Module 3: Design Principles

Evaluating DC

Keep modules simple when programming

* Evaluate DC its manageable
* **Design Complexity** = **both** **classes** & the **methods within them**
  + Referred to as modules
* Metrics used to evaluate DC are **coupling** & **cohesion**

Coupling

* **Focuses on complexity between a module & other module.**
* Coupling is either:
  + **Tightly coupled**:
    - Module too reliant on another module
  + **Loosely coupled**:
    - Module finds it easy to connect to other modules through defined interfaces. Good design.
* To evaluate **coupling** of **module,** metrics to consider are:
  + **Degree**
    - # of connections between module & others
    - Degree should be small for coupling
  + **Ease**
    - How obvious are the connections between the module & others
    - Connections should be easy to make without needing to understand the implementations of other modules.
  + **Flexibility**
    - How interchangeable the other modules are for this module.
    - Other modules should be easily replaceable for something better in the future.
* Signs that a system is tightly coupled and has a bad design are:
  + 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 – no interchange

Cohesion

* **Focuses on complexity within a module & represents the clarity of the responsibility of a module.**
* Cohesion can work between 2 extremes:
  + **High cohesion:**
    - Module performs 1 task and nothing else.
    - Clear purpose
    - Good design has high cohesion.
  + **Low cohesion:**
    - Module encapsulates more than one purpose
    - Unclear purpose
    - Low Cohesion
    - **If module has more than one responsibility. Good idea to split the module**

Separation of Concerns

To understand why decomposition is necessary, the principle of **separation of concerns** must be examined.

Concern

* **Anything that matters in providing a solution to a problem.**
* **Separation of concerns, is keeping different concerns in your design separate**
* Separation of concerns is a key idea that underlies O-O modeling & programming.
* When Addressing concerns separately, more cohesive classes are created and the Design principles **AB|EN|DE|GE** are enforced.
  + **AB**:
    - Occurs as each concept is separated with its own relevant attributes & behaviours
  + **EN**:
    - Attributes & behaviour are gathered into their own class.
    - Access to the class from the rest of the system & implementation are separated.
  + **DE**:
    - Occurs as whole class can be separated into multiple classes.
  + **GE**:
    - Commonalities are recognized, & subsequently separated and generalized into a superclass

Examples:

Text

Description automatically generated

Using separation of concerns, we can identify that the Smartphone class has 2 concerns.

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

With concerns identified, it’s possible to separate them into their own cohesive classes and **EN** all the details about each concern into functionally distinct & independent classes.

Diagram

Description automatically generated

Text

Description automatically generated with low confidence

Text

Description automatically generated

In this example, **separation of concerns** was used:

* Separating out notions of camera & phone through **GE** & defining 2 interfaces.
* Separating out the functionality for a first gen camera & traditional phone by applying **AB** & **EN** & defining 2 implementing classes.
* Applying **DE** to the smartphone so the parts are separated from the whole.

Information Hiding

A well-designed system is well organized and uses design principles to limit the amount of information available to modules so that only the minimum amount needed to use them correctly is provided. This is known as **information hiding**.

**Information hiding** a technique used in SD to limit access to sensitive data or hide changeable details such as algorithms or data representations. This practice is commonly associated with sensitive information. Though, assumptions 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.**

**Good rule of thumb** things that might change like implementation details, should be hidden, things that do not change like assumptions are revealed through interfaces

Info hiding is closely associated with encapsulation.

**EN** is a technique in OOP that bundles attributes and behaviors into a class, while also managing access to the class through interfaces and limiting access to certain behaviors or functions.

Info hiding can be accomplished through **access modifiers.** Access modifiers change which classes can access **BE|AT**. They also determine which **BE|AT** a superclass will share with its subclasses.

Four Levels of Access in Java:

* **Public**
  + Accessible by any class in your system.
  + Other classes can retrieve & modify **AT** or change.
  + Methods can be **public**,
    - Doesn’t allow other classes to change the implementation of the **BE** for the method.
    - Implementation remains hidden through **EN**.
* **Protected**
  + **BE|AT** that are **protected** are not accessible to every class in the system.
  + Only available to the **EN** class itself, all subclasses & classes within the same package.
  + **Packages** – Organizes related classes into a single namespace.
* **Default**
  + Only allows **BE|AT** to subclasses or classes that are part of the same **package** or **EN**
* **Private**
  + Not accessible by any other class other than by the **EN** class itself

Conceptual Integrity

* **Conceptual integrity** is a concept in software development that refers to creating a consistent, cohesive, and unified design of a software system.
* This is achieved by agreeing to use certain design principles and conventions, so that even if multiple people work on the system, it appears as if it was created by one person.

Multiple ways to achieve conceptual integrity:

* **Communication**
  + Effective communication among team members helps maintain consistency in the code by discussing & agreeing on libraries & methods used for addressing issues.
  + Agile development practices such as daily stand-up meetings & sprint retrospectives can foster this communication.
* **Code Reviews**
  + Systematic examination of written code by developers, like peer review in academic writing.
  + This helps identify mistakes in the software and create consistency among different developers' code.
* **Using certain design principles & programming constructs**
  + like Java interfaces and design patterns can help maintain conceptual integrity by creating consistency in the software and defining expected behaviors.
* **Having well-defined design or architecture underlying the software** 
  + A well-defined design or architecture underlying software also increases conceptual integrity.
* **Unifying concepts**
  + finding commonalities and treating them similarly can also increase conceptual integrity.
* **Having a small core group that accepts each commit to the code base.** 
  + help solve design issues and create consistency.

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

Generalization Principles

* **GE** & **IN** difficult topics to master in OOP & modelling.
* **IN** powerful design tool that helps create clean, reusable, and maintainable software systems.
  + Its misuse can lead to poor code.
  + May create more problems than they solve.
* To identify if **IN** is being misused
  + Keep a couple of **GE** principles in mind
* 1st Principle:
  + Am I using **IN** to simply share **BE|AT** without further adding anything special in my subclasses
    - **Ans Yes**, its being misused
* 2nd Principle:
  + The **Liskov substitution principle** states that a subclass should be able to replace a superclass without altering its functionality. Misuse of **IN** occurs when a subclass changes the behaviors of the superclass.
* Diagram

  Description automatically generatedThe Java Stack class, which inherits from Vector, has behaviors not expected from a stack, such as accessing elements at specific indices. This is an example of bad **IN** in Java.

Diagram

Description automatically generatedBad Example

Good Example

Specialized UML Class Diagrams

The use of UML class diagrams for technical design was reviewed. There are different types of UML class diagrams, including specialized versions that can enhance technical design.

UML Sequence Diagrams

**UML sequence diagrams** are a type of UML diagram used for planning in software design to show object interactions and complete tasks. They are like maps of conversations between objects.

**Sequence diagrams** can help you visualize the classes you will create in your software & determine the functions that will need to be written

Diagram

Description automatically generated

In UML sequence diagrams:

* **Boxes** represent a role played by an object, usually named after the class.
* **Lifelines** are used to indicate an object over time.
* **Solid line arrows** show messages sent between objects, with description above arrow.
* **Dotted line arrows** show return of data/control to initiating objects, with description above arrow.
* **Method activation** is indicated by small rectangles on an object's lifeline.
* **Actors,** represented as stick figures, can also be included in sequence diagrams if they interact with objects.

Diagram

Description automatically generated

UML State Diagrams

* **State diagrams** describe system behavior and response by showing changes in the states of a system or object.
* They depict changing states of an object in response to events and a **state** represents the object's condition at a particular time based on its attributes.

A picture containing text

Description automatically generated

* + **State names** should be short, meaningful titles for the state of the object. Each state should have a least a state name.
  + **State variables**, data relevant to the state of the object.
  + **Activities** are actions that are performed when in a certain state. 3 types of activities:
    - **Entry Activities** – Actions that occur when the state is just entered from another state.
    - **Exit Activities** – Actions that occur when the state is exited & moves on to another state.
    - **DO Activities** – Actions that occur while the object is in a certain state.
  + Arrows indicate **transitions** from one state to another

Graphical user interface, website

Description automatically generated

**State diagrams** help determine events & object behavior during its lifetime, showing changes in state visually. They simplify understanding of object behavior & events compared to reading code.

Model Checking

Different testing techniques used include unit testing, beta testing, simulations, and model checking. **Model checking** is a systematic examination of a system's **state model** in all possible states and helps identify errors missed by other tests.

Model checking involves simulating events to identify software errors by checking all its states. This helps detect any violations of rules in state model behavior.

Model checks are done by model checking software, with various types available for different languages, some free. Model checking is performed during software testing.

Model checking helps detect deadlocks in software, where two tasks wait for the same resource and the system can't continue. The model checker simulates the system's states and identifies any deadlock possibilities, providing details of any violations.

Model checkers generate a state model from code, which is an abstract state machine with various states. The model checker verifies the state model's compliance with specified behavioral properties, checking for errors like race conditions by exploring all possible states of the model.

3 different phases in model checking

1. **Modeling phase** involves entering the model description and desired properties in the same language as the system. This phase also performs simple tests, called sanity checks, before using model checkers. Sanity checks are quick and easy tests based on simple logic, like turning the system on and off, to detect simple errors before focusing on more complex properties.
2. The second phase in model checking is the **running phase**, where the model checker is executed to evaluate the model's conformity to the desired properties specified in the modeling phase.
3. The final phase of model checking is the **analysis phase**, where the satisfaction of all desired properties is checked, and any violations (**counterexamples**) are identified. The model checker provides descriptions of violations for analysis.

The model checker's information helps revise and fix software problems.

Running the model checker repeatedly until software is correct with respect to desired properties is good practice.

Model checking ensures software is well-designed, meets desired behavior and works as intended.

Acronyms

DC: Design Complexity

O-O: Object-Oriented

**AB|EN|DE|GE: Ab**straction, **En**capsulation, **De**composition & **Ge**neralization

**IN:** Inheritance

SD: Software Design

**BE|AT**: Behaviour & Attribute