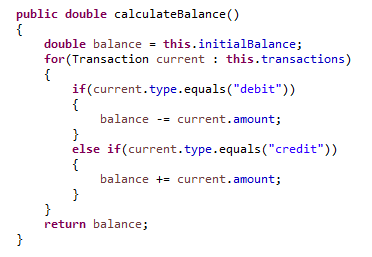
As defined by the requirements, we will need at least two classes:

1. **Account**. This class keeps track of the account holder's name, initial balance, and a list of transactions which have been posted to the account. "Posted" is an excellent example of domain language which might be used by a client. A developer might be tempted to use "added" or "applied", but the word "posted" has special meaning when used in the context of financial systems. The Account class is required to be able to add a new transaction to the current list of transactions, as well as calculating a new balance based on the existing balance.
2. **Transaction**. This class keeps track of the date on which the transaction was posted, as well as the amount involved in the transaction. It is split into two sub-types:
   1. CreditTransaction. The amount tracked by this type of transaction will be added to the opening balance.
   2. DebitTransaction. The amount tracked by this type will be removed from the opening balance.

At this point, your class diagram may appear as follows:



This would be acceptable as an early analysis class diagram, but would likely lead to problems when implemented. Consider the following code snippet, showing a possible implementation of the *calculateBalance* method, when using the design implied above:



This code, as written, violates at least one of the object oriented design principles discussed earlier. Think through the design principles described in this learning outcome and pick at least one that you think this code is violating. Write down why you think that principle is being violated.

## Refactoring

Refactoring is the process of changing the design of a software system to improve its modularity or flexibility without impacting its existing functionality. In this case, the code presented on the previous page would likely work fine in a basic implementation. However, it violates several important principle of software design, which could make it more difficult to expand upon in the future. The two principles being violated are:

1. **Single responsibility principle.** Classes should have one, and only one, reason to change. In this case, what would happen if we added another type of Transaction? If the client were to request that an "interest" transaction type be added, which calculates based on the current balance, a new section of code would have to be added to *calculateBalance* to implement that functionality.

Account has more than one reason to change - instead of only tracking and operating on Account specific data, it must be changed when requirements around Transactions are changed, and becomes less flexible and the system more difficult to expand.

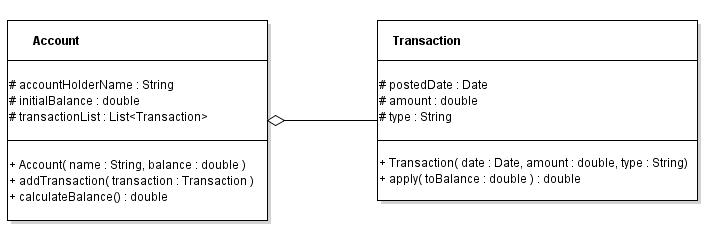
1. **Tell, don't ask.** In the code sample above the *calculateBalance* method of Account is performing an operation using data that another class (Transaction) contains. It is making decisions about the meaning of a Transaction outside of the transaction class. This creates a strong bond between Account and Transaction that need not exist.

The Account class must currently know exactly what values for the "type" property of Transaction have what meaning (debit subtracts, credit adds). A better design would delegate the responsibility for applying a transaction to a balance to the Transaction class instead.

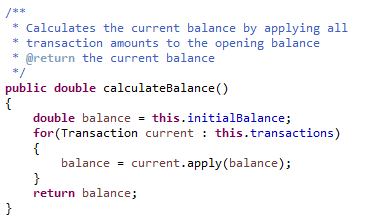
How could these issues be solved? One possibility would be to move the decision around how a Transaction modifies a balance into the Transaction class itself. This would, at least, solve the current issue with "Tell, don't ask". Before continuing to the next topic, try to refine your analysis class diagram so that the behaviour of applying a transaction to a balance is implemented as part of the Transaction class.

## Refining Transaction

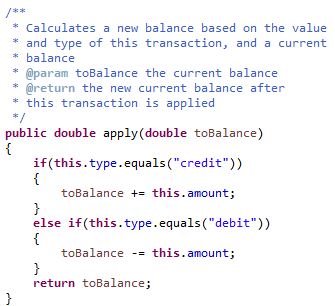
Below is a refactored class diagram for the system. Note that access modifiers and types have been added to improve clarity and show encapsulation.



At this point, the code for the *calculateBalance* method of Account could be envisioned as follows:



With the following for the *apply* method of Transaction:



The design of this system has improved, but it is still not completely ideal. Consider the Single Responsibility Principle: classes should have only one reason to change. At this point, Account will no longer need to change as new types of transactions are added. However, Transaction will have to change if new types are created, and still relies on the value of the "type" parameter passed in from outside.

What would happen if another developer created a Transaction with a new value for the type parameter, but did not correctly create, or omitted, the "if" statement to handle it within *apply*?

We need to consider that this class design can be further modified to improve its modularity and ease of expansion.

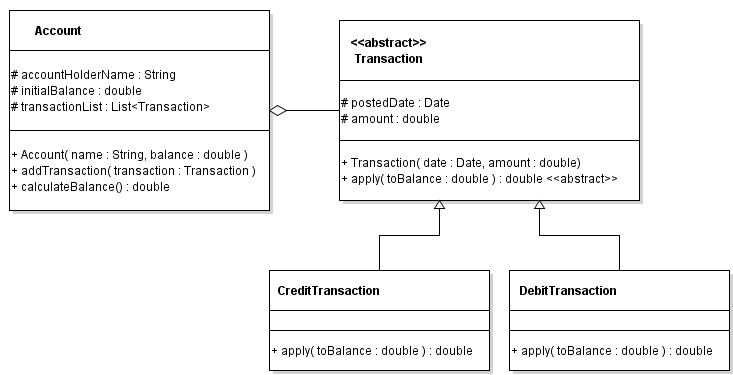
Recall the "Open-Closed Principle" which states that classes should be open to expansion, but closed to modification. In other words, when new or changed behaviour is added to a certain type, the behaviour should be implemented in a class which extends the original class and overrides the specific behaviour. In the case of Transaction, the requirements described two subtypes of Transaction with specific behaviour. This is an indication that Transaction should be represented as an inheritance relationship - Transaction will define the behaviour common to all Transactions, while subclasses such as CreditTransaction and DebitTransaction should implement the behaviours specific to their situation.

Moving to this design will also allow more flexibility for future expansion. Rather than having to modify the Transaction class itself when a new Transaction behaviour is added, a new subclass can be created which implements the behaviour.

Before moving on to the next section, modify your class diagram to create an inheritance relationship between a Transaction parent class and two child classes, DebitTransaction and CreditTransaction. Specify in the diagram what attributes and methods each of the classes will contain.

## Final Model

An example design class diagram for a model which closely follows the principles of object oriented design could be as follows.



In the final model, the Account class is only bound to the Transaction parent class, which itself is abstract. Transactions can be added to Account which are of any Transaction sub-type, due to the polymorphic behaviour of object oriented programming languages (CreditTransaction is-a Transaction). When the previous *calculateBalance* method is called, it continues to call the *apply* method of each Transaction instance, just as it did before. Rather than Transaction handling the calculation itself, the polymorphic behaviour of the language calls the *apply* method of the subclass instead. Since the type-specific behaviour is now implemented in the Transaction child classes, the "type" parameter to the Transaction class is no longer necessary.

The final code for this project can be found on Dropbox. As an exercise, add a new Transaction subclass, "InterestTransaction". When its *apply* method is called, have it return a new balance which is the existing balance plus a percentage of the existing balance. The percentage should be passed in to InterestTransaction as a constructor parameter. Modify your class diagram before beginning, then write the class and code to create an InterestTransaction to test its behaviour.

Account.java

Transaction.java

DebitTransaction.java

CreditTransaction.java

1. Robert Martin in "Clean Code". http://objectmentor.com/resources/articles/srp.pdf [↑](#footnote-ref-2)
2. Robert C Martin, Agile Software Development Principles, Patterns and Practice [↑](#footnote-ref-3)