Object Oriented Design

Status	Completed

Lecture 1: Object-Oriented Analysis and Design

02/11/24

Software architect and design roles in industry

- Like many roles in industry, software architect or design roles can look very different from company to company.
  - Typically, a software designer role would be responsible for outlining a software solution to a specific problem.
  - A software architect would be responsible for choosing the appropriate technologies and determining how they interact with each other.
    - Their job is to be the interface between the customer and the engineering team, and are responsible for the overall integrity of the project.
      - Soft skills are very important for communication surrounding a project.
    - It is important to know what technologies will be useful for a particular project.
      - One way to stay up to date with useful technologies is to look at what larger companies are doing.
  - Software design looks at the lower-level aspects of a system, where software architecture looks at the higher-level aspects.
    - The most important principal in software design is to “keep it simple, stupid”, and to avoid overcomplication.
      - Simplicity increase the chances of implementing software correctly.
- Software design architecture is important for creating stable, long-lived software.
  - Most major software failures can be traced back to “bad” software architecture or design.
- The best way to become a software architect is to read a lot, code a lot, and learn from your mistakes.

Object-oriented modeling

- Object-oriented modeling (OOM) involves representing key concepts through objects in software.
  - OOM keeps code organized by having related details and specific functions in distinct places.
    - This creates flexible and reusable code.

Software requirements, conceptual and technical designs

- Developing software is an iterative process that solves a problem and produces a software solution.
  - Diving immediately into programming is a common point of failure for software.
  1. The first step to designing good software is eliciting requirements, which means asking questions to gain a better understanding of the design requirements.
  2. The next step is to create a conceptual design with a conceptual mockup, and then a technical design with a technical diagram.

    a. Conceptual mockups identify major components and the connections between them.
        i. The clearer a conceptual design is, the easier the technical design will be.
        ii. Mockups are typically hand-drawn or digitally designed.
    b. The technical design specifies the technical details of every component.
        i. Components are broken down into smaller sub-components until they are specific enough to be designed in detail.
        ii. When components are refined enough, they turn into collections of functions, classes, and other components.
            1. These simple problems can then be individually delegated and implemented.

<u>Expressing requirements</u>

- Often times, the requirements for a software system can be determined using user stories.
    - User stories state a requirement for an end user in natural language.

> As a _____, I want to _____ so that ______.

- For example:

> As an *online shopper*, I want to *add an item to my shopping cart*, so that *I can purchase it*.

- Objects usually identified by the nouns in the sentence, such as *online shopper, item*, and *shopping cart*.
- Methods for these objects are usually identified by the verbs, such as *add*, and *purchase*.

<u>Categories of objects is design</u>

- Entity objects are objects that correspond to some real-world entity in the problem space.
- Boundary objects are objects that sit on the boundary between different systems.
    - For example, an object that uses an API would be a boundary object, as it sits on the boundary between an application and the internet.
    - An object with the responsibility of showing output or taking input, such as a user interface, would also be a boundary object.
- Control objects are objects which are responsible for the coordination of other objects.
- Organizing software into these different types of objects will allow for more flexible, reusable, and maintainable code.

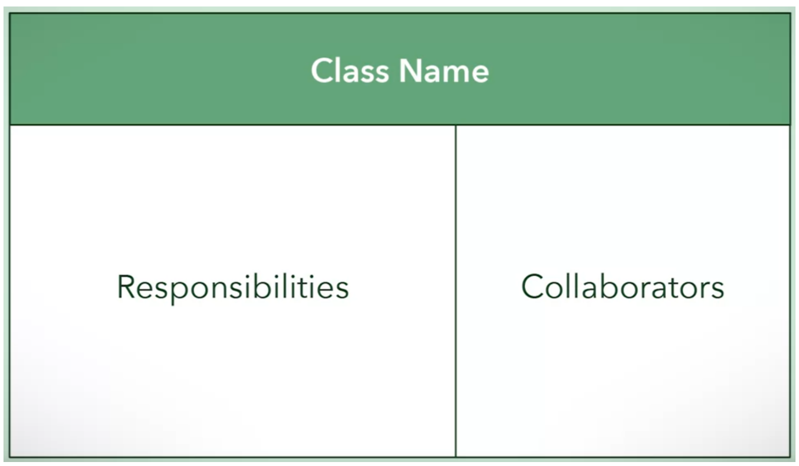
<u>Competing qualities and trade-offs</u>

- Certain decisions when designing software involve trade-offs in different quality attributes.
    - This could be the trade-off between performance and security, or code efficiency and readability.
- One job of a software architect is to advocate for quality attributes within the context of the business.
    - The major balance is between quality and the time to market.
- It is important to gain many perspectives to create the best possible solution for all parties.
    - Qualities should be verified through reviews, tests, and user feedback.

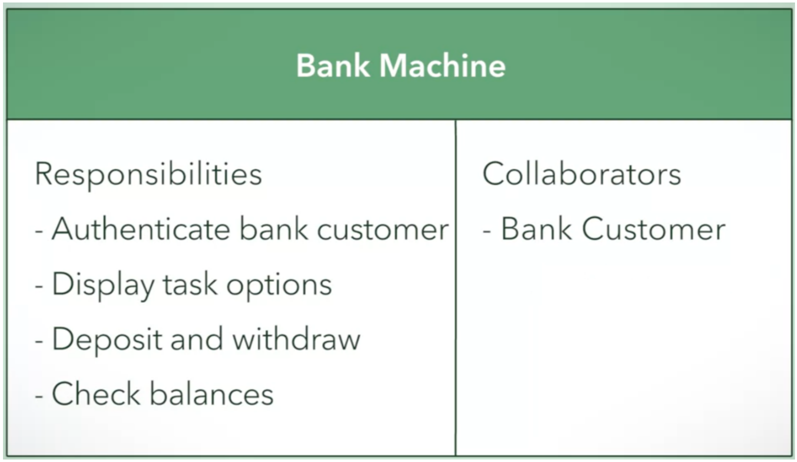
- Functional requirements describe what a system or application is expected to do.
    - Design requirements will put constraints on a software.
- Non-functional requirements describe how well a system or application does what it does, either generally or in particular situations.
    - It is important to also consider performance, resource usage, and efficiency.

Record, organize, and refine components

- Components, connections, and responsibilities for software are identified, which form a conceptual design.
- Class, responsibility, collaborator (CRC) cards organize components into classes, identify their responsibilities, and determine how they will collaborate with each other.
    - These are used to record, organize, and refine components in a design.
    - Components and responsibilities are placed in the collaborators and responsibilities sections respectively.
    - The small size of CRC cards forces one to break down components until each one has only a single responsibility.
    - Using CRC cards helps identify new components and connections that may not have been considered before.



    - For example, this would be the CRC card for a *bank machine* component.
    - Responsibilities generally start with verbs, and collaborators are generally nouns.



Lecture 2: Object-Oriented Modeling

02/11/24

Models: bridging concepts and solutions

- It is important to design software in a structure that makes sense to both end users and developers.
- For many complex problems, it makes sense to use object-oriented programming.
    - The goal of software design is to construct and refine models of all the objects.
    - Therefore, object-oriented analysis and object-oriented design need to be done before OOP.
- Object models are often expressed in a visual notion called Unified Modeling Language (UML).
    - Much like software, models are expected to be flexible, reusable, and maintainable.

Languages evolution

- Programming paradigms describes the style in which a programming language achieves objectives.
    - Imperative paradigms break up large programming into smaller programs called subroutines.
        - Languages like Fortran and COBOL use an imperative paradigm.
    - Abstract data types (ADT) are custom datatypes that are not built into a programming language.
    - Object oriented design (OOP) makes ADTs easier to write by structuring a system around ADTs called classes.
        - These classes are able to inherit/extend from one another.
        - Languages like Java, C#, and C++ use OOP.

Abstraction

- Abstraction simplifies concepts in the problem domain to their essentials within some context.
    - In object-oriented design, the context is important for deciding how a class will be abstracted.
- The rule of least astonishment states that abstraction should capture the essential attributes and behavior for a concept with no surprises and no definitions that fall beyond its scope.
- Attributes correspond to the data an object has.

- Behaviors correspond to the methods an object has.
- Abstractions are not fixed, but a direct result of the problem they are created for.

Encapsulation

- Encapsulation involves bundling attribute values (data) and behaviors (methods) together into a self-contained object.
  - Only certain data and methods from these objects are able to be exposed from outside the object through an interface, and others are restricted.
    - In practice, all data and methods are restricted by default.
    - This is helpful for increasing code security.
- An object's data should only contain what is relevant for that object.
- Black box programming is when the behavior of a function or object is not dependent on its implementation.
  - Changing the implementation without changing the behavior is called refactoring.
- Encapsulation achieves abstraction, and therefore decreases complexity.

Decomposition

- Decomposition means either taking a whole component and dividing it into separate components with different functionalities, or conversely, combining many components with different functionalities into one component that achieves a common purpose.
  - This allows one to further break down problems into pieces that are easier to understand and solve.

Generalization

- Generalization reduces redundancy when solving problems by generalizing data or behaviors.
  - Generalization is achieved in OOP through inheritance.
    - Repeated or common characteristics between multiple classes are factored out into another.
      - For example, a `Cat` class and a `Dog` class would both inherit common data and behaviors from an `Animal` class.
    - Another benefit of generalization is that creating additional similar classes is easier through sub-classing.
      - For example, a new `Bird` class could be sub-classed from `Animal`.

Abstraction in Java and UML

- UML class diagrams represent designs in more detail than CRC cards.
  - Each class in a class diagram has a class name, properties section, and operations section.
    - The properties section lists the names and types of variables in a class.
    - The operations section lists the names, parameters, and return types of methods in a class.

> *All class diagrams in these notes use Mermaid integrated into Notion.*

```
classDiagram
    Food
```

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

        isOnSale(): boolean
  }
```

- Unlike CRC cards, class diagrams are more code focused and are too detailed for conceptual design.

Encapsulation in Java and UML

- A `+` sign is used to denote accessible (public) data or methods on class diagrams, and a `-` sign is used to denote hidden (private) data or methods.

```
classDiagram
    Student
    class Student {
        -gpa: float
        -degreeProgram: String

        +getGPA(): float
        +setGPA( float ): void
        +getDegreeProgram(): String
        +setDegreeProgram( String): void
  }
```

```
// An example of encapsulation where a person object has its data and methods colle
// This object's data is protected by an interface of getter and setter methods
public class Person {
    private String name;
    private int age;

    public void Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName () {
        return this.name;
    }

    public void setName (String name) {
        this.name = name;
    }
```

```
    public int getAge () {
        return this.age;
    }

    public void setName (int age) {
        this.age = age;
    }
}
```

- With any data, a protective layer is necessary to prevent unintended manipulation.
  - Getter methods and setter methods create a controlled interface for interaction with an object and its data.

02/12/24

Decomposition in Java and UML

- The three relationships found in decomposition are association, aggregation, and composition.
- Association is a lose relationship between two objects, which may interact with each other for some time.
  - A straight line between two UML objects denotes an association.
  - A number `x` in `x..*` denotes that an object is associated with `x` or more of its associated object.
  - One object does not belong to the other, and can exist without the other.

```
---
title: Association
---
classDiagram
    direction RL
      Person "X..*" -- "X..*" Airline
    end
  class Person {
  }
  class Airline {
  }
```

```
// An example of association in decomposition where Food and Wine do not belong to
public calss Wine {
    public void pair (Food food) {
        ...
    }
}
```

- Aggregation is a relationship where a whole has parts that belong to it.
  - This relationship is “weak”, in that although the parts belong to the whole, they can also exist independently.
  - An empty diamond denotes which object is the “whole” in the relationship, and is the symbol for aggregation.

- A number `x` in `x..*` denotes that an object is associated with `x` or more of its associated object.

```
---
title: Aggregation
---
classDiagram
    direction RL
      Airliner "0..*" --o "0..*" CrewMember
    end
  class Airliner {
  }
  class CrewMember {
  }
```

```
// An example of aggregation in decomposition where Airliner and CrewMemver are in
public class Airliner {
    private ArrayList<CrewMember> crew;

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

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

- Composition is an exclusive relationship where a whole has parts that belong to it.
    - This relationship is “strong”, in that the parts could not exist independently without the whole.
        - Most often, the parts can only be accessed through the whole.
    - A solid diamond denotes which object is the “whole” in the relationship, and is the symbol for composition.
    - A number `x` in `x..*` denotes that an object is associated with `x` or more of its associated object.

```
---
title: Composition
---
classDiagram
    direction RL
      House "0..*" --* "0..*" Room
    end
  class House {
    }
  class Room {
  }
```

```java
// An example of composition in decomposition where Human and Brain are in a "set"
// And the set is exclusive to the whole, as in the set could not exist without the
public class Human {
    private Brain brain;

    public Human () {
        brain = new Brain();
    }
}
```

- Decomposition is simply about whole objects containing part objects.

<u>Generalization with inheritance in Java and UML</u>

- A solid lined arrow indicates that two classes are connected by inheritance.
  - The super-class is at the head of arrow, and the sub-class is at the tail.
    - Inherited data or methods are not listed for the sub-classes to avoid redundancy.
  - A `#` symbol is used to indicate that a class's attributes are protected.
    - In Java, a protected attribute or method can only be accessed by the encapsulating class itself or its subclasses.

```
---
title: Generalization with inheritance
---
classDiagram
    direction TB
        Animal <|-- Dog
    end
   class Animal {
        #numberOfLegs: int
        #numberOfTails: int
        #name: String

        +walk()
        +run()
        +eat()
   }
   class Dog {
        +playFetch()
   }
```

```java
// Generalization of a Dog class by inheriting from an Animal super-class
public class Dog extends Animal {
    public Dog (string name, int legs, int tails) {
        super(name, legs, tails);
    }
```

```
    public void playFetch () {
        ...
    }
}
```

- Inheritance in Java is declared using the keyword `extends`.
    - An explicit constructor allows one to initialize object attributes (data), where an implicit constructor will set all object attributes to `0` or `Null` on instantiation.
    - `super` is called so that a super-class can initialize its attributes for its sub-class.
    - Sub-classes can override super-class methods with their own definitions.
- In Java, subclasses can only inherit from one superclass.

Generalization with interfaces in Java and UML

- The type of an object signifies what that object can do.
- Unlike classes, Java interfaces only declare method signatures.
    - They do not declare constructors, attributes, or method bodies.
- When a class is provided with an interface, it is expected to behave like that interface by implementing all the methods the interface provides.
- Interfaces in Java are implemented using the keyword `implements`.
    - It is convention to use `I` at the beginning of a class name to denote an interface.

```
// The "I" denotes that IAnimal is an interface, not a class
public interface IAnimal {
    // None of these behaviors are defined, as that is the job of the classes that
    // Only methods signatures can be defined, as attributes do not describe behavi
    public void move ();
    public void eat ();
}
```

- A solid dotted arrow indicates that an interface and a class are connected.
    - The interface title is headed by `<<interface>>` to explicitly denote an interface.
    - The interface is at the head of arrow, and the class is at the tail.

```
---
title: Generalization with interfaces
---
classDiagram
    direction TB
        Animal <|.. Dog
    end
   class Animal {
        <<interface>>
```

```
        +eat()
    }
    class Dog {
        +eat()
    }
```

- Interfaces are a way to implement polymorphism, in that the behavior of the same method can be different in different classes.
    - In this way, they are most similar to abstract classes.
- Interfaces can inherit from other interfaces using `extends`.
    - However this capability should not be abused, and should only be used if the sub-interface can make full use of the methods they inherit from the super-interface.

```
// The Cat and Dog method both implement the Animal interface, yet their implementa
public class Dog implements IAnimal {
    public void speak () {
        System.out.println("Bark!");
    }
}

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

- Java does not support multiple inheritance as to avoid data ambiguity.
    - However, classes can implement multiple interfaces.
    - Multiple interfaces implemented are separated by a `,`.

```
// Multiple implementations of interfaces for a Person class
public class Person implements IPublicSpeaking, IPrivateConversation {
    public void speak() {
        System.out.println("This is fine");
    }
}
```

Lecture 3: Design Principles

02/15/24

Coupling and cohesion

- Once design complexity exceeds what developers can mentally handle, bugs will occur more often.

    - Therefore, it is important to have a way to evaluate design complexity.
    - The metrics used to evaluate design complexity are coupling and cohesion.
- Modules refers to classes and the methods within them.
    - Coupling focusses on the complexity *between* modules.
        - If a module is highly reliant on other modules, that module is tightly coupled.
        - If a modules is easy to connect to other modules, that modules is loosely coupled.
        - Three factors should be considered when coupling a module:
            1. Degree is the number of connections a module has with other modules.
                a. Degree should be minimized.
            2. Ease is how obvious the connections between a module and other modules are.
                a. Connections should be obvious.
                b. For example, not respecting encapsulation can lead to low ease, resulting in tight coupling.
            3. Flexibility refers to how interchangeable other modules connections are for a module.
                a. Modules should be able to be easily replaced.
    - Cohesion focusses on the complexity *within* a module.
        - In other words, it represents the clarity of responsibilities within a module.
        - If a module performs one task and nothing else, or has a clear purpose, that module has high cohesion.
        - If a module performs multiple tasks or has an unclear purpose, that module has low cohesion.
    - In general, there will always be a balance between low coupling and high cohesion.

02/17/24

<u>Separation of concerns</u>

- Separation of concerns is the principle of separating programs into sections with their own distinct functionality.
    - A concern is anything that matters in providing a solution to a problem.
    - Separation of concerns is a key idea that applies throughout object-oriented modelling and programming.
- A high number of concerns in a single class should be avoided, as this leads to low cohesion within the class.
    - Classes should be separated into functionally distinct subclasses when necessary.
    - Although classes become more cohesive through separation of concerns, this process often increases coupling between classes.

<u>Information hiding</u>

- A module should only have access to the minimum amount of information that it needs to perform its functionality.
    - Information can be limited throughout a system using information hiding.
        - Information is hidden through encapsulation, and revealed through interfaces.
        - Encapsulation essentially hides information behaviors, as the only access to a module’s data is through interface specific methods.

- Information hiding allows work to be done on modules separately, without the implementation details of other modules needing to be known.
    - This decreases module coupling and increases module security.
- In Java, access modifiers like `public`, `protected`, and `private` change which classes are able to access attributes and behaviors.
- A package are the means by which Java organizes related classes into a single namespace.
    - `public` behaviors and attributes are accessible by any class in a system.
    - `protected` behaviors and attributes are accessible to the encapsulating class itself, its subclasses, and the classes in the same package as that class.
    - `default` behaviors and attributes are accessible to the encapsulated class itself and the classes in the same package as that class.
        - It is also called the no modifier access because it does not need to be explicitly declared.
    - `private` behaviors and attributes are only accessible to the encapsulated class itself.

Modifier	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
`no modifier*`	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Conceptual integrity

- Conceptual integrity concerns creating *consistent* software.
    - It is important for making decisions about software design so that even if multiple people are working on a system, the direction of the work follows as if only one person were working on it.
    - Specifically, everyone on a team should agree about certain design principles and conventions.
    - Conceptual integrity does *not* mean that each person should work as a mindless drone with no opinion.
- A well defined design/system architecture is important for maintaining conceptual integrity.
- Agile development strategies like daily stand-ups and sprint retrospectives can help with communication between team members, which in turn increases conceptual integrity.
- Code reviews are systematic reviews of written code, which are used to find mistakes in software or inconsistencies in naming conventions.
- Design patterns are conventional structures to solve common design issues.
- In short, conceptual integrity is important for maintaining the structural integrity of a software system, so that it is easier to build upon and make changes to it.

Inheritance issues

- Generalization and inheritance are often the most difficult topics to master in OOP and OOM.

- While inheritance is valuable for writing clean and reusable software systems, the misuse of inheritance can lead to more problems than solutions.
- One should follow key points to make sure that inheritances is not being misused.
    1. Inheritance should *not* be used simply to share attributes or behaviors without further specializing a class's sub-classes.
        a. A good indication of misuse is when a sub-class only contains references to `super`, with no other expressions.

```
classDiagram
    direction RL
        note for PepperoniPizza "PepperoniPizza provides no new functionality fr
    Pizza <|-- PepperoniPizza
    end
    class Pizza {
        Pizza(topping: String)
    }
    class PepperoniPizza {
        PepperoniPizza(topping=pepperoni: String)
    }
```

1. Breaking the Liskov Substitution Principle, stating that a sub-class can replace a super-class only if the sub-class does not change the functionality of the super-class.
    a. In other words, sub-classes would be able to inherit behaviors or attributes that do not make sense for them to inherit.

```
classDiagram
    direction RL
        note for Whale "Even though Whale is a sub-class of animal, it does not
    Animal <|-- Whale
    end
    class Animal {
        walk()
    }
    class Whale {
    }
```

- Generally, it is important that OOM design concepts are used to design flexible and maintainable systems, but can create rigid and failure-prone systems if they are used improperly.

<u>UML sequence diagrams</u>

- Sequence diagrams show how objects in a program interact with each other to complete tasks.
    - Knowing how to break down a system into classes is necessary for creating sequence diagrams.
    - Sequence diagrams are commonly used as planning tools before a development team starts programming.

- A sequence diagram has three main components:
  1. Boxes are used to represent roles played by objects, often times by the name of the object.
     - Objects in the sequence diagram are called participants.
     - People that interact with objects, or actors, are typically drawn as stick figures.
  2. Vertical lines, also called lifelines, represent the flow of time.
     - Rectangles on an object's lifeline shows when that object is activated.
       - An object is activated whenever is send, receives, or is waiting for a message.
  3. Horizontal lines show how objects interact with each other.
     - Messages/method calls are denoted with a solid line.
     - Return values are denoted with a dashed line.
- To keep the diagram easy to understand, actions should start out going from left to right.

```
sequenceDiagram
    actor Customer
    participant Payment
    participant Kitchen

    note over Payment: "This example shows the sequence of actions between classes
    activate Customer
    Customer->>Payment: order(burger)
    activate Payment
    Payment->>Kitchen: queue(burger)
    deactivate Payment
    activate Kitchen
    Kitchen-->>Customer: burger
    deactivate Kitchen
    deactivate Customer
```

- Loops and alternative processes can also be represented in sequence diagrams.
  - `alt` denotes an alternative process.
    - `else` denotes if a condition in a sequence diagram is false, and shows what other sequences may occur in that case.
  - `loop` denotes a loop, along with a conditional statement for the loop

```
sequenceDiagram
    actor TV Viewer
    participant Remote
    participant TV

    note over Remote: "A TV viewer will change channels until they are on the chann
    alt TV Viewer knows what channel they want.
        activate TV Viewer
```

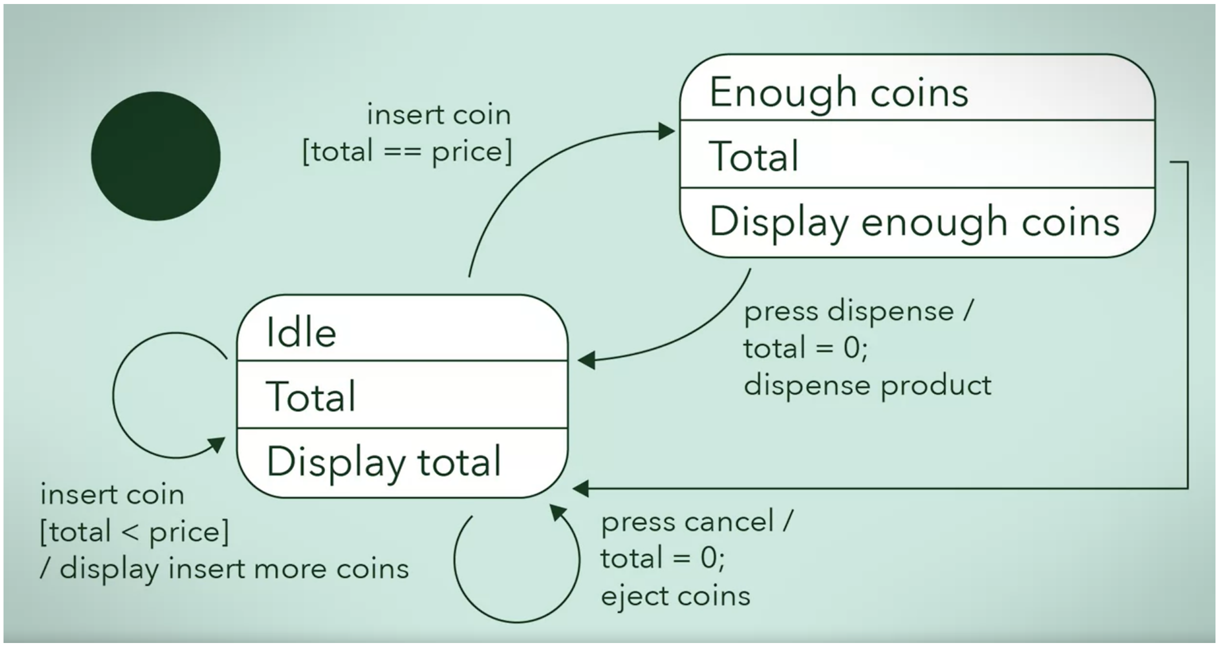
```
        TV Viewer->>Remote: pressNumbers(number)
        activate Remote
        Remote->>TV: changeChannel(number)
        deactivate Remote
        activate TV
        TV-->>TV Viewer: show change of channel
        deactivate TV
        deactivate TV Viewer
    note over Remote: "If the viewer does not like the channel, they will change it
    else else
        loop TV Viewer does not like the channel
            activate TV Viewer
            TV Viewer->>Remote: pressUpOrDown(arrow)
            activate Remote
            Remote->>TV: changeChannelUpOrDown(arrow)
            deactivate Remote
            activate TV
            TV-->>TV Viewer: show change of channel
            deactivate TV
            deactivate TV Viewer
        end
    end
```

02/18/24

UML state diagrams

- State diagrams describe how a system behaves and responds to events that occur.
  - State diagrams show different states of a system as nodes connected by events.
- State diagrams can also describe how a single object behaves in response to events in a system.
- States are indicated by rounded rectangles.
  - Each state has three components:
    1. A name that is meaningful for the state of the object.
    2. Variables that represent data relevant to the state of the object.
    3. Activities that are performed when in a certain state.
       - There are three types of activities for a state:
         - Entry activities, which occur when a state is just entered from another state.
         - Do activities, which occur one or more times when an object is in a certain state.
         - Exit activities, which occur when a state is exited and moves on to another state.
  - The starting state of a state diagram is indicated by a solid circle.
  - Termination, a process being completed, is represented by an open circle with a filled circle inside of it.
    - Not all state diagrams need a termination, as they might run continuously.

- Transitions between states are shown with arrows between nodes.
    - Transitions may be labeled with a condition, to show that the transition happens only when that condition is true.



<u>Model checking</u>

- There are many methods to verify that a system is working as intended, such as running tests to check system behavior.
- Model checking is a systematic check of a system's state model in all possible states.
    - Model checking is done after software design, but before a software is deployed.
    - Model checks are done using model checking software.
- There are two complementary ways for reasoning about software.
    1. Programs can be thought of as a mathematical artifact with provable properties based on a programming language's semantics.
    2. For programs with many special cases/states, all the cases/states should be systematically enumerated over to check for errors.
        - In other words, a program can be proven sound if it never in certain states.
        - A rule that software is *required* to satisfy, is that it must not produce a deadlock.
            - Deadlocks are situations where a program cannot continue because two tasks are waiting for the same resource.
        - Model checking is used for programs that are expressed this way.
- Model checking is essentially a state checker, which explores all possible ways for proceeding through a program.
    - If a model checker finds that a system is sound in all the necessary ways, the model passes.
    - If not, the model checker gives examples where the unwanted cases happen.

- One problem with model checking is state space explosion.
    - As models become larger, their possible states become exponentially complex.
    - Therefore efficient model checking relies on extracting only the necessary pieces of a program that need to be modelled.
        - Model checking relies on the small world assumption, where parts of a system go wrong in small cases.
        - Confirming that a model works on small cases confirms, to a reasonable degree, that the model works on larger cases.
- Model checkers work by producing a state model from a program's code.
    - State models are abstract state machines that can be in one of various states.
        - State models check if they conform to certain behavioral properties.
- There are three phases to model checking.
    1. The modelling phase, where the model description and desired properties are entered.
        - Sanity checks, which are quick checks that derive from clear and simple logic, can be done in this phase.
    2. The running phase, where the model is checked to see if it conforms to the desired properties specified in the modelling phase.
    3. The analysis phase, where it is checked if all the desired properties are satisfied.
        - Any violations of these properties are called counterexamples, and the model checker will provide descriptions of them if these violations are raised.