Object-Oriented Design

COURSE NOTES

Version 0.1.0

TABLE OF CONTENTS

COURSE OVERVIEW

This course examines object-oriented design as part of the foundation for becoming an experienced software architect. The material starts with an introduction to object-oriented thinking, examining the role of design and quality attributes in the software development process. This will also include an overview of the Class Responsibility Collaborator cards technique.

The course then transitions into object-oriented analysis and design. Building on the material presented in the first module, you will learn about object-oriented modelling, particularly how design principles are communicated and expressed in Java and Unified Modelling Language (UML).

The last section of this course will focus on the more complex aspects of design principles that support object-oriented design, which must be understood to create flexible, reusable, and maintainable software. Some more specialized UML diagrams are also explained.

This course assumes basic understanding of the programming language Java.

Upon completion of this course, you will be able to:

1) Explain object-oriented analysis and design.
2) Engage in object-oriented modelling.
3) Explain design principles for object-oriented programming.

MODULE 1: OBJECT-ORIENTED ANALYSIS AND DESIGN

Upon completion of this module, you will be able to:

(a) Explain object-oriented thinking.
(b) Understand the role of design and communication in the software process as well as the link between these concepts and the use of diagrams.
(c) Design for quality attributes.
(d) Model Class Responsibility Collaborator (CRC) cards.

Object-Oriented Thinking

Object-oriented modelling is a major topic in this specialization. Before we can discuss this topic in depth, it is important to learn how to think about problems and concepts as object oriented.

You probably associate the term "object-oriented" with coding and software development. While that is true, the notion of being object-oriented can apply outside of the role of a developer. Object-oriented thinking involves examining the problems or concepts at hand, breaking them down into component parts, and thinking of those as objects. For example, a tweet on Twitter or a product on an online shopping website could be considered objects.

When translated to object-oriented modelling, object-oriented thinking involves representing key concepts through objects in your software. Note that concepts are broad in nature. Even instances of people, places, or things can be distinct objects in software.

Objects may have specific details associated with them, which are relevant to users. For example, a person object may have details such as name, age, gender, and occupation. A place object may have a size, or a name. An inanimate object may have dimensions or a colour.

Objects might also have behaviours or responsibilities associated with them. For example, a person may have associated behaviours such as sitting down or typing. An electronic device may be responsible to power on or off, or to display an image.

By using objects to represent things in your code, the code stays **organized**, **flexible**, and **reusable**.

- Objects keep code organized by putting related details and specific functions in distinct, easy-to-find places. In the above examples, the details of the objects stay associated with the objects themselves.
- Objects keep code flexible, so details can be easily changed in a modular way within the object, without affecting the rest of the code. In the above example of a person object, a person's details such as occupation may change, and not affect the rest of the code.
- Objects allow code to be reused, as they reduce the amount of code that needs to be created, and keeps programs simple.

Objects are self-aware in software production, even if they are inanimate objects. For example, a mobile phone "knows" its specifications. Similarly, in object-oriented modelling, an object such as a chair would know its dimensions and location.

In object-oriented thinking, often everything is considered an object, even if animate or live. And objects are all self-aware, even if inanimate.

It is good practice to prepare for **object-oriented design** by accustoming yourself to thinking about the world around you in terms of objects, and the attributes those objects might have.

DID YOU KNOW?

A good exercise to help you start object-oriented thinking is to look at the room around you and identify what might be objects. For example, you might see a computer, a person, or some kind of furniture. These all have their own specific details and may have behaviours or responsibilities that would be relevant to a user of that object. For a computer system, details might include operating software or display resolution. Responsibilities might include turning on and off, or displaying a screen.

Even the room itself is an object! It may have a seating capacity, a room number, or a purpose that provide specific details and responsibilities to the room as an object.

What objects are around you? What kinds of details and behaviours might they have?

Design in the Software Process

When software is developed, it generally goes through a **process**. In simple terms, a process takes a problem and creates a solution that involves software. A process is generally iterative. These iterations consist of taking a set of requirements based on the identified problem(s) and using them to create **conceptual design** mock-ups and **technical design** diagrams, which can then be used to create a working software **implementation**, which must also pass testing. This process is repeated for each set of requirements, eventually creating a complete solution for the project.

Many projects fail when this process is skipped over, especially when work immediately begins with coding, and there is a lack of understanding of the requirements and design.

It is important to allot time to form the requirements and design, even if they are not perfectly established. The work of coding relies on certain assumptions, and it can be difficult to change those assumptions once coding has begun. Requirements and design activities help you to understand what assumptions you need so that you create the right product.

DID YOU KNOW?

In a survey from The Standish Group, 13% of respondents noted incomplete requirements impaired their projects! Diving straight into implementation work is a leading cause of project failure.

Let us briefly examine the steps of **requirements** and **design** activities in the software process. These steps will require you to think like an architect, so you will need to consider the structure and behaviour of your software. By the end of this lesson you will understand that design work involves outlining a solution and it may include evaluating different alternatives.

Requirements

Requirements are conditions or capabilities that must be implemented in a product, based on client or user request. They are the starting point of a project–you must understand what your client wants.

However, in order to elicit requirements, it is important to ask for more than simply the client's vision. Instead, eliciting requirements involves actively probing the client vision, clarifying what may not have been told, and asking questions about issues the client may not have even considered. This allows you to understand the full scope of what you to build and what your client wants in a product before you actually start coding.

In addition to establishing specific needs for the project, it is also important to establish potential trade-offs the client may need to make in the solution. For example, a client may decide to sacrifice a feature in order to ensure that a program runs faster, if speed is an important need.

Once requirements and trade-offs are established, they may serve as the foundation for design.

To better understand requirements, imagine you are an architect building a house. Requirements allow you to understand what a homeowner wants in a house before you start building. The homeowner may tell you what rooms they want, but you may need to ask follow-up questions about what rooms may be missing from their list, what size the house and rooms might be, any constraints on the house based on restrictions, how clients want rooms to be placed, or what direction the house should face. These help you better understand what you will be building.

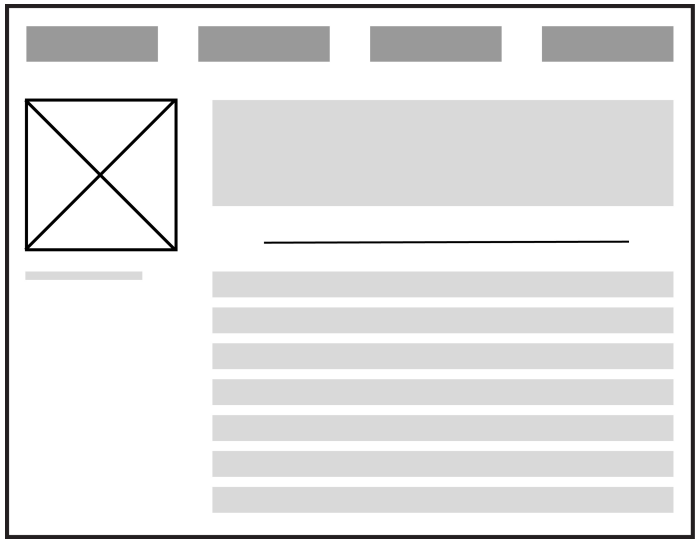
Design

When the initial set of requirements has been created, the next step in the process is to produce a conceptual design and a technical design. This results in the creation of two different kinds of artifacts: **conceptual mock-ups** and **technical diagrams**.

Conceptual Design

Conceptual designs are created with an initial set of requirements as a basis. The conceptual design recognizes appropriate **components**, **connections**, and **responsibilities** of the software product. However, more specific technical details are deferred until the technical design. Conceptual designs outline the more high-level concepts of your final software product.

Conceptual designs are expressed or communicated through **conceptual mock-ups**. These are visual notations that provide initial thoughts for how requirements will be satisfied. Mock-ups for software involving user interfaces are often presented as wireframes, which are a kind of blueprint or basic visual representation of the product. See the example below of a wireframe mock-up for a web page. Whether for user interfaces or for the software product itself, conceptual mock-ups can be hand-drawn sketches or drawings made using computer tools. Conceptual mock-ups help to clarify design decisions with clients and users by providing a simple way to illustrate and discuss how a product will work.



Mock-ups illustrate major components and connections, or relations between the components. Once you start to create a mock-up, you may more easily see what components are missing or may not work. These flaws would require further clarification with your client or involve additional conceptual design work. Every component also has a task it needs to perform. This is known as its **responsibility**. Mock-ups do not outline technical details, because that falls outside the scope of conceptual design.

For example, let us return to the metaphor of building a house. The components for the architectural example of building a house might be: the lot the house will be built on, the house, and rooms inside the house. Connections might be how rooms are accessible to each other. The house has the responsibility of providing enough power, water, and support for all the components within it. Rooms in the house, such as the kitchen, may also have responsibilities, such as providing space for storing kitchenware, appliances, food supplies, plus power and water for meal preparation. However, specifics about wiring and plumbing are not mentioned in the conceptual design. These technical details cannot be fully addressed until the conceptual mock-ups are completely understood. For example, the size of the electrical distribution panel for the house will require adding up the power requirements of each of the rooms.

Best practice is to form the conceptual design before moving on to the technical design. The clearer the conceptual design, the better the technical design, and the more likely your software will be built right.

Technical Design

Technical designs build on conceptual designs and requirements to define the technical details of the solution. In the conceptual design, the major components and connections as well as their associated responsibilities of the software being developed are outlined. The technical design brings this information to the next stage—it aims to describe how these responsibilities are met. The technical design is not finished until each component has been refined to be specific enough to be designed in detail.

In order to accomplish this, technical designs begin by splitting components into smaller and smaller components that are specific enough to be designed in detail. By breaking down components more and more into further components, each with specific responsibilities, you get down to a level where you can do a detailed design of a particular component. The final result is that each component will have their technical details specified.

In order to communicate technical design, **technical diagrams** are used. Technical diagrams visualize how to address specific issues for each component, as conceptual mock-ups are generally not specific enough to capture this information. There are many different technical diagrams that can be used to describe the structure and behaviour of components, which will be addressed later on in this specialization. Technical diagrams therefore help co-ordinate development work.

To continue with the architectural example used throughout this lesson, imagine having to design a kitchen. A kitchen is a component of a house on its own, but it will require further smaller components, such as flooring. The technical design may indicate that the flooring will need to be made of a material that is easy to clean, particularly if the client plans on doing a lot of cooking—cooking can be a messy business!

Compromise in Requirements and Design

When in the design phase, there may need to be compromises in creating an acceptable solution. Constant communication and feedback is key to creating the right solution that meets client needs and works within any restrictions that may exist.

Drawing on the architectural example used throughout this lesson, imagine the client would like an open kitchen in their house that has no obstructions between it and the dining room. But what if a post and beam is needed in that area to support the second floor of the house? The homeowner and the project will need to work out a compromise in that situation.

Designs will need to be reworked if components, connections, and the responsibilities of the conceptual design prove impossible to achieve in the technical design, or if they fail to meet requirements. It is important to continually check with clients that conceptual mock-ups capture what they want. It is easier to re-design in the planning stages, than once coding has started.

Larger systems generally require more design time. There are more components, connections, and responsibilities to keep track of in larger systems. And as these components themselves are large, they may need to be refined down to smaller components before their design can be detailed.

Once a feasible design has been agreed upon, the technical diagrams become the basis for constructing the intended solution. Components at this stage may be refined enough to become collections of functions, classes, or other components. These pieces become a more manageable problem that developers can individually implement.

There are many design techniques that may be used to get the most out of the design process. The rest of this specialization will examine those techniques.

Design for Quality Attributes

When developing software, it is important to take a broad view on how to achieve the desired requirements. This lesson examines how competing ideals, roles and perspectives, potential trade-offs, and project realities need to be taken into account and balanced in **software design**.

Trade-offs

This course has reviewed the importance of requirements and design in creating software. Sometimes, there are restrictions on design that require compromise. Besides software requirements based on desired functionality, there are also **quality attributes** to define how well this functionality must work. But your decisions may also involve trade-offs in different quality attributes, such as performance, convenience, and security, and these attributes need to be balanced.

For example, it is important to consider how quality attributes can compete in a proposed solution under different situations. Then, taking this into account and weighing it against the requirements of the product, a suitable compromise can be determined. This balancing act is an ongoing constant for **software architecture**. Software architects must find the best balance between quality attributes—often by evaluating which one is more important. Deadlines can also influence what is feasible to do within a certain time frame.

Let's consider, for example, designing a front door to a house. Security is a quality attribute that might be important, but if you add too many locks to the door, it may be difficult to open easily and will become inconvenient to use. A good design should balance security with convenience and performance.

Context and Consequences

Context provides important information when deciding on the balance of qualities in design. For example, software that stores personal information, which the public can access, may have different security requirements than software that is only used by corporate employees. In order to establish context, it is important to talk to stakeholders.

Software design also must consider the consequences. Sometimes, choices made in software design have unintended consequences. For example, an idea that seems to work fine for a small amount of data may be impractical for large amounts of data.

A good practice is to seek other perspectives on technical designs for a more well-rounded implementation. This can be done by asking other developers for their opinion, or by having a design review session. It is also good practice to test a system carefully before fully implementing a system. During the design process, you might consider prototyping alternative ideas and running tests to see what works best. Tests can help catch unintended consequences. For example, testing with both small and large amounts of data in the above example might reveal the system limitations.

Satisfying Qualities

Qualities are achieved through satisfying functional and non-functional requirements, which in turn are the basis for the design process.

Functional requirements describe what the system or application is expected to do. A key quality to achieve by satisfying a functional requirement is that of correctness. For example, if you are designing a music app, the app must be able to download and play a song. The design needs to be able to outline a solution that correctly meets this requirement.

Non-functional requirements specify how well the system or application does what it does. Non-functional requirements to satisfy might include performance, resource usage, and efficiency; these requirements can be measured from the running software. For example, the music app may have non-functional requirements to download music only to a certain memory limit. Other qualities that software often satisfies in non-functional requirements include **reusability, flexibility,** and **maintainability**. This helps inform how well the code of software can evolve and allow for future changes.

Requirements are often incomplete at first, but are resolved with further interactions with clients and end users.

> ***Instructor's Note***:
> *If you are interested in learning more about functional and non-functional requirements, see the requirements course of the Coursera specialization offered by the University of Alberta on* ***Software Product Management***.

Functional and non-functional requirements are important to satisfy, but there may be important constraints and limitations that will lead to compromises. For this reason, it is important to communicate and determine what is acceptable to stakeholders. Consider this example: all cars meet the functional requirement of providing transportation; however, non-functional requirements and the emphasis on certain qualities can vastly change the final product—different accelerations, handling, weight, and fuel economy can make the difference between a minivan and a sports car.

Reviews and tests should also be used to verify that required qualities on design and software implementation are satisfied. Some qualities may also be validated with feedback from end users.

> ***Instructor's Note***:
> *If you are interested in learning more about reviews, see the reviews course of the Coursera specialization offered by the University of Alberta on* ***Software Product Management.***

Compromise

In addition to balancing qualities and meeting functional requirements when designing software, it is important to consider multiple perspectives. Software must satisfy qualities that matter to users as well as developers. In other words, how the software structure is organized may affect the quality of performance, as understood by users, and the qualities of reusability and maintainability, as understood by developers.

Below are some common trade-offs in qualities for software design:

- Performance and maintainability – High performance code may be less clear and less modular, making it harder to maintain. Alternately, extra code for backward compatibility may affect both performance and maintainability.
- Performance and security – Extra overhead for high security may lessen performance.

Balance between qualities must be understood and taken into account during design. It is important to prioritize and understand what qualities are needed. A good question to ask to help you determine what compromises can be made is: Is there a way to cut back on a certain quality to balance another?

DID YOU KNOW?

Some common qualities to take into account in software design include: performance, maintainability, security, and backwards compatibility.

It is also important to consider the constraints by project realities on your project. To develop the product, qualities must be balanced with resources available such as cost, time, and manpower.

Class Responsibility Collaborator

So far, this module has reviewed the process of eliciting requirements and using conceptual design to gather initial thoughts on how to satisfy those requirements in software development. Components, connections, and responsibilities for some requirements are established during this stage.

This module has also examined how components and connections are refined through the technical design process, and by taking quality attributes into account, in order to establish technical details. This allows components and connections to be more easily implemented.

This next lesson presents an important technique to help represent the components, responsibilities, and connections at a high level when forming the conceptual design. This technique is the use of Class, Responsibility, Collaborator (CRC) cards. CRC cards help record and organize components into classes, identify component responsibilities, and determine how they collaborate with each other. Therefore, they also help refine the components in your software design.

CRC Cards

During the process of conceptual design, it is helpful not only to identify components, responsibilities, and connections but also to represent them. One technique is to use **Class, Responsibility, Collaborator (CRC) cards**.

CRC cards are used to record, organize, and refine the components of system design. They can be compared to note cards, which are used to organize talking points. CRC cards are designed with three sections: the **class name** at the top of the card, the **responsibilities** of the **class** on the left side of the card, and the **collaborators** on the right side of the card. See the image below for an example of what a CRC card might look like, about the size of a physical index card.

Class Name	
Responsibilities	**Collaborators**

To keep track of each candidate component and its responsibilities using a CRC card, you place a component's name in the class name section, and the responsibilities in the responsibilities section. Connections are captured in the collaborators section. Connections or collaborators indicate other classes that the class at the top of the card interacts with to fulfill its responsibilities. These steps are repeated iteratively and new cards are created until all the classes, responsibilities, and collaborators are identified for a system.

In system design, CRC cards has a purpose—it forces designers to keep breaking components down into smaller components and classes that can be individually described on a card.

Prototyping and Simulation

The use of CRC cards is a simple system that has many advantages. They are cheap, editable, and widely available. They help sort information into manageable pieces.

A key advantage of using CRC cards is that they allow you to physically reorganize your design. As each of the components are represented by a card, you can move related cards together, or situate cards to suggest relationships. This allows you to theoretically explore how your system will work and to identify any shortcomings in the design.

You can also experiment with moving these cards around in new orders and analyzing the resulting consequences, allowing you to play with alterative designs. This means that CRC cards can be used to prototype and simulate a system for conceptual design.

When you develop designs, these are sometimes referred to as CRC models. CRC cards should be organized by placing closely collaborating components together. This makes it easier to understand the relationships or connections between classes or components.

CRC cards are excellent tools to bring to software development team meetings. All the cards can be placed on the table, and facilitate a discussion or a simulation with the team of how these classes work together with other classes to achieve their responsibilities. This allows you to both visually explain your system and gain potential input from other parties.

CRC cards are useful tools, but they are most powerful when used for prototyping and simulation for conceptual design. Many other techniques have been developed to help you design more effectively. The rest of this specialization will focus on some of these various design techniques.

MODULE 2: OBJECT-ORIENTED MODELLING

Upon completion of this module, you will be able to:

(a) Describe issues in creating models for design.
(b) Understand how programming languages evolved toward object orientation.
(c) Explain the four major design principles used in object-oriented modelling:
    a. Abstraction
    b. Encapsulation
    c. Decomposition
    d. Generalization
(d) Express the above design principles in using UML class diagrams and Java code.
(e) Explain and express implementation inheritance.
(f) Explain and express interface inheritance.

The previous module in this course provided an introduction to the importance of design in the software development process, and ended with explaining the advantage of using CRC cards to complete a conceptual design.

This module will explore object-oriented modelling even further. It will begin by examining modelling problems and how programming languages evolved towards object orientation. Then, the four major design principles of abstraction, encapsulation, decomposition, and generalization will be discussed. These principles help in problem solving and lead to developing software that is flexible, reusable, and maintainable. They are key principles to follow for developing a good design for your software.

This module will also explore how to express design structure in Java code and UML class diagrams using the principles of abstraction, encapsulation, and decomposition. Finally, it will discuss implementation and interface inheritance within the design principle of generalization.

Creating Models in Design

It is important when working on a software development project not to jump right into creating code to solve the problem. Instead, making the right product involves understanding the full requirements of your product and using good design.

The design step falls between understanding your requirements and building the product. It iteratively deals with both the problem space and the solution space. The design should also present and describe concepts in a way that users and developers both understand, so they may discuss using common terms.

Design is such an important step in software development, and there have been many approaches developed over time to help make this process easier. For example, some design strategies and programming languages have been created for specific kinds of problems.

One approach to help make the design process easier is the object-oriented approach. This allows for the description of **concepts** in the problem and solution spaces as **objects**—objects are a notion that can be understood by both users and developers, because object-oriented thinking applies to many fields. This shared knowledge makes it possible for users and developers to discuss elements of complex problems. Object-oriented programming with object-oriented languages is therefore a popular means for solving complex problems.

A good design does not just jump from a concept within the problem space to dealing with it in the solution space. Object-oriented design is no exception. As reviewed in Module 1, object-oriented design consists of:

- **Conceptual design** uses **object-oriented analysis** to identify the key objects in the problem and breaks down the problem into manageable pieces.
- **Technical design** uses object-oriented design to further refine the details of the objects, including their attributes and behaviours, so it is clear enough for developers to implement as working software.

These design activities happen iteratively and continuously.

The goal during software design is to construct and refine "**models**" of all the objects of the software. Categories of objects involve:

- **entity objects**, where initial focus during the design is placed in the problem space
- **control objects** that receive events and co-ordinate actions as the process moves to the solution space
- **boundary objects** that connect outside services to your system, as the process moves towards the solution space

Software models help you understand and organize the design process for the objects. Design principles and guidelines are applied to complex problems: to simplify objects in the model and break them down into smaller parts and to look for commonalities that can be handled consistently. Models should be continuously critiqued and evaluated to ensure the original problem is addressed and qualities such as reusability, flexibility, and maintainability are satisfied. Models also serve as design documentation for your software. In fact, models are often mapped to skeletal source code, particularly for an object-oriented language like Java.

Software models are often expressed in a visual notation, called Unified Modelling Language (UML). **Object-oriented modelling** has different kinds of models or UML diagrams that can be used to focus on different software issues. For example, a structural model might be used to describe what objects do and how they relate. This is analogous to a scale model of a building, which is used in architecture.

Now that you have an understanding of the roles models play in design and of the relationship between models and coding languages, the next lesson will turn to reviewing the history of programming languages.

Evolution of Programming Languages

Language is the word that we use to describe a system for communicating thoughts and ideas with each other. Writing, reading, speaking, drawing pictures, and making gestures are all part of language! Languages must be continually evolving in order to stay "alive" and be used by people.

Programming languages are no exception to this, and just like traditional languages, they have evolved over time. Often, programming languages evolved to provide solutions or more effective solutions to needs or problems that the current programming languages cannot meet. New languages or ideas may also arise to address new data structures. The ideas used in computer languages caused shifts in **programming paradigms**.

DID YOU KNOW?

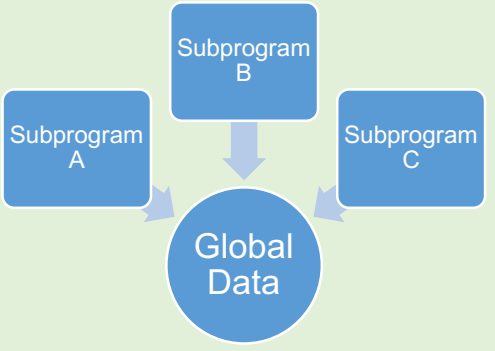
An example of design strategies and programming languages suited for specific kinds of problems that you may be familiar with is top-down programming.

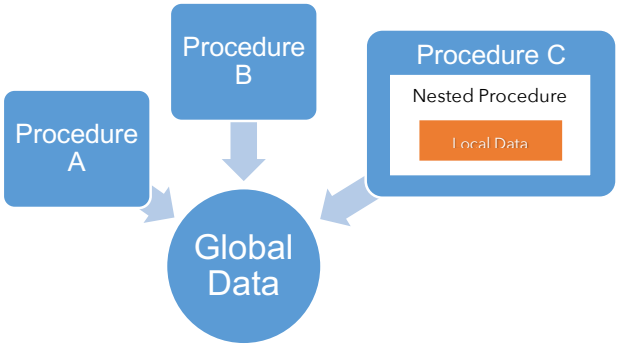
Top-down programming is generally used to solve data- processing problems. This design strategy consists of mapping processes in the problem to routines to be called, beginning with the "top" process. Generally, this design is expressed through a tree of routines.

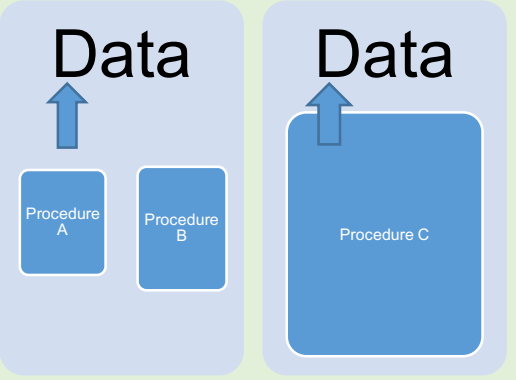
These routines would be implemented in a programming language that supported subroutines.

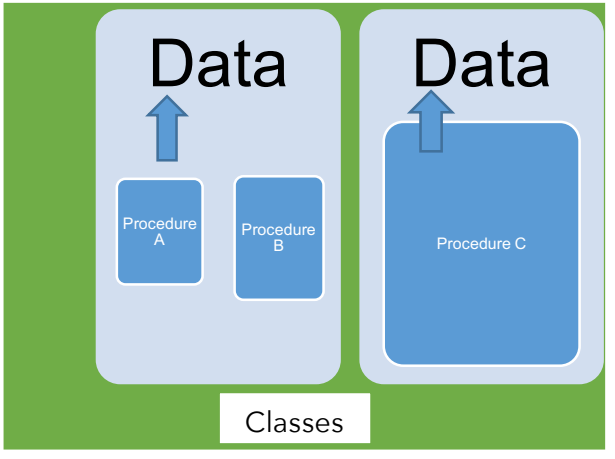
It is important to know the history of programming paradigms. As a software developer, you may still encounter systems that use older languages and design paradigms. As well, although object-oriented programming is a powerful tool, there may be problems that are best or more efficiently solved with another paradigm. Finally, it is important to understand this history, as new languages may not force new structures but only modify existing ones. Some old ways of doing something or old paradigms may be expanded on so much that the new structures may be difficult to recognize. Knowledge of the past may help.

Below is a table summarizing major programming paradigms in the history of programming languages.

Programming Language	Time period	Solutions afforded by Programming Language	Unresolved issues of Programming Language
COBOL **Fortran**	1960s	COBOL and Fortran followed an **imperative paradigm** that broke up large programs into smaller programs called subroutines. As computer processing time was costly, it was important to maximize processing performance. To solve this problem, **global data** was used so that data was located all in one place in the computer's memory and accessible anywhere for a program. This meant that subroutines only had to go to one place to access variables.  	If changes are made to the data, then subroutines might run into cases where the global data is not what was expected. Better data management is needed to avoid these problems.

Programming Language	Time period	Solutions afforded by Programming Language	Unresolved issues of Programming Language
Algol 68 **Pascal**	Early 1970s	In the 1960s, global data was used. However, any changes to the data may result in issues for the subroutines. The solution introduced the idea of scopes and **local variables** – subroutines or procedures could each have their own variables.  These languages supported the use of **abstract data type**, which is defined by the programmer and not built into the language. This is a grouping of related information that is denoted with a **type**. This allows information to be organized in a meaningful way. By having data bundled and passed into different procedures through the use of data types, this means that a procedure can be the only one that modifies a piece of data. There no longer needs to be a worry that data will be altered by another procedure.	Towards the mid-1970s, computer processing time became less expensive. At the same time, human labour was more expensive and became the more time-consuming factor in software development. The advances in computer processing allowed more complex problems to be asked of computers. But it also meant that software was quickly growing, and having one file to maintain programs was difficult to maintain.

Programming Language	Time period	Solutions afforded by Programming Language	Unresolved issues of Programming Language
C **Modula-2**	Mid-1970s	By the mid-1970s computers were faster and able to tackle more complex problems. However, this meant that the programs of the past were quickly becoming too big to maintain. This led to new languages, that provided a means to organize programs into separate files, and allow developers to more easily create multiple, but unique, copies of abstract data types.  For example, in the programming language C, each file contained all the associated data and functions that manipulated it, and declared what could be accessed through a separate file called a **header file**.	It is not easy for an abstract data type to inherit from another in these languages. This means that although as many data types as wanted can be created, one type cannot be declared an extension of another.

Programming Language	Time period	Solutions afforded by Programming Language	Unresolved issues of Programming Language
Object-Oriented Programming (Java, C++, C#, etc.)	1980s to present	Although programs were become easier to manage through abstract data types, there was still no way to for data types to inherit from each other. The concepts of object-oriented design became popular during this time period as a solution to these problems. Object-oriented design seeks to: • make an abstract data type easier to write • structure a system around abstract data types called classes • introduce the ability for an abstract data type to extend another through a concept known as inheritance 	Object oriented programming is the predominant **programming paradigm** now.

Programming Language	Time period	Solutions afforded by Programming Language	Unresolved issues of Programming Language
		Under this paradigm, software systems can be built of entirely abstract data types. This allows the system to mimic the structure of the problem—in other words, the system can represent real-world objects or ideas more accurately. Class definition files in object-oriented programming replace the files in C and Modula-2. Each class defines a type with associated data and functions. These functions are also known as **methods**. A class acts like a factory, making individual objects all of a specific type. This allows data to be compartmentalized and manipulated into its own separate classes.	

Four Design Principles

As described in the first lesson of this module, object-oriented programming allows you to create models of how objects are represented in your system. However, to create an object-oriented program, you must examine the major design principles of such programs. Four of these major principles are: **abstraction**, **encapsulation**, **decomposition**, and **generalization**.

Abstraction

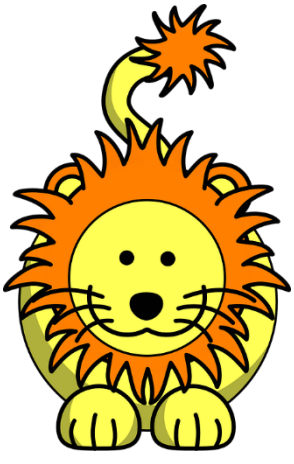
Abstraction is one of the four major design principles that will be examined in this lesson. Abstraction is one of the main ways that humans deal with complexity. It is the idea of simplifying a concept in the problem domain. Abstraction breaks a concept down into a simplified description that ignores unimportant details and emphasizes the essentials needed for the concept, within some context.

An abstraction should follow the **rule of least astonishment**. This rule suggests that essential attributes and behaviours should be captured with no surprises and no definitions that fall beyond its scope. This prevents irrelevant characteristics from becoming part of an abstraction and helps to ensure that the abstraction makes sense for the concept's purpose.

Program constructs includes elements such as functions, classes, enumerations, and methods. In object-oriented modelling, abstraction pertains most directly to the notion of a class. When abstraction is used to determine the essential details for some concept, those details may be defined in a class. Any object created from a class has the essential details to represent an instance of some concept, but it may have some individual characteristics as well. Think of a cookie cutter used to create gingerbread men. Each instance of a cut cookie belongs to the class of "gingerbread men" and share essential characteristics such as head, arms, and legs, even if they are decorated differently.

Context or a specific perspective is critical when forming an abstraction. This is because context might change the essential characteristics of a concept. For example, consider the essential characteristics of the concept of a person. This can be hard to understand without context, as this concept is vague and the person's purpose is unknown. But, in a gaming app, the essential characteristics of a person would be in the context of a gamer. In a running exercise app on the other hand, the essential characteristics of a person would be in the context of an athlete. It is up to the designer to choose the abstraction that is most appropriate to the context of the software development, and the context must be understood before creating an abstraction.

The essential characteristics of an abstraction can be understood in two ways: through basic **attributes** and through basic **behaviours** or **responsibilities**.



Basic attributes are characteristics that do not disappear over time. Although their values may change, the attributes themselves do not. For example, the concept of a lion may have an age attribute. That value may change, but the lion always has an age attribute.

In addition to basic attributes, an abstraction describes a concept's basic behaviours. A lion may have behaviours such as hunting, eating, and sleeping. These are also responsibilities that the lion abstraction does for its purpose of living.

An abstraction, as explained above, should only convey a concept's essential attributes and behaviours. Context helps determine what is relevant. For example, when considering the lion in a hunting setting, it is irrelevant to consider what position the lion prefers to sleep in. If context changes, the right abstraction may change as well.

There are many benefits from the principle of abstraction. It helps to simplify class designs, so they are more focused, succinct, and understandable to someone else viewing them. As abstractions rely strongly on context or perspective, it is important to carefully consider what is relevant. Likewise, if the purpose of the system being built, or if the problem being solved changes, it is important to re-examine your abstractions and change them accordingly.

Encapsulation

Encapsulation is the second major design principle that will be examined in this lesson. This principle involves a concept that allows something to be contained in a capsule, some of which you can access from the outside and some of which you cannot.

There are three ideas behind encapsulation. These are:

- The ability to "bundle" attribute values (or **data**) and behaviours (or **functions**) that manipulate those values, into a self-contained object.
- The ability to "expose" certain data and functions of that object, which can be accessed from other objects, usually through an **interface**.
- The ability to "restrict" access to certain data and functions to only within the object.

"Bundling" occurs naturally when a class is defined for a **type** of object. The principle of abstraction helps determine what attributes and behaviours are relevant about a concept in a determined context. The principle of encapsulation takes this a step further and ensures that these characteristics are bundled together in the same class.

Encapsulation therefore allows distinct objects created from a particular class to have their own data values for the attributes and exhibit resulting behaviours. This makes programming much easier, as the data and the code that manipulate that data are located in the same place.

An object's data should only contain what is relevant for that object. For example, a lion object "knows" what food it hunts but does not know what animals live in a different continent, because that is not relevant data. A class therefore only knows what attributes or data is relevant to it.

A class also defines behaviours through methods. Methods manipulate the attribute values or data in the object to achieve the actual behaviours. Certain "methods" can be exposed or made accessible to objects of other classes. This provides an interface to other objects to use the class.

Integrity and Security

As one of the ideas of encapsulation is restricting access to certain data and functions to only within an object, this naturally links encapsulation to **data integrity** and the security of sensitive information.

If certain attributes and methods are restricted from outside access, except through specific methods, then the data cannot be changed through variable assignments. This prevents assumptions or dependencies from breaking for the data within an object.

Likewise, restriction of access helps keep sensitive information from being revealed, even from queries that rely on sensitive data to provide answers.

Changeable Implementation

Encapsulation is also a useful principle for implementing software changes. As the ability to "expose" data is separate from the "bundle" of attributes itself, this means that the implementation of attributes and methods can change, but the accessible interface of a class can remain the same. Users accessing or querying the class do not need to worry about how the implementation works behind the interface—they will still use the same means to access information.

Black box

The idea in encapsulation that supports the line of thinking in changeable implementation is also tied to a concept known as **black box** thinking. The computation steps taken within a class never need to be known by any other class, as long as they are able to access the interface. A class is therefore like a black box that you cannot see into for details about how attributes are represented or how methods compute their results. What happens in the "box" to achieve an expected behaviour doesn't matter, as long as it is possible to provide inputs and obtain outputs by calling methods.

Encapsulation achieves an **abstraction barrier** through black box thinking where the internal workings of a class are not relevant to the outside world. This results in an abstraction that reduces complexity for users of the class.

Encapsulation also increases reusability because of black box thinking. Another class only needs to know the right method to call to get a desired behaviour, what arguments to supply as inputs, and what appears as outputs or effects. In other words, encapsulation keeps software modular and easy to work with. Classes are easy to manage, as their internal behaviour is not relevant to other classes, as long as they can interface together.

Decomposition

Decomposition is the third major design principles that will be examined in this lesson. It consists of taking a **whole** thing, and dividing it into different **parts**. Alternately, decomposition can also indicate taking separate parts with different functionalities and combining them to create a whole. Decomposition allows problems to broken into smaller pieces that are easier to understand and solve.

The general rule for decomposition is to look at the different responsibilities of a whole and evaluate how the whole can be separated into parts that each have a specific responsibility. Each of these parts are in fact separate objects that can be created from separate classes in your design. In this way, decomposition is similar to abstraction where you are dividing a whole into objects with essential characteristics.

Each different kind of part within a whole can prescribe a class, so we can keep our parts better organized and encapsulated on their own. The class for the whole object then relates to the classes for the constituent part objects.

The Nature of Parts

A whole might have a fixed or dynamic number of a certain type of part. If there is a fixed number, then over the lifetime of the whole object, it will have exactly that much of the part object. Think of an oven with four burners. The number of burners is fixed for the oven object. Some parts, on the other hand, may have a dynamic number. This means the whole object may gain new instances of those part objects over its lifetime. Think of items of food within a refrigerator object – these might change from day to day.

Note that a part can also serve as a whole, which is made up of further constituent parts. For example, a kitchen is a part of a house. But the kitchen may be made up of further parts, such as an oven and a refrigerator.

Another issue worth noting in decomposition is that whole objects and part objects have lifetimes. Sometimes, these lifetimes are closely related, and the part shares the same lifetime as the whole—one cannot exist without the other. If the temperature cooling gauge for a fridge dies, then the fridge will cease to work. But sometimes, the part and the whole can exist independently and have different lifetimes. For example, if an item of food goes bad in the fridge, the fridge will still continue its function.

Whole things may also contain parts that are shared with another whole at the same time. However, sometimes sharing a part is not possible or intended.

Generalization

Generalization is the final one of the four major design principles that will be examined in this lesson. Generalization helps reduce redundancy when solving problems. It is a common principle used in many disciplines outside of software development.

In coding, algorithmic behaviours are often modelled through **methods**. A method allows a programmer to generalize a behaviour, so the behavior can be applied to different input data. This generality reduces the need to have identical code throughout a program.

In object-oriented modelling, generalization is a main design principle, but beyond creating a method that can be applied to different data, object-oriented modelling achieves generalization by classes through **inheritance**. In generalization we take repeated, common, or shared characteristics between two or more classes and factor them out into another class.

This allows you to have two kinds of classes: a parent class, and a child class. Child classes inherit attributes and behaviours of parent classes. This means that repeated, common, or shared characteristics go into parent classes. Parent classes capture **general ideas** and generally have broader application.

It is possible for multiple child classes to inherit from a single parent class. Those multiple child classes will all receive these common attributes and behaviours, although it is likely that each child class will have additional attributes and behaviours that allow them to be more **specialized** in what they can do.

In standard terminology, a parent class is known as a **superclass** and a child class is called a **subclass**. From the above explanation, we can understand that inheritance allows a superclass to form a generalization and for its subclasses to be more specialized.

Parent classes save time and prevent errors, particularly when they are used for multiple child classes. Without parent classes, systems are not flexible, maintainable, or reusable.

Instructor's Note:

A good tip to remember for naming superclasses and subclasses – although classes can be named after anything you want, it is good practice to name them after things you are trying to model. This makes code easier to understand!

Generalization presents many advantages to object-oriented modelling. Since subclasses inherit attributes and behaviours from superclasses, this means that any changes to the code that is common to both subclasses need only be made once in the superclass. In other words, changes to software are easier to apply and maintain. Another advantage is that subclasses can be easily added without having to recreate all the common attributes and behaviours for them, so software is easier to expand. Generalization provides more robust software solutions and allows for more reusable code because the same blocks of code can be used for different classes.

DID YOU KNOW?

Both methods and inheritance exemplify the generalization design principle through the **D.R.Y.** or **"Don't Repeat Yourself" rule**. Methods and inheritance allow developers to reuse code, resulting in less code and repetition overall.

Design Structure in Java and UML Class Diagrams

The design process, as explained in Module 1 of this course, consists of both the conceptual design and the technical design. Conceptual design, including prototyping and simulating higher-level designs, can be visualized through CRC cards. CRC cards make it easy to communicate with client, and they allow you to create designs without the distraction of code. However, to guide technical design, a more sophisticated technique that can communicate your needs clearly to developers is needed. One technique used for technical design is that of **UML class diagram**, also known as simply a class diagram. These class diagrams provide more detail than CRC cards and allow for easier conversion to classes for coding and implementation.

This lesson will look at how the design principles of abstraction, encapsulation, decomposition, and generalization work with Java and UML class diagrams.

Abstraction

The design principle of abstraction allows for the simplification of a concept to its essentials within some context. Abstraction can be applied at the design level using UML class diagrams. The design is eventually turned into code.

CRC cards capture components in systems design. Components can eventually be refined into functions, classes, or collections of other components. As this course uses Java, where abstractions are formed in a class, this lesson will focus on classes.

Let us look at how a CRC card might translate into a class diagram. Below is an example of a CRC card, as abstracted for a food item in the context of a grocery store.

Food	
Know grocery ID Know name Know manufacturer Know expiry date Know price Check if on sale	

Here is the same concept as a class diagram.

Food
groceryID: String name: String manufacturer: String expiryDate: Date price: double
isOnSale() : boolean

Every concept or class in a class diagram is represented with a box, as above. You will notice three sections in the box.

Class Name
Properties
Operations

- The **class name** is the same as the class name in your Java class.
- The **properties** section is equivalent to Java's member variables. This section defines the attributes of the abstraction by using a standard template for variable name and variable type. Variable types can be classes or primitive types.

```
<variable name>:<variable type>
```

- The **operations** section is equivalent to Java's methods. This section defines the behaviours of the abstraction, using a standard template for the operation name, parameter list, and return type.

```
<name>( <parameter list> ) : <return type>
```

In the example above, food objects have a method to return if it is on sale or not. This method has been named "isOnSale". The method will return a boolean to represent if it is on sale. A **Boolean value** is either true or false. The isOnSale operation takes no parameter, so no parameter list is included.

If you were to add a parameter to the operation, such as a date in this case, the parameter would follow the same template as the class diagram's properties. The final section would read:

```
isOnSale(date: Date) : Boolean
```

You will notice that class diagrams distinguish a difference between responsibilities that become properties and responsibilities that become operations, whereas CRC cards list them together. Helping distinguish this ambiguity makes class diagrams easier to translate into code for a programmer.

Drawing on our food example, we can see how easy it is to turn a class diagram into a class in Java.

```
public class Food {
      public String groceryID;
      public String name;
      public String manufacturer;
      public Date expiryDate;
      public double price;

      public boolean isOnSale( Date date ) {

      }
}
```

The class name in the class diagram turns into a class in Java. The properties turn into member variables. Operations become methods. It is possible to use this mapping in reverse to turn code into class diagrams.

> *Instructor's Note: The above example has everything **public**, meaning the member variables and methods can be accessed from any other code besides this class. This assumption applies for now. In later lessons, you will learn about **access modifiers** in Java for more controlled access.*

Encapsulation

The design principle of encapsulation involves three ideas:

- Data and functions that manipulate that data are "bundled" into a self-contained object.
- Data and functions of the object can be exposed or made accessible from other objects.
- Data and functions of the object can be restricted to only within the object.

In a UML class diagram, encapsulation is expressed by having all of the object's relevant data defined in attributes of the class, and by providing specific methods to access those attributes.

UML class diagrams can express encapsulation. The class diagram itself already bundles data and functions in a self-contained object. However, access and restriction (two aspects of visibility) can be represented as well, through the use of symbols – and +. Below is an example of a UML class diagram for a student.

Student
-gpa: float -degreeProgram: String
+getGPA(): float +setGPA(float) +getDegreeProgram(): String +setDegreeProgram(String)

In this example, the attributes gpa and degreeProgram are hidden from **public** accessibility, as indicated by the minus sign (-). In other words, the minus sign indicates that a method or attribute is **private** and can only be accessed from within the class. On the other hand, the operations are public, as indicated by the plus sign (+). In other words, the plus sign indicates that a method can be accessed publicly. In this case, the public methods can be used to manipulate the student's GPA. This prevents the student's GPA attribute from being directly manipulated, and it controls the data is accessed and changed.

Encapsulation in UML class diagrams helps you determine the "gate" to controlling data, by using only public methods to access the data attributes of the class. For every piece of essential data, the use of public methods to access private data creates protection from unexpected direct change of that data. This preserves the **data integrity**.

There are two different kinds of methods typically used to preserve data integrity. These are:

- Getter methods, which are used to retrieve data. These methods typically have the format: get<Name of the attribute>, where the attribute is the value that will be returned through the method. Getters often retrieve private data.

- Setter methods, which are used to change data. These methods typically have the format: set<Name of the attribute>, where the attribute is what will be changed through the method. Setters often set a private attribute in a safe way.

These kinds of methods help ensure that data is accessed in an approved way.

Decomposition

The design principle of decomposition takes a whole thing and divides it into different parts. It also does the reverse, and takes separate parts with different functionalities, and combines them to form a whole. There are three types of relationships in decomposition, which define the interaction between the whole and the parts:

- **Association**
- **aggregation**
- **composition**

All three are useful and versatile for software design. Let us examine each of these relationships.

Association

Association indicates a loose relationship between two objects, which may interact with each other for some time. They are not dependent on each other—if one object is destroyed, the other can continue to exist, and there can be any number of each item in the relationship. One object does not belong to another, and they may have numbers that are not tied to each other.
An example of an association relationship could be a person and a hotel. A person might interact with a hotel but not own one. A hotel may interact with many people.

Association is represented in UML diagrams as below:

The straight line between the two UML objects denote that there is a relationship between the two UML objects of person and hotel, and that relationship is an association. The "zero dot dot star" (0…*) on the right side of the line shows that a Person object is associated with zero or more Hotel objects, while the "zero dot dot star" on the left side of the line shows that a Hotel object is associated with zero or more Person objects.

Association can be represented in Java code as well.

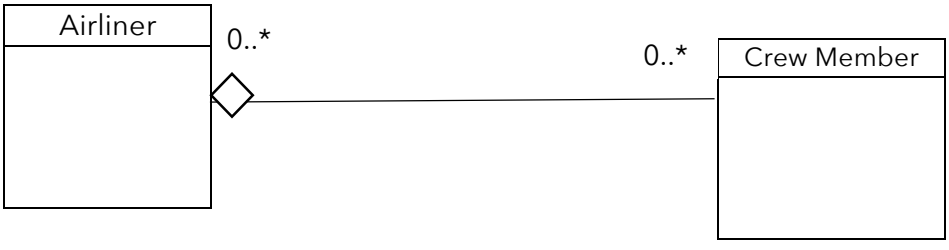
```
public class Student {
      public void play( Sport sport ){
            execute.play( sport );
      }
      …
}
```

In this code excerpt, a student is passed a sport object to play, but the student does not possess the sport. It only interacts with it to play. The two objects are completely separate, but have a loose relationship. Any number of sports can be played by a student and any number of students can play a sport.

Aggregation

Aggregation is a "has-a" relationship where a whole has parts that belong to it. Parts may be shared among wholes in this relationship. Aggregation relationships are typically weak, however. This means that although parts can belong to wholes, they can also exist independently. An example of an aggregate relationship is that of an airliner and its crew. The airliner would not be able to offer services without the crew. However, the airliner does not cease to exist if the crew leave. The crew also do not cease to exist if they are not in the airliner.

Aggregation can be represented in UML class diagrams with the symbol of an empty diamond as below:



In this diagram, a straight line is used again to symbolize a relationship between the Airliner object, and the Crew Member object. "Zero dot dot stars" (0..*) are used to show again that an object might have a relationship with zero or more of the other object. In this case, the "zero dot dot star" on the right side of the line indicates that a Airliner object might have zero or more crew members. The "zero dot dot star" on the left side of the line indicates that a Crew Member object can be had by zero or more Airliner objects. The empty diamond indicates which object is considered the whole and not the part in the relationship.

Aggregation can be represented in Java code as well.

```
public class Airliner {
      private ArrayList<CrewMember> crew;

      public Airliner() {
            crew = new ArrayList<CrewMember>();
      }

      public void add( CrewMember crewMember ) {
            …
      }
}
```

In the Airliner class, there is a list of crew members. The list of crew members is initialized to be empty and a public method allows new crew members to be added. An airliner has a crew. This means that an airliner can have zero or more crew members.

Composition

Composition is one of the most dependent of the decomposition relationships. This relationship is an exclusive containment of parts, otherwise known as a strong "has-a" relationship. In other words, a whole cannot exist without its parts, and if the whole is destroyed, then the parts are destroyed too. In this relationship, you can typically only access the parts through its whole. Contained parts are exclusive to the whole. An example of a composition relationship is between a house and a room. A house is made up of multiple rooms, but if you remove the house, the room no longer exists.

Composition can be represented with a filled-in diamond using UML class diagrams, as below:

The lines between the House object and the Room object indicates a relationship between the two. The filled-in diamond next to the House object means that the house is the whole in the relationship. If the diamond is filled-in, it symbolizes that the "has-a" relationship is strong. The two objects would cease to exist without each other. The one "dot dot star" indicates that there must be one or more Room objects for the House object.

Composition can be represented using Java code as well.

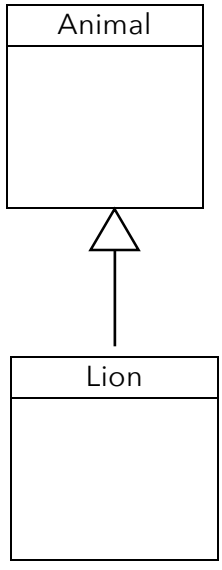
```
public class House {
      private Room room;

      public House(){
            room = new Room();
      }
}
```

In this example, a Room object is created at the same time that the House object is, by instantiating the Room class. This Room object does not need to be created elsewhere, and it does not need to be passed in when creating the House object. The two parts are tightly dependent with one not being able to exist without the other.

Generalization

The design principle of generalization takes repeated, common, or shared characteristics between two or more classes and factors them out into another class, so that code can be reused, and the characteristics can be inherited by subclasses.

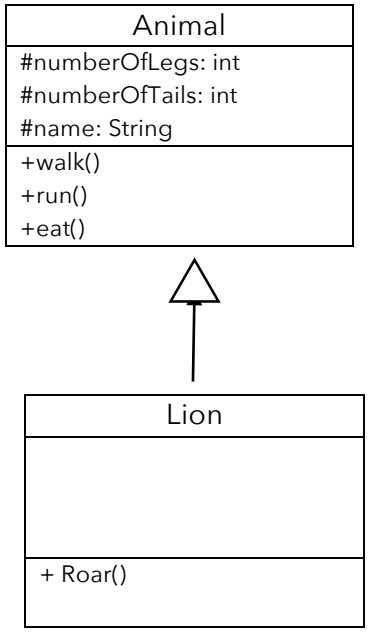
Generalization and inheritance can be represented UML class diagrams using a solid-lined arrow as shown below:



The solid-lined arrow indicates that two classes are connected by inheritance. The superclass is at the head of the arrow, while the subclass is at the tail. It is conventional to have the arrow pointing upwards. The class diagram is structured so that superclasses are always on top and subclasses are towards the bottom.

Inherited superclass' attributes and behaviours do not need to be rewritten in the subclass. Instead, the arrow symbolizes that the subclass will have the superclass' attributes and methods. Superclasses are the generalized classes, and the subclasses are the specialized classes.

It is possible to translate UML class diagrams into code. Let us build on the example above.



In this diagram, the Lion class is the subclass and the Animal class is the superclass. The subclass inherits all the attributes and behaviours from the superclass. As explained above, it is therefore unnecessary to put all of the superclass' attributes and behaviours in the subclass of the UML class diagram.

You may also notice the use of a # symbol. This symbolizes that the Animal's attributes are **protected**.

Protected attributes in Java can only be accessed by:

- the encapsulated class itself
- all subclasses
- all classes within the same **package**

In Java, a package is a way to organize classes into a **namespace** that represents those classes.

The UML class diagrams can be translated into code.

```
public abstract class Animal {
      protected int numberOfLegs;
      protected int numberOfTails;
      protected String name;

      public Animal( String petName, int legs, int
tails ) {
            this.name = petName;
            this.numberOfLegs = legs;
            this.numberOfTails = tails;
      }

      public void walk() { … }
      public void run() { … }
      public void eat() { … }
}
```

Since the Animal class is a generalization, it should not be created as an object on its own. The keyword **abstract** indicates that the class cannot be **instantiated**. In other words, an Animal object cannot be created.

The Animal class is a superclass, and any class that inherits from the Animal class will have its attributes and behaviours. Those subclasses that inherit from the superclass will share the same attributes and behaviours from the Animal class. Here is the code for creating a Lion subclass:

```
public class Lion extends Animal {
      public Lion( String name, int legs, int tails
) {
            super( name, legs, tails );
      }

      public void roar() { … }
}
```

None of the attributes and behaviours inherited from the Animal class need to be declared. This mirrors the UML class diagram, as

only specialized attributes and methods are declared in the superclass and subclass. Remember, the UML class diagram represents our design. As inherited attributes and behaviours do not need to be re-stated in the code, they do not need to be re-stated in the subclass in the diagram.

Inheritance is declared in Java using the keyword **extends**. Objects are instantiated from a class by using constructors. With inheritance, if you want an instance of a subclass, you must give the superclass a chance to prepare the attributes for the object appropriately. Classes can **have implicit constructors** or **explicit constructors**.

Below is an example of an implicit constructor:

```
public abstract class Animal {
      protected int numberOfLegs;

      public void walk() { … }
}
```

In this implementation, we have not written our own constructor. All attributes are assigned zero or null when using the **default** constructor.

Below is an example of an explicit constructor:

```
public abstract class Animal {
      protected int numberOfLegs;

      public Animal( int legs ) {
            this.numberOfLegs = legs;
      }
}
```

In this implementation, an explicit constructor will let us instantiate an animal with as many legs we want. Explicit constructors allow you to assign values to attributes during instantiation.

```
public abstract class Animal {
      protected int numberOfLegs;

      public Animal( int legs ) {
            this.numberOfLegs = legs;
      }
}
```

```
public class Lion extends Animal {
      public Lion( int legs ) {
            super( legs );
      }
}
```

A subclass' constructor must class its superclass' constructor if the superclass has an explicit constructor. Explicit constructors of the superclass must be referenced by the subclass; otherwise, the superclass attributes would not be appropriately initialized. To access the superclass' attributes, methods, and constructors, the subclass uses the keyword **super**.

Subclasses can **override** the methods of its superclass, meaning that a subclass can provide its own implementation for an inherited superclass' method.

```
public abstract class Animal {
      protected int numberOfLegs;

      public void walk() {
            System.out.println( "Animal is walking"
);
      }
}

public class Lion extends Animal {
      public void walk() {
            System.out.println( "I'd rather nap" );
      }
}
```

In the above example, the Lion class has overridden the Animal class's walk method. If asked to walk, the system would tell us that a Lion object would rather nap.

Types of Inheritance

Java is capable of supporting several different types of inheritance. The above examples in the lesson are **implementation inheritance**.

In Java, only single implementation inheritance is allowed. This means that while a superclass can have multiple subclasses, a subclass can only inherit from a single superclass.

For example, the Animal class might be a superclass to multiple subclasses: a Lion class, a Wolf class, or a Deer class. Each of these classes might have specialized behaviours or characteristics, so a Lion object knows how to roar but might not know how to howl, like a Wolf object.

Subclasses can also be a superclass to another class. Inheritance can trickle down through as many classes as desired.

Inheritance allows the generalization of related classes into a single superclass, and it still allows the subclasses to retain the same set of attributes and behaviours. This removes redundancy in the code and makes it easier to implement changes.

Interface Inheritance

Other languages like C++ support **multiple inheritance**. This is when a subclass can have two or more superclasses. Java addresses the restriction of single implementation inheritance by offering **interface inheritance**, another form of generalization. To understand that, first some programming language and design notions need to be explained.

A class denotes a type for its objects. The type signifies what these objects can do through public methods. In modelling a problem, it may be necessary to express subtyping relationships between two types. For example, instances of a Lion class are Lion-classed objects, which may perform specialized lion behaviours. A Lion type may also be a subtype of an Animal type. So, a Lion object is of a Lion type and an Animal type—a Lion object behaves like a lion and like an animal. Therefore, a lion "is" an animal.

In Java, implementation inheritance with the keyword **extends** is often used for subtyping. So, if a subclass extends a superclass, it will behave like the superclass and like its own class. The subclass inherits the "implementation details" of the superclass.

A Java interface also denotes a type, but an interface only declares method signatures, with no constructors, attributes, or method bodies. It specifies the expected behaviours in the method signatures, but it does not provide any implementation details.

A Java interface may also be used for subtyping. If a class implements an interface, then the class not only behaves like itself, it is also expected to behave according to the method signatures listed in the interface. The class needs to provide the method body details for what it means to implement the interface. An interface is like a contract to be fulfilled by implementing classes.

In implementation inheritance, there is consistency between the superclass type and the subclass type. A subclass object is usable anywhere in your program where you are dealing with the superclass type. Similarly, in interface inheritance, there is consistency between the interface type and the implementing class type.

In Java, the keyword **interface** is used to indicate that one is being defined. The letter "I" is sometimes placed before an actual name to indicate an interface.

```
public interface IAnimal {
      public void move();
      public void speak();
      public void eat();
}
```

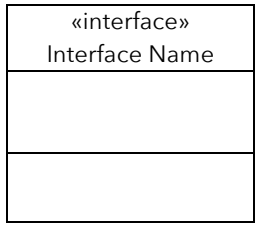
The interface shows that an animal has the behaviours of moving, speaking, and eating, but these behaviours are not implemented here, and there is no description as to how these behaviours are performed. The interface also does not encapsulate any attributes of the superclass—this is because attributes are not behaviours.

> ***Instructor's Note***:
> *The latest Java allows default implementations of the methods in interfaces. There are resources in the Course Readings section of the notes if you wanted to read more about this change!*

In order to use an interface, you must declare that you are going to fulfill the contract as described in the interface. The keyword in Java for this action is **implements**.

```
public class Lion implements IAnimal {
      /* Attributes of a lion can go here */

      public void move() { … }
      public void speak() { … }
      public void eat() { … }
}
```

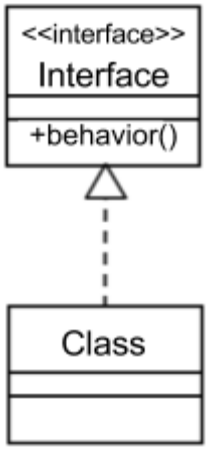
In this example, the Lion class has declared that it will implement or describe the behaviours that are in the interface—in this case, move, speak, and eat methods. You must have all the method signatures explicitly declared and implemented in the class.



A Java interface can describe the expected common behaviour of multiple classes, without directly implementing that behavior. An interface can be implemented by multiple classes, with each implementing class defining their own, appropriate version of the behaviour. Also, a class may implement multiple different interfaces.

Interfaces can be drawn in a similar way to classes in UML diagrams. Interfaces are explicitly noted using guillemets, or French quotes, to surround the word «interface».

The interaction between an interface and a class that is implementing the interface is indicated using a dotted arrow. The implementing class touches the tail end of the arrow and the interface touches the head of the arrow. The conventional way to draw interfaces on your UML class diagrams is to have the arrow pointing upward, so the interface is always on the top, and the classes that implement them are towards the bottom.



There are several advantages to interfaces. Understanding what these are will help you determine if you should use interfaces or inheritance when designing a system.

Like **abstract classes**, which are classes that cannot be instantiated, interfaces are a means in which you can achieve **polymorphism**. In object-oriented languages, polymorphism is when two classes have the same description of a behaviour, but the implementations of that behaviour may be different. An example of this might be how animals "speak." A lion may roar, but a wolf howls. Both animals can speak, but the behaviour implementation is different.

This can be demonstrated through code, as below:

```
public class Lion implements IAnimal {
      public void speak() {
            System.out.println( "Roar!" );
      }
}

public class Wolf implements IAnimal {
      public void speak() {
            System.out.println( "Howl!" );
      }
}
```

Interfaces can inherit from other interfaces, but interfaces should not be extended if you are simply trying to create a larger interface. Interface A should only inherit from interface B if the behaviours in interface A can fully be used as a substitution for interface B.

Examine the example below to better understand this concept.

```
public interface IVehicleMovement {
      public void moveOnX();
      public void moveOnY();
}
```

In this example, a vehicle can only travel either along the x-axis or the y-axis. But what if we want a different type of vehicle like a plane or a submarine to be able to move along a third axis as well? In order to avoid adding an extra behaviour to the interface, so as not to affect vehicles who can only move along two axes, we create a second interface that inherits from the first.

```
public interface IVehicleMovement3D extends
IVehicleMovement {
      public void moveOnZ();
}
```

Another advantage of interfaces relates back to multiple implementation inheritance. This is because inheriting from two or more superclasses can cause **data ambiguity**—if a subclass inherits from two or more superclasses that have attributes with the same name or behaviours with the same method signature, then it is not possible to distinguish between them. As Java cannot tell which one is referenced, so it does not allow for multiple inheritance to prevent data ambiguity.

Interfaces, however, do not have this issue. In Java, a class can implement as many interfaces as desired. This is because interfaces are only contracts and they do not enforce a specific way to complete these contracts, so overlapping method signatures are not a problem. A single implementation for multiple interfaces with overlapping contracts is acceptable. There is no ambiguity, as a class may only have one definition of a specific method, and it is the same implementation no matter which interface. Java avoids data ambiguity in this way.

Classes can implement one or more interfaces at a time, allowing for multiple types. Interfaces enable you to describe behaviours without the need to implement them, which allows reuse of those abstractions. Interfaces allow the creation of programs with reusable and flexible code. It is important to remember, however, that you should not generalize all behavioural contracts into interfaces. Interfaces fulfill a specific need: to provide for a way for related classes to work with consistency.

Now that you have an introduction to design principles and their manifestation in technical diagrams, this course will now turn to examining more nuanced understandings of those same design principles.

MODULE 3: DESIGN PRINCIPLES

Upon completion of this module, you will be able to:

(a) Understand the general guidelines for evaluating the structure of your software solution so that it's flexible, reusable, and maintainable. These guidelines are:
   a. evaluating design complexity with coupling and cohesion
   b. the separation of concerns
   c. information hiding
   d. conceptual integrity
   e. generalization principles

(b) Model behaviours of the objects in your software using the specialized UML state and UML sequence diagrams.

(c) Explain the importance of model checking.

This module covers general guidelines for evaluating the structure of software solutions. These guidelines help ensure that software is flexible, reusable, and maintainable. It will also cover modelling behaviours of the objects in your software using the UML state and UML **sequence diagrams**.

Evaluating Design Complexity

It is important to keep modules simple when you are programming. If your design complexity exceeds what developers can mentally handle, then bugs will occur more often. To help control this, there must be a way of evaluating your design complexity.

Design complexity applies to both classes and the methods within them. This lesson will use the term **module** to refer to program units containing classes and the methods within them.

A system is a combination of various modules. If the system has a bad design, then modules can only connect to other specific modules and nothing else. A good design allows any modules to connect together without much trouble. In other words, in a good design, modules are compatible with one another and can therefore be easily connected and re-used.

The metrics often used to evaluate design complexity are **coupling** and **cohesion**.

Coupling

Coupling focuses on complexity between a module and other modules. Coupling can be balanced between two extremes: tight coupling and loose coupling. If a module is too reliant on other modules, then it is "**tightly coupled**" to others. This is a bad design. However, if a module finds it easy to connect to other modules through well-defined interfaces, it is "**loosely coupled**" to others. This is good design.

In order to evaluate the coupling of a module, the metrics to consider are: **degree**, **ease**, and **flexibility**.

Degree is the number of connections between the module and others. The degree should be small for coupling. For example, a module should connect to others through only a few parameters or narrow interfaces. This would be a small degree, and coupling would be loose.

Ease is how obvious are the connections between the module and others. Connections should be easy to make without needing to understand the implementations of other modules, for coupling purposes.

Flexibility indicates how interchangeable the other modules are for this module. Other modules should be easily replaceable for something better in the future, for coupling purposes.

Signs that a system is tightly coupled and has a bad design are:

- a module connects to other modules through a great number of parameters or interfaces
- corresponding modules to a module are difficult to find
- a module can only be connected to specific other modules and cannot be interchanged

Cohesion

Cohesion focuses on complexity within a module, and represents the clarity of the responsibilities of a module. Like complexity, cohesion can work between two extremes: **high cohesion** and **low cohesion**.

A module that performs one task and nothing else, or that has a clear purpose, has high cohesion. A good design has high cohesion. On the other hand, if a module encapsulates more than one purpose, if an encapsulation has to be broken to understand a method, or if the module has an unclear purpose, it has low cohesion. A bad design has low cohesion. If a module has more than one responsibility, it is a good idea to split the module.

It is important to balance between low coupling and high cohesion in system design. Both are necessary for a good design. However, in complex systems, complexity can be distributed between the modules or within the modules. For example, as modules are simplified to achieve high cohesion, they may need to depend more on other modules, thus increasing coupling. On the other hand, as connections between modules are simplified to achieve low coupling, the modules may need to take on more responsibilities, thus lowering cohesion.

Separation of Concerns

One of the design principles examined in the previous module was that of decomposition. Decomposition divides a whole into different parts. To understand why decomposition is necessary in design, the principle of **separation of concerns** must be examined.

A **concern** is a very general notion: it is anything that matters in providing a solution to a problem. Separation of concerns is about keeping the different concerns in your design separate. When software is designed, different concerns should be addressed in different portions of the software.

Consider a software system that solves a problem. That problem could either be simple, with a small number of subproblems, or complex, with a large number of subproblems. Concepts can be abstracted from the problem space. When these abstractions are implemented in the software, it can lead to more concerns. For example, some of these concerns might involve what information the implementation represents, what it manipulates, and what gets presented at the end. In order not to become lost in the resulting concerns and subproblems, the design must be organized so all concerns are carefully considered and addressed. To do this, different subproblems and concerns are separated into different sections during design and construction of the software system. This applies the principle of separation of concerns.

Separation of concerns provides many advantages. They allow you to develop and update sections of the software independently. Using separation of concerns also means that you do not need to know how all sections of code work in order to update a section. Finally, separation of concerns allows changes to be made to one component without requiring a change in another.

Separation of concerns is a key idea that underlies object-oriented modelling and programming. When addressing concerns separately, more cohesive classes are created and the design principles of abstraction, encapsulation, decomposition, and generalization are enforced:

- Abstraction occurs as each concept in the problem space is separated with its own relevant attributes and behaviours.

- Encapsulation occurs as the attributes and behaviours are gathered together into their own section of code called a class. Access to the class from the rest of the system and its implementation are separated, so details of implementation can change while the view through an interface can stay the same.

- Decomposition occurs as a whole class can be separated into multiple classes.

- Generalization occurs as commonalities are recognized, and subsequently separated and generalized into a superclass.

Separation of concerns is an ongoing process throughout the system design process. Because of the relationship of the separation of concerns with design principles, using this concept in software design creates a system that is easier to maintain because each class is organized so that the class only contains the code that it needs to do its job. Modularity is increased in turn, which allows developers to reuse and build up individual classes without affecting others.

It is important to note that the boundaries of each class will not always be obvious in practice. Deciding how to abstract, encapsulate, decompose, and generalize to address the many concerns for a given problem is at the core of designing modular software.

Separation of Concerns Example

This example will illustrate separation of concerns. Consider a smartphone. Smartphones are capable of many behaviours: taking photos, scheduling meetings, sending and receiving email, browsing the Internet, sending texts, and making phone calls. This example will only focus on two functions, for the sake of simplicity: the use of a camera and traditional phone functions.

```
public class SmartPhone {
      private byte camera;
      private byte phone;

      public SmartPhone() { … }

      public void takePhoto() { … }
      public void savePhoto() { … }
      public void cameraFlash() { … }
      public void makePhoneCall() { … }
      public void encryptOutgoingSound() { … }
      public void decipherIncomingSound() { … }
}
```

This code has a SmartPhone class with attributes called camera and phone, and associated behaviours. This system has low cohesion, as there are behaviours that are not related to each other. The camera behaviours do not need to be encapsulated with the behaviours of the phone in order for the camera to do its job. The components also do not offer modularity. For example, it is not possible to access the camera or the phone separately if another system is built that requires only one or the other. The camera also would not be able to be replaced with a different camera, or even a different object, without removing the code for the camera completely from this class.

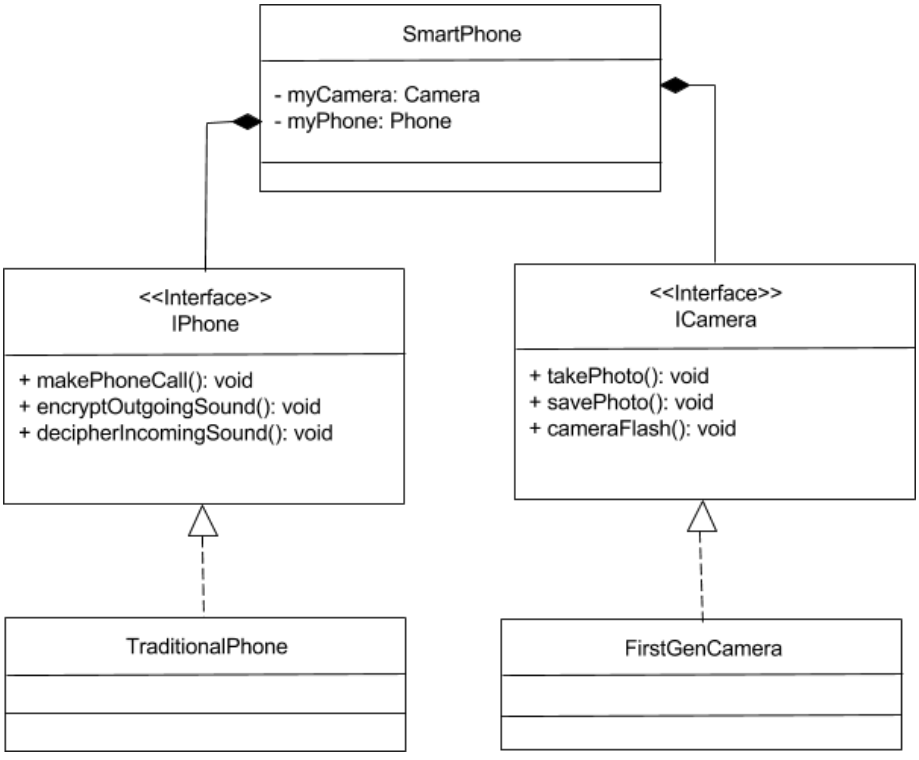
The SmartPhone class needs to be more cohesive, and each component of the smartphone should have distinctive functionality. Using the separation of concerns, we can identify that the SmartPhone class has two concerns:

1. To act as a traditional telephone.
2. To take pictures using the built-in camera.

With the concerns identified, it is possible to separate them into their own more cohesive classes and encapsulate all the details about each concern into functionally distinct and independent classes.

The SmartPhone class will reference instances of the new classes so that the smartphone can act as a coordinator of the camera and the phone. This will let our smartphone provide access to all the behaviours of the camera and the phone, without having to know how each component behaves.

Using a UML Class diagram, this is how the new design for the SmartPhone system might look:



The attributes and behaviours for the phone and camera have been separated into two distinct interfaces. These are each implemented with a corresponding class.

The code would translate as below:

```
public interface ICamera {
      public void takePhoto();
      public void savePhoto();
      public void cameraFlash();
}

public interface IPhone {
      public void makePhoneCall();
      public void encryptOutgoingSound();
      public void deciphereIncomingSound();
}

public class FirstGenCamera implements ICamera {
      /* Abstracted camera attributes */

public class TraditionalPhone implements IPhone {
      /* Abstracted phone attributes */
}
```

The code for the SmartPhone class will also need to be redesigned so that it refers to the two separate classes:

```
public class SmartPhone {
      private ICamera myCamera;
      private IPhone myPhone;

      public SmartPhone( ICamera aCamera, IPhone
aPhone ) {
            this.myCamera = aCamera;
            this.myPhone = aPhone;
      }

      public void useCamera() {
            return this.myCamera.takePhoto();
      }

      public void usePhone() {
            return this.myPhone.makePhoneCall();
      }
}
```

With this redesign, the SmartPhone class provides the functions of both the camera and the phone. However, the camera and phone classes are separated out, so their functionalities are hidden from each other, but they are still aggregated under the SmartPhone class. There is also a SmartPhone constructor with a camera and a phone as parameters. It is possible to create a new instance of the SmartPhone class by passing in instances of classes that implemented the ICamera and IPhone interfaces. Who creates the appropriate phone and camera objects is left as a separate responsibility, as the SmartPhone class does not actually need to know this. Finally, the SmartPhone class has methods that forward the responsibilities of using the camera and phone to these objects.

This creates a modular design: if the design later calls for the camera or phone classes to be swapped for something else, then the SmartPhone class's code does not need to be touched. The code is simply changed to instantiate the SmartPhone and its parts.

The SmartPhone class is now more cohesive but has increased coupling as the SmartPhone class needs to know about the Camera and Phone interfaces and is indirectly dependent on other classes.

In this example, separation of concerns was used by:

- Separating out the notions of camera and phone through generalization and defining the two interfaces.
- Separating out the functionality for a first-generation camera and traditional phone by applying abstraction and encapsulation, and defining two implementing classes.
- Applying decomposition to the smartphone so the constituent parts are separated from the whole.

Information Hiding

A well-designed system is well organized, achieved through the help of a number of design principles. This lesson will further investigate the concept of information access. Not every component of a software system needs to know about everything else. Modules should only have access to the information it needs to do its job. Limiting information to modules so that only the minimum amount of information is needed to use them correctly and to "hide" everything else is done through **information hiding**.

Information hiding is commonly associated with sensitive data –the more sensitive the data, the more likely it should have limited access. In software design, information hiding is also specifically used to hide changeable details, such as algorithms or data representations. Assumptions, on the other hand, are not hidden and are typically expressed in APIs and interfaces.

Information hiding allows developers to work on modules separately, without needing other developers to know the implementation details of the module they are working on. The module is instead used through its interface.

A **good rule of thumb** is therefore things that might change, like implementation details, should be hidden, and things that do not change, like assumptions, are revealed through interfaces.

Information hiding is closely associated with encapsulation. Encapsulation bundles attributes and behaviours into their appropriate class, but it also deals with providing access to modules through interfaces and restricting access to certain behaviours or functions. Since through encapsulation, the implementation of behaviours is hidden behind an interface, which is the only way to access specific methods, other classes rely on information in these method signatures, and not the underlying implementations. Information hiding through encapsulation allows the implementation to change without changing the expected outcome. The expectations for a behaviour can be fulfilled without exposing how it is accomplished.

In addition to hiding implementation or behaviours, it is possible to hide attributes. This prevents critical information of a class from being changed directly. For example, if an attribute is critical to all the behaviours of a class, then we do not want any external classes changing it directly.

Information hiding can be accomplished through the use of **access modifiers**. Access modifiers change which classes are able to access attributes and behaviours. They also determine which attributes and behaviours a superclass will share with its subclasses. The four levels of access in Java are:

- Public
- Protected
- Default
- Private

Let us examine each of these in turn.

Public

Attributes with a **public** access modifier are accessible by any class in your system. This means that other classes can retrieve and modify the attribute or change. Methods can also be given a public level of access, so any class in the system can access the method. However, this access does not allow other classes to change the implementation of the behaviour for the method. A publicly

accessible method simply allows other classes to call the method or invoke the behaviour and receive any output from it. However, implementation remains hidden through encapsulation.

Protected

Attributes and methods that are **protected** are not accessible to every class in the system. Instead, they are only available to the encapsulated class itself, all subclasses, and classes within the same package. Packages are the means by which Java organizes related classes into a single namespace.

Default

A **default** access modifier only allows access to attributes and methods to subclasses or classes that are part of the same package or encapsulation. This access modifier is also known as the no modifier access because it does not need to be explicitly declared in the code.

Private

Attributes and methods that are **private** are not accessible by any other class other than by the encapsulating class itself. This means these attributes cannot be accessed directly and these methods cannot be invoked by any other classes.

The following example in code shows how access modifiers can be indicated:

```
public class Person {
      String name;
}
```

In this example, the access modifier `public` has been used for the Person class, and the name member variable has default access as no modifier is indicated. Access modifiers `protected` or `private` could also be used for the name to apply different access.

Conceptual Integrity

Conceptual integrity is a concept related to creating consistent software. Conceptual integrity entails making decisions about the design and implementation of a software system, so even if multiple people work on it, it would seem cohesive and consistent as if only one mind was guiding the work. This is generally achieved through agreement to use certain design principles and conventions for creating the system.

It is important that the concept of conceptual integrity not be mistaken for ignoring the opinion of members of the development team about the software. These thoughts are still important and should be discussed openly. However, any ideas should abide by the agreed upon principles and conventions.

There are multiple ways to achieve conceptual integrity. These include:

- communication
- code reviews
- using certain design principles and programming constructs
- having a well-defined design or architecture underlying the software
- unifying concepts
- having a small core group that accepts each commit to the code base.

Effective communication maintains conceptual integrity and allows team members to discuss and agree to use certain libraries or methods when addressing certain issues. This in turn helps create more consistent code. Some good practices to foster communication include agile development practices like daily stand-up meetings and sprint retrospectives.

> ***Instructor's Note:***
>
> *To learn more about stand-up meetings and sprint retrospectives, see the course on reviews and metrics for software improvements in the Coursera* ***Software Product Management*** *Specialization!*

Code reviews are systematic examinations of written code. These are similar to peer review in academic writing. Developers go through code line by line and uncover issues in each other's code. This helps identify mistakes in the software, but it also helps create consistency among different developers' code.

Using certain design principles and programming constructs helps maintain conceptual integrity. Notably, Java interfaces are a construct that can accomplish this. An interface defines a type with a set of expected behaviors. Implementing classes of that interface will have these behaviors in common. This creates consistency in the software, and increases conceptual integrity. Later in this specialization, we will also cover **design patterns**, which provide conventional structures for classes to solve design issues. These design patterns also provide conceptual integrity.

Having well-defined design or architecture underlying software helps create conceptual integrity. While software design is typically associated with guiding the internal design of software running as a single process, software architecture describes how software running as multiple processes work together and how they relate to each other. This helps create consistency.

Unifying concepts in your software also increases conceptual integrity. This involves taking different concepts and finding a commonality, so that each concept can be seen and treated in similar ways. For example, in the Unix operating systems, every resource can be seen and manipulated as if it were a file. The same set of operations can then be used on different types of resource, simplifying the system so that any resource can be treated the same way, which avoids special cases. This helps create consistency.

Another means of increasing conceptual integrity is having a small core group that accepts each commit to the code base. This is similar to exercising code reviews, but it restricts the review to only core members of the software team. These members are responsible for ensuring that any software changes follow the overall architecture and design of the software. By having only an individual or small group in charge of this helps solve design issues and creates consistency.

Conceptual integrity is often considered the most important consideration in system design.

DID YOU KNOW?

The well-known computer architect, Fred Brooks discusses conceptual integrity in his book *The Mythical Man-Month*. In this book, he states,

> It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas."

Practicing conceptual integrity helps guide the software development team by making the design and logic of the software consistent and easy to follow for any team member. This helps team members know how and where to change software to meet any new requirements, which makes the software easier to maintain. Conceptual integrity can serve as structure and framework for any software project by preventing informal and unguided code, which could lead to confused and unorganized work.

Generalization Principles

The previous module in this course reviewed the four design principles of abstraction, encapsulation, decomposition, and generalization. These principles help guide the choices that must be made when designing an object-oriented system.

However, some design decisions are easier to make than others. Generalization and inheritance are some of the more difficult topics to master in object-oriented programming and modelling.

Inheritance is a powerful design tool that can help create clean, reusable, and maintainable software systems. However, its misuse can lead to poor code. If design principles are used improperly, they may create more problems than they solve.

In order to identify if inheritance is being misused, it is good practice to keep a couple of generalization principles in mind.

One principle can be formulated as a question to ask yourself about whether a subclass should exist: "Am I using inheritance to simply share attributes or behaviour without further adding anything special in my subclasses?" If the answer to this question is "yes," then inheritance is being misused, as there is no point for the subclasses to exist. The superclass should already be enough.

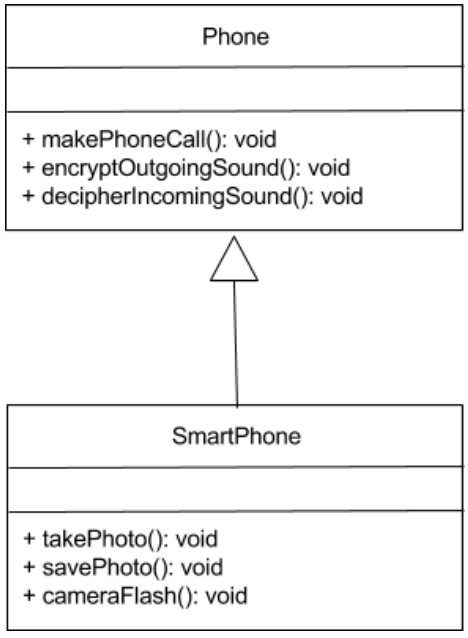
For example, an employee is a general type for managers, salespeople, and cashiers, but each of those subtypes of employee perform specific functions. Inheritance makes sense in this case. However, if you are creating different kinds of pizza, there is no true specialization between different kinds of pizza, so subclasses are unnecessary.

Another technique is determining if the **Liskov substitution principle** is broken. The Liskov substitution principle states that a subclass can replace a superclass, if and only if, the subclass does not change the functionality of the superclass. This means that if a subclass replaces a superclass, but replaces all the superclass behaviours with something totally different, then inheritance is being misused.

For example, if a Whale class which exhibits swimming behaviour is substituted for an Animal class, then functions such as running and walking will be overridden. The Whale no longer behaves in the way we would expect its superclass to behave, violating the Liskov substitution principle.

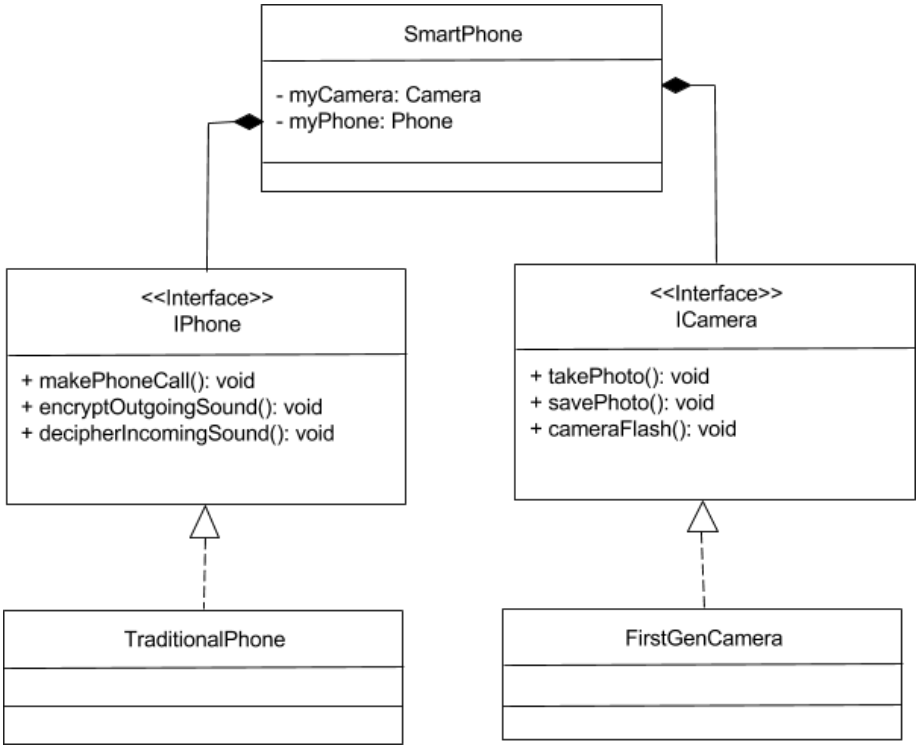
In Java, there is a library with a Stack class, which is an example of bad inheritance. A stack is understood as first-in-last-out data structure, with a small number of well-defined behaviours like peek, pop, and push. But, the Java Stack class inherits from a Vector superclass. This means that the Stack class is able to return an element at a specified index, retrieve the index of an element, and even insert an element into a specific index. These are not behaviours normally expected from a stack.

In cases where inheritance is not appropriate, decomposition may be the solution. For example, a smartphone is better suited for decomposition than inheritance. Recall our example from the section on separation of concerns, where a smartphone might have the two functions of a traditional phone and as a camera.



In this example, it does not make sense to use inheritance from a traditional phone to a smartphone, and then to add camera methods to smartphone subclass.

Instead, decomposition helps extract the camera's responsibilities into their own class. This allows the SmartPhone class to provide the responsibilities of the camera and the phone through separate classes. The SmartPhone class does not need to know how these classes work.



Although inheritance is a powerful principle, it is important to know when to properly use a technique, or risk introducing more problems to a software system.

Specialized UML class diagrams

We have already reviewed the use of UML class diagrams to express technical design. However, many different kinds of UML class diagrams exist. Below, the following section explores two specialized versions of class diagrams, and how they can be used to enhance your technical design.

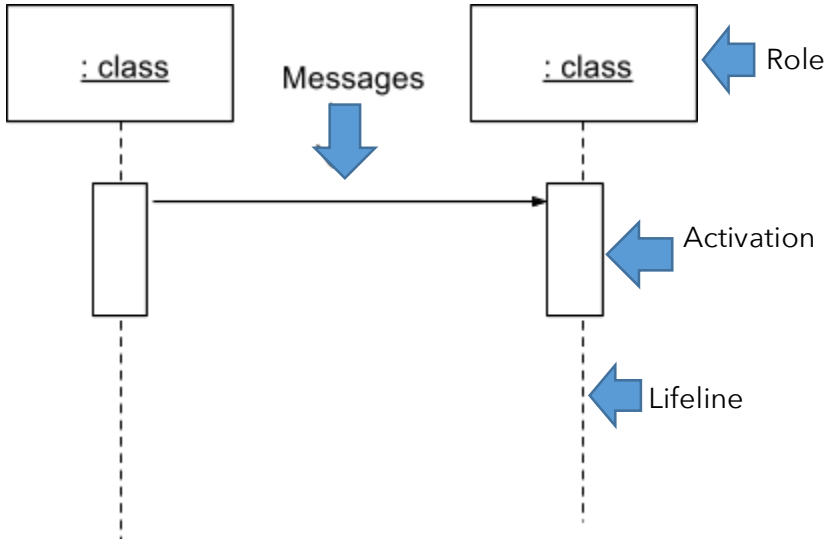
UML Sequence Diagrams

UML sequence diagrams are another important technique in software design. They are a type of UML diagram, commonly used as a planning tool before the development team starts programming. Sequence diagrams are used to show a design team how objects in a program interact with each other to complete tasks. In simple terms, a sequence diagram is like a map of conversations between different people, with the messages sent from person to person outlined.

Sequence diagrams can help you visualize the classes you will create in your software and determine the functions that will need to be written. It may also illustrate problems in your system that were previously unknown.

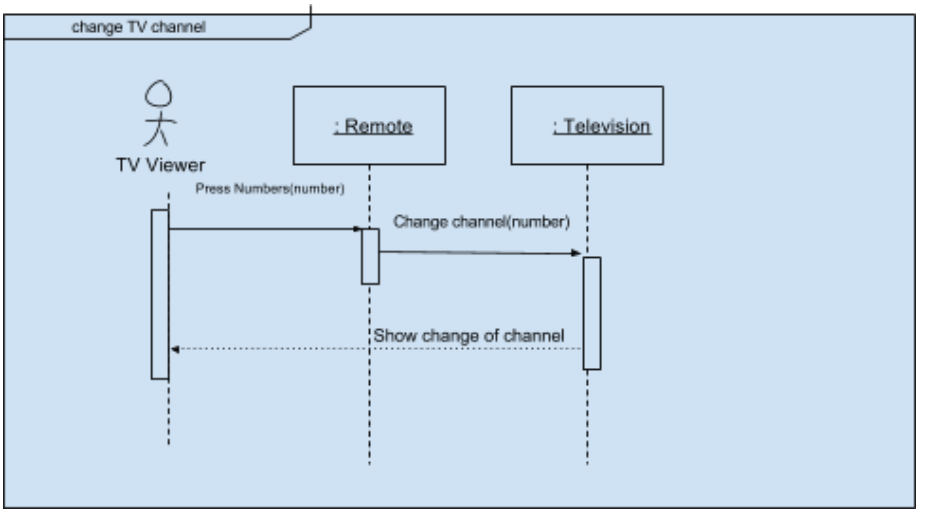
Let us examine each component of a sequence diagram.

- Boxes are used to represent a role played by an object. The **role** is typically named after the class for the object.
- "Lifelines," which are vertical dotted lines, are used in the diagram to indicate an object as time goes by.
- Solid line arrows are used to show **messages** that are sent from one object to another, or a sender to a receiver. Receivers are at the pointed end of an arrow. A short description of the message is usually included above the arrow.
- Dotted line arrows are used to show a return of data and control back to initiating objects. A short description of the return of data or control is usually included above the arrow.
- Small rectangles along an object's lifeline denote a method **activation**. You activate an object whenever an object sends, receives, or is waiting for a message.
- People, or **actors**, may also be included in sequence diagrams if they use or interact with objects. Actors are typically represented with stick figures.



Sequence diagrams are typically framed within a large box. Titles for the diagram are indicated in top, left corners. It is good practice to provide a meaningful title, as the diagram will be referenced for development. Another good practice is to draw objects from left to right in the sequence as they interact with each other.

Below is an example of a sequence diagram for changing the channel of your television using a remote control, with all of the elements described above.



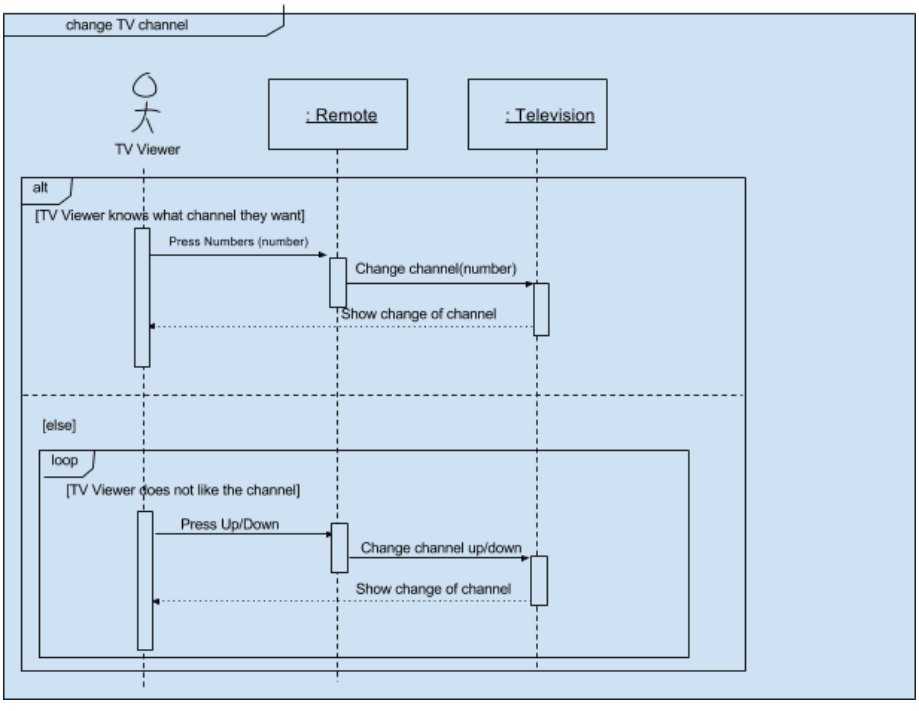
A sequence diagram can contain other sequence diagrams within them. For example, if you are creating a sequence diagram for an ATM machine, there might be a different sequence for withdrawals and deposits; and during a single process someone might want to both withdraw and deposit money. In your sequence diagram, you would have one big sequence of **activities**, with two smaller sequences inside them.

As you design software, your sequence diagrams may get much more complicated. Loops and alternative processes can also be demonstrated in a sequence diagram. An alternative process is a sequence of actions that will occur if a condition is true. An alternative sequence can be placed in a box and labelled "alt" for alternative in the top right corner.

Let us build on the previous example to illustrate this. Imagine a scenario where the TV viewer is unsure what channel to go to and would like to surf channels until they pick one they like. This scenario can be illustrated under the previous sequence with the condition "[else]." This indicates that this scenario occurs only if all the other alternatives are false.

This scenario also contains a loop. This can be illustrated through adding a box labelled "loop." Under the label, the conditional statement for the loop should be written. If that statement is true, then the system will go through the loop. In this example, the loop sequence should continually occur if the TV viewer is unhappy with the channel they are watching.

Inside the loop, the TV viewer presses the up or down arrow on the remote to change channels. This sends a message to the remote. The remote then sends a message to the TV with this action. The TV changes the channel and displays that to the viewer. The final sequence will look as below:



Sequence diagrams are a useful technique to help create clean, well- designed programs.

UML State Diagrams

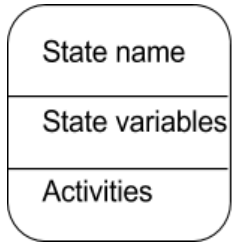
UML state diagrams are another kind of UML diagram. They are a technique used to describe how systems behave and respond. They follow the states of a system or a single object and show changes between the states as a series of events occur in the system.

A state diagram illustrates object behaviour by depicting the changing states of an object. These change in response to different events. A **state** is the way an object exists at a particular point in time. The state of an object is determined by the values of its attributes.

A good metaphor for states is that of a car. A car with an automatic transmission might have different states: park, reverse, neutral, and drive. If a car is in reverse, it can only behave by moving backwards. If you want to move forward, you need to change the state of the car to drive. This is similar to states of objects in a software system. When an object is in a certain state, it behaves in a specific way or has attributes set to specific values. Let us examine the different elements of state diagrams:

- A filled circle indicates the starting state of the object. Every state diagram begins with a filled circle.

- Rounded rectangles indicate other states. These rectangles have three sections: a state name, state variables, and **activities**.

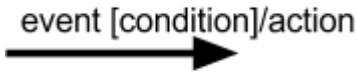


    - State names should be short, meaningful titles for the state of the object. Each state should have at least a state name.

- State variables are data relevant to the state of the object.
- **Activities** are actions that are performed when in a certain state. There are three types of activities

present for each state: Entry, Exit, and Do activities. **Entry activities** are actions that occur when the state is just entered from another state. **Exit activities** are actions that occur when the state is exited and moves on to another state. **Do activities** are actions that occur while the object is in a certain state.

- Arrows indicate **transitions** from one state to another. Transitions are typically triggered by an event. This event is generally described above the arrow.



  Each transition arrow will always have an event, and it may even have a guard condition and an action. The transition and action happens from a given state if the event occurs and the condition is true.
- A circle with a filled circle inside indicates termination. Termination represents an object being destroyed or the process being completed. Not all diagrams have a termination—some may run continuously.

State diagrams can be useful to help determine the events that might occur during an object's lifetime, such as different user inputs, and how that object should behave when these events occur, like checking conditions and performing actions. Sometimes it may be easier to see changes in state in a diagram, rather than reading through source code.

State diagrams can also help identify issues in a software system, such as discovering a condition that was unplanned for. They can also help create tests—knowing the different states of a system can help ensure that tests are complete and correct.

Model Checking

In addition to understanding techniques for designing a software system, it is important to know techniques for verifying the system. Some of these techniques include unit testing, beta testing, and simulations. Another one such technique is **model checking**, which is a systematic check of your system's **state model** in all its possible states. Model checking helps find errors that other tests cannot.

In model checking, you check all the various states of the software to try and identify any errors, by simulating different events that would change the states and variables of the software. This will help expose any flaws by notifying you of any violation of the rules that occur in the behaviour of the state model. Typically, model checks are performed by model checking software. There are different types of software available for such tests, some of which are free and available for developers using different languages. Model checking is typically performed during testing of the software.

Imagine software that has a rule not to produce a **deadlock**. Deadlock is a situation where the system cannot continue because two tasks are waiting for the same resource. The model checker would simulate the different states that could occur in your system, and if a deadlock was possible, it would provide details of this violation.

Let us go through the process for model checking software.

Model checkers begin by generating a state model from your code. A **state model** is an abstract state machine that can be in one of various states. The model checker then checks that the state model conforms to be certain behavioural properties. For example, the model checker can examine the state model for flaws like race conditions, exploring all the possible states of your model.

There are three different phases in model checking.

The first is the **modelling phase**. During this phase, the model description is entered in the same programming languages as the system. Any desired properties are also described. This phase also performs **sanity checks**. Sanity checks are quick checks that should be easy to do, as they come from clear and simple logic. It is beneficial to test for these simple errors before using model checkers, so the focus can be on specifying the more complex properties to check. Sanity checks might include something as simple as turning the system on and off.

The second phase is the **running phase**. The running phase is when the model checker is run to see how the model conforms to the desired properties described in the modelling phase.

The third and final phase is the **analysis phase**. This phase is when all desired properties are checked to be satisfied, and if there are any violations. Violations are called **counterexamples**. The model checker should provide descriptions of violations in the system, so you can analyze how they occurred.

Information provided by the model checker allows you to revise your software and fix any problems. Once problems are fixed, it is good practice to run the model checker again. Repeat this process until you are sure the software is correct with respect to the desired properties.

Model checking helps ensure not only that software is well designed, but also that software meets desired properties and behaviour, and it works as intended.

COURSE RESOURCES

Course References

- Baier, C., & Katoen, J. P. (2007). *Principles of model checking*. Cambridge, MA; Massachusetts Institute of Technology. Retrieved from http://is.ifmo.ru/books/_principles_of_model_checking.pdf

- Bell, D. (2004, February 16). UML basics: The sequence diagram. IBM: developerWorks. Retrieved from http://www.ibm.com/developerworks/rational/library/3101.html

- Clarke, E. M. (n.d.). The birth of model checking. Retrieved from https://www7.in.tum.de/um/25/pdf/Clarke.pdf

- Cvijanovic, D. (n.d.). Model checking. Retrieved from http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/3C05-02-03/aswe17-essay.pdf

- Katoen, J. P. (2013, October 14). *Introduction to model checking: Lecture #1: motivation, background, and course organization* [Powerpoint presentation]. Retrieved from https://moves.rwth-aachen.de/wp-content/uploads/SS16/mc/lec1.pdf

- Nadig, D. (2011, October 29). The importance of conceptual integrity [Blog post]. Retrieved from http://architecture.typepad.com/architecture_blog/2011/10/the-importance-of-conceptual-integrity.html

- Palshikar, G. K. (2004, February 12). An introduction to model checking. Retrieved from http://www.embedded.com/design/prototyping-and-development/4024929/An-introduction-to-model-checking

- Sironi, G. (2012, July 25). Lean tools: Conceptual integrity. Retrieved from https://dzone.com/articles/lean-tools-conceptual-0

- Trace Modeler. (2012). A quick introduction to UML sequence diagrams. Retrieved from http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

Glossary

Word	Definition
"Don't Repeat Yourself" rule (D.R.Y. rule)	A rule related to the design principle of generalization. D.R.Y. suggests that we should write programs that are capable of performing the same tasks but with less code. Code should be reused because different classes or methods can share the same blocks of code. D.R.Y. helps make systems easier to maintain.
Abstract	A keyword used in Java to indicated that a class cannot be instantiated.
Abstract classes	Classes that cannot be instantiated.
Abstract data type	A data type that is defined by the programmer and not built into the language. It is a grouping of related information that is denoted with a type that allows data to be organized in a meaningful way.
Abstraction	A design principle that suggests that a concept in the problem domain should be simplified down to its essentials within some context.
Abstraction barrier	A barrier achieved through encapsulation, which allows the internal workings of a class to be hidden from the outside world when it is not relevant. It reduces complexity for users of a class.
Access modifiers	Change that classes are able to access attributes and behaviours.
Activities	Actions that are performed when in a certain state.
Aggregation	A type of relationship of the design principle of aggregation. Aggregation indicates a "has-a" relationship, where a whole has parts that belong to it. Parts might be shared among wholes in this relationship.
Algol 68	A programming language from the 1970s that supports the notion of an abstract data type.
Analysis Phase	A phase in model checking, where you check if all the desired properties are satisfied, and if any are violated.

Association	A type of relationship of the design principle of decomposition. Association indicates that there is a loose relationship between two objects. This means that objects may interact with each over for a time.
Attributes	The characteristics of an object. Basic attributes are ones that do not disappear over time, although their values may change.
Behaviours	The responsibilities that an abstraction does for its purpose.
Black box	A kind of thinking associated with encapsulation. In black box thinking, a class acts like a "black box" that you cannot see inside for details about how attributes are represented or how methods compute their result. What happens in the box does not matter to achieving expected behaviours. When a method is called, inputs and outputs may be obtained, even if the inner workings of the box are hidden.
Boolean value	A value that is either true or false.
Boundary objects	A type of object, usually introduced in the solution space of a software problem that connect to services outside of the system.
C	A programming language from the mid-1970s, that provided a means to organize problems and allowed developers to create multiple, but unique, copies of their abstract data types more easily.
Class	An abstract concept that unifies a component together. For example, a component might be a dog. The dog would be part of a dog class.
Class name	The name provided to a class of objects in design.
Class, responsibility, collaborator (CRC) cards	A visual technique for recording, organizing, and refining components in software design at a high level during conceptual design. CRC uses small cards indicating class, responsibilities, and collaborations of components in software.
COBOL	A popular programming language from the 1960s, that follows an imperative paradigm.
Code reviews	Systematic examinations of written code, similar to peer review in writing.

Cohesion	Design complexity that occurs within a module. It represents the clarity of the responsibilities of a module.
Collaborators	Components that work or interact with another identified component in software design, in order to fulfill its responsibilities.
Components	Components correspond to objects or concepts of a software that must work together for the software to work.
Composition	A type of relationship of the design principle of decomposition. Composition is an exclusive containment of parts, otherwise known as a strong has-a relationship. In a composition relationship, a whole cannot exist without its parts, and if a whole is destroyed, then the parts are destroyed too.
Concept	Concepts may include instances of people, places, or things, often grouped as a distinct object in software design.
Conceptual design	The stage of software design, where broad-level concepts are planned out, often with the help of conceptual mock-ups. The conceptual design helps establish what the major components, connections, and associated responsibilities are for the software solution.
Conceptual integrity	The concept of creating consistent software. Conceptual integrity requires making decisions about how a system will be designed and implemented so it seems as if only a single mind guided all the work.
Conceptual mock-ups	Provide a visual expression of initial thoughts for how requirements will be satisfied in a software product.
Concern	A very general notion. Could include anything that matters in providing a solution to a problem.
Connections	Connections indicate how different components of a software design relate to each other.
Context	The background or framework surrounding a problem. For example, the problem of creating an app could be within a gaming context.

Control objects	A type of object, usually introduced in the solution space of a software problem, that receives events and coordinates actions.
Counterexamples	A violation of a desired property, as discovered during the analysis phase of model checking.
Coupling	Design complexity that occurs between a module and other modules.
Data	The attribute values of an object.
Data ambiguity	When inheritance occurs from two or more superclasses, and the inherited attributes inherited have the same name or the inherited behaviours have the same method signatures, and it is impossible to tell between them.
Data integrity	The assurance that the attribute values of an object have been changed in an appropriate manner by an approved party.
Deadlock	A situation when the system cannot continue because two tasks are waiting for the same resource.
Decomposition	A major design principle of object-oriented modelling and programming. It takes a whole thing and divides it up into different parts. It could also take separate parts with different functionalities, and combine them together to form a whole.
Default	An access modifier and level in Java. Also known as the no modifier access, as it does not need to be explicitly declared. Attributes and methods with no modifier will only allow access to subclasses and to the encapsulating class.
Degree	The number of connections between a module and other modules in a system.
Design	The activity of planning out a software solution, including evaluating different alternatives.
Design patterns	A reusable solution to a problem identified in software design.
Do activities	Actions that occur once, or multiple times while the object is in a certain state.

Ease	How obvious the connections are between a module and other modules in a system.
Encapsulation	A fundamental design principle in object-oriented modelling and programming. Encapsulation involves three ideas. It bundles attribute values and behaviours that manipulate those values into a self-contained object. It also exposes certain data and functions of the object to other objects, or alternately restricts access to certain data and function to only within that object.
Entity object	A type of object, often identified in the problem space, that represent items used in the application.
Entry activities	Actions that occur when the state is just entered from another state.
Exit activities	Actions that occur when the state is exited and moves on to another state.
Explicit constructor	A constructor in Java that indicated that values can be assigned to attributes during instantiation.
Extends	A keyword in Java that indicated inheritance.
Flexible	Software flexibility refers to the ability for the solution to adapt to possible or future changes in its requirements
Flexibility	Indicates how interchangeable other modules are for a specific module. The more replaceable, the better the design.
Fortran	A popular programming language from the 1960s, that follows an imperative paradigm.
Functions	The behaviours that manipulate attribute values within an object.
General idea	A class or broad term used to describe a large grouping of more distinct classes.
Generalization	A design principle that helps reduce the amount of redundancy when solving problems, by taking repeated, common, or shared characteristics between two or more classes, and factoring them out into another class, so that code can be reused and characteristics can be inherited by subclasses.

Global data	Data that is located all in one place in the computer's memory for a program.
Header file	A separate file in C, that declares what can be accessed in other files. In C, each file contains all associated data and functions that manipulate that data.
High cohesion	Occurs if a module performs one task and nothing else, or if it has a clear purpose. This is considered a characteristic of good design.
Imperative paradigm	A paradigm that broke up large programs into smaller programs, called subroutines. This paradigm made use of global data.
Implementation	The process of putting a method or event into effect.
Implements	A keyword in Java that indicates a declaration that the contract is going to be fulfilled as described in the interface.
Implementation inheritance	A kind of inheritance that suggests that a superclass can have multiple subclasses, but that a subclass can only inherit from one superclass. This kind of single implementation is the only one allowed in Java.
Implicit constructor	A constructor in Java that occurs when a constructor has not been written. All attributes are assigned zero or null when using the default constructor.
Information hiding	Allows modules in a system to give others the minimum amount of information needed to use them correctly and "hide" everything else. This allows modules to be worked on separately.
Inheritance	According to the principle of generalization, repeated, common, or shared characteristics between two or more classes are taken and factored into another class. Subclasses can then inherit the attributes and behaviours of this generalized or parent class.
Instantiated	The creation of an instance of an object.

Interface	The point where two objects meet and interact. Interfaces are created through encapsulation, when certain methods are exposed and made accessible to objects of other classes. Interfaces are not classes, but are used to describe behaviours. Can also indicate a keyword in Java, that indicates an interface is being created.
Interface inheritance	Interfaces cannot be instantiated, but can be used as reference type for the object of an implementing class, and can be extended by another interface. This is known as interface inheritance.
Liskov substitution principle	A principle that states that a subclass can replace a superclass, if and only if the subclass does not change the functionality of the superclass.
Local variables	An idea where subroutines or procedures could contain nested procedures, allowing each to have their own variables.
Loosely coupled	Occurs if a module finds it easy to connect to other modules. Considered a characteristic of good design.
Low cohesion	Occurs if a module tries to encapsulate more than one purpose, or if it has an unclear purpose. This characteristic is considered bad design.
Maintainable	Maintainability refers to the ease of modifying a software system or component in order to correct or improve faults, performance, or other attributes, or the ability to adapt to a changed environment.
Methods	Methods manipulate the attribute values or data in an object of a class, in order to achieve the actual behaviours.
Model checking	A systematic check of a system's state model in all its possible states.
Modelling Phase	A phase in model checking, where the model description is entered, sanity checks are performed, and desired properties are described. This would be provided in whatever programming language your system uses.
Models	A visual representation of a software design.

Modula-2	A programming language from the mid-1970s that provided a means to organize problems and allowed developers to create multiple, but unique, copies of their abstract data types more easily.
Module	A program unit that includes classes and the methods within them.
Multiple inheritance	A form of inheritance that occurs when a subclass has two or more superclasses. Java doesn't support multiple inheritance.
Namespace	A namespace is a package that classes can be organized and represented by.
Object-oriented analysis	A type of analysis that identifies key objects in a conceptual design problem.
Object-oriented design	The method revolving around perceiving concepts as objects, which requires programmers to plan out their code to have better software that is flexible, reusable, and maintainable. It involves refining the details of objects, including attributes and behaviours.
Object-oriented modelling	Involves the practice of representing key concepts through objects in the software.
Operations	A section of UML class diagrams, where behaviours of the abstraction are defined. This is equivalent to methods in a Java class.
Override	Occurs when a subclass can provide its own implementation for an inherited superclass's method. The subclass's methods override the superclass's.
Package	The means by which Java organizes related classes into a single namespace.
Parts	Portions or fragments with different functionalities that can be combined together to form a whole thing. Related to decomposition.
Pascal	A programming language from the 1970s that supports the notion of an abstract data type.
Polymorphism	In object-oriented languages, polymorphism is when two classes have the same description of a behaviour, but the implementation of the behaviour may be different.

Private	An access modifier and level in Java. Private attributes and methods are not accessible by any class other than by the encapsulating class itself.
Programming Paradigms	A typical pattern or example of the thought-process or theory underlying programming languages and coding within a specific time frame.
Properties	A section of UML class diagrams where attributes of the abstraction are defined. This is equivalent to Java's member variables.
Protected	An access modifier and level in Java. A protected attribute or method is one that can only be accessed by the encapsulating class itself, all subclasses, and all classes within the same package.
Public	An access modifier and level in Java that indicates that an attribute or method can be accessed by any other class in the system. The implementation of the behaviour for the method cannot be changed by other classes at this level, although attributes can be retrieved and modified by other classes.
Quality attributes	Aspects of the software that impact how the software functions such as performance, convenience, and security.
Requirement	A condition or capability that must be implemented in a product based on your client's request.
Responsibility	A task that a component needs to perform. For example, a screen might have a responsibility to turn on.
Reusable	Reusability refers to the ability to use existing products and byproducts of software development, including code, designs, and documentation, more than once in the software development process.
Rule of least astonishment	A rule related to abstraction that says that an abstraction should capture the essential attributes and behaviour for a concept with no surprises and no definitions that fall beyond its scope.
Running Phase	A phase in model checking, when the model checker is run to see how the model conforms to the desired properties expressed in the Modelling Phase.

Sanity checks	Quick checks that are easy to do because they come from very clear and simple logic.
Separation of concerns	A principle of software design that suggests that software should be organized so that different concerns in the software are separated into different sections and addressed effectively.
Sequence diagrams	A diagram used to visually represent how objects in a program interact with each other to complete a specific task.
Service-oriented Architecture	A style of software design that examines architectures for web applications.
Software architecture	An aspect of the software development process that examines the high-level aspects of a system.
Software design	An aspect of the software development process that examines the lower-level aspects of a system.
Specialized	A subclass with customized or special behaviours.
State	The way an object exists at a particular point in time. The state of an object is determined by the values of its attributes.
State diagrams	A visual technique used to express how a single object in a system behaves in response to a series of events in the system.
State model	An abstract state machine that can be in one of various states.
Subclass	A subclass is a child class that inherits characteristics and attributes from another class, but also presents its own specialized functions to separate it from other child classes. Subclasses are specialized classes.
Super	A keyword in Java that allows a subclass to access the superclass's attributes, methods, and constructors.
Superclass	A superclass is a parent class that other classes might inherit characteristics and attributes from. Superclasses are generalized classes.

Technical design	The process of design where technical details are planned out around concepts and components, which explain how responsibilities are met. Technical design can be used to create code, often with the help of technical diagrams.
Technical diagrams	Provide a visual expression of technical details for how requirements and concepts will be satisfied in a software product.
Tightly coupled	A kind of design complexity that occurs when a module is highly reliant on one or more other modules. Tightly coupled modules are considered bad design.
Transitions	A shift from one state to another, as triggered by an event.
Type	A class denotes a type for its objects. The type signifies what these objects can do through public methods.
Unified Modelling Language (UML) class diagram	A visual notation for expressing or communicating technical design and models in software production. There are several different kinds of UML class diagrams.
Whole	When several different parts of a thing are brought together to create a cohesive, working thing. Related to decomposition.