Bryan Melanson

How to Not Fail
Software Design

While never going to class

Computer Engineering 2020 $\mathcal{BM}$

Contents

1 UML Class Diagrams

Unified Modeling Language or **UML** is a standardized modelling language used to aid in visualizing a system. It is designed to represent activities, architecture, and interactions of software entities.

1.1 Class Relationships in UML

Types of Relationships:

1. Association
2. Aggregation
3. Composition
4. Dependence
5. Generalization

1.1.1 Association (takes a relationship)

General purpose relationship. Pointers (C++) and ref variables (Java) An association is a general purpose relationship. Whenever two classes need to communicate with each other. They need some sort of association. Typically uses a reference variable or a pointer. In a UML diagram this is represented with a line. These relationships should be named When you can, add multiplicity to the UML diagram. One to One, One to many, Many to one etc. These are shown with * means multiplicity Or Navigability is important as it helps defines a classes role, and how it communicates within the program. Navigability is denoted with an arrow. No arrow means navigable in both directions.

1.1.2 Aggregation (Whole-Part Relationship)

Aggregation is a special type of association. If there's no association then there is no aggregation. It is used when a class is entirely purposed, or composed of, another class. Denoted with a diamond at the whole end

1.1.3 Composition (Whole-Part relationship with same lifetime)

Composition builds off aggregation, although its a little more hardcore. In Composition, when the parts are deleted, the whole is deleted as well. Looking back on our club/person example. If the club ends, the people dont end as well. Unless its Jones Town lol. An example of this is the following: When a polygon is destroyed, all the points that it is composed of are also destroyed. If a point is destroyed, then so is the polygon. Also note the multiplicity here, 1 polygon is composed of 3 or more points marked by 3*.

1.1.4 Summary of Class Relationships

Remember, UML diagrams show the possible relationships between classes. That doesn't mean that they all will. Summary diagram:

1.1.5 Recursive Associations (Boomerang Relationships)

An association that relates a class to itself Objects of the class may or may not be associated to themselves as well

1.1.6 Degrees of Belonging (Lifetimes)

Attribute: lifetime of attribute == lifetime of object that contains it Composition: lifetime of part == lifetime of whole Aggregation: Whole-part relationship Association: Relationship is not whole-part

1.1.7 Interfaces

An interface is a class with no attributes (variables) and no implementations for operations. It's simply a template for a class.
Remember: Abstract class can have some implementation Interfaces cannot have any You can implement multiple interfaces, you can only extend one class When something implements an interface, it promises to use all its specified methods and attributes
Here we see interface payment, with credit and cash being two classes that implement this interface. By doing so they are guaranteed to have a calcamount() operation. The sale class is associated with the payment interface, by doing so it knows that whatever kind of payment technique is passed to the sale it will be able to calculate the amount of the payment. Generalization == Inheritance

1.1.8 Abstract Operations

Abstract operations of a generalized class are represented in italics in UML Operation O is abstract in class C if it does not have an implementation in class C Implementation will be in specializations of C In VP, abstract classes have a little checkbox:

1.2 Dependence

Out of all the relationships, dependence is the weakest. If a class has any sort of mention of another class in its parameters, returns, variables, then it is dependent. Generally, dependence on a class is a bad thing as it makes a class harder to reuse, isolate debugging, and harder to understand. It is better to have dependence on an interface. Referring back to this example: The sale class has a dependency on the payment interface. For reasons explained earlier. Without it the sale class would have two dependencies, one to each kind of payment. Then there would be no guarantee that all the different kinds of payments would have a calcamount class. With this design the sale class doesn't give a flying fuck how many different kinds of payment classes there are. You could be paying in bitcoin, who cares! As long as it interfaces the payment class then we know it will have a calcamount operation and the sale class knows it can work with it.

1.3 UML Diagrams from Assignments / Google to look at

2 UML – Sequence and Communication Diagrams

3 Principles of Design

A design blows if it Is hard to change. This means it is Rigid Easy to break, making it fragile Hard to re-use, not modular, immobile Hard to do what its intend to do, constantly doing work arounds Viscous Needless Complexity Needless Repetition

3.1 Design Smells

Ex. Rigidity, fragility, immobility, viscosity, needless-¿complexity, repetition, opacity. Over time, software rots

3.2 Single Responsibility Principle (SRP)

A class should only have one responsibility A class should only have one reason to change Several responsibilities creates unnecessary couplings between those responsibilities

3.3 Open-Closed Principle (OCP)

Software entities (classes, functions, etc.) should be open for extension, but closed for modification To change behavior, add new code rather than changing existing code How? Abstraction.

3.4 Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types Violated when the reasonable expectations for a base class are not met for subclass. Can avoid violating with strong PRE/POST conditions, i.e, design by contract Meyers Rule: A routine redeclaration [i.e. an overridden method] may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger. LSP enables OCP.

3.5 Dependency-Inversion Principle (DIP)

High level modules should not depend on low level modules. Both depend on abstractions. Abstractions should not depend on details. Details depend on abstractions. You shouldn't depend on a concrete class. All relationships in a program should terminate on an abstract class or interface.

3.6 Interface Segregation Principle (ISP)

Interface that provides different groups of methods to different clients is bad SRP applied to interfaces. Point is to avoid fat interfaces that have little cohesion between methods

4 Design Patterns

Design pattern is a general solution to a commonly encountered problem in OO design.

4.1 Iterator

Provide a way to access elements of an aggregate object sequentially without exposing the underlying representation Can apply DIP to abstract out the type of iterator required for a certain data structure.

4.2 Strategy

Define a set of interchangeable algorithms. Ex: dumbAI/smartAI are different concrete strategies. They can change, but changes are insulated from the client code.

4.3 Factory

Create concrete objects through an abstract factory interface

4.4 Singleton

Singleton class is suitable when we need exactly one object, with global access. Ex: 1 OS has 1 file system. Ship has one captain. Sometimes referred to as an anti-pattern. Extendable.

4.5 Facade

With many small classes, may be difficult for clients to understand your design. Facade provides a simple interface to a complex system. (can be singleton)

4.6 Composite

Allows clients to treat individual objects and compositions of objects uniformly. I.e, you can call methods on classes, or collections of classes.

4.7 Adapter

Converts interface of a class into another form. Ex: Switch -¿ Switchable Interface(on,off) -¿ light(on,off) But, what if light is 3rd party and only has toggle method? Turn into: Switch -¿ Switchable Interface(on,off) -¿ light adapter(on,off)-¿light(toggle)

4.8 Observer

Sometimes, objects need to update when activity occurs in a subject object. Observing/listening objects subscribe to the subject Subject automatically notifies observer objects when something is changed Subjects extend Subject, observers implement Observer

4.9 Decorator

Add new behaviors to an object without inheritance Inheritance: new behaviors at compile time. Decorator: run time Look at note examples

4.10 Command

Treats requests as objects. Instead of function call, we create an object that provides an execute method and stores the parameters of the function call. Applications: Queue up commands, provide logging, transactions, unlimited undo/redo

5 Threads and Concurrency in Java

5.1 Concurrency

Multiple agents running at the same time and interacting Ex. Interacting processes running on different computers (running Internet Explorer and Chrome at the same time) Multithreading: multiple threads of control Concurrency can be intermachine, interprocess or multithreading Reasons for using: o Speed multiple threads on multiple processors (faster processing?) o Distribution may want different parts of a system located on different machines o Asynchrony it is easier to deal with multiple sources of events (multiple inputs) by having one thread dedicated to each stream of incoming or outgoing events (I/O ports or signals)

5.2 Threads

Each thread has its own: o Program counter o Registers o Local variables and stack All threads share the same heap (objects)

5.3 Multiple Processors

Time splicing implementation (for running threads concurrently): Single Processor the CPU is switched at unpredictable times Multiple Processor thread may occasionally migrate

5.4 Thread Objects in Java

- Remember: o Objects are shared by threads o In Java, access to an objects methods is uncontrolled Use the run() method Starting and new thread: o Calling t.start() starts and new thread, which executes t.run() Example output: When t.run() completes the thread stops, then the program exits

5.5 Race Conditions

A race condition is when a systems correctness depends on a certain order of events, but the order of those events is not sufficiently controlled by the design of the system Race conditions occurs when 2 agents have uncontrolled access to a shared resource Example in notes: trains crossing bridge with only one track Need control measures acquiring and relinquishing of tokens See example of a counter in the notes (Threads Part 1 Slide 21), to summarize: o Two threads created that share the same object Counter c o Printing the output shows that our results have been distorted o The increment operation counter.increment() reults in multiple bytecode instructions that get interleaved (mixed up) o ++count example: threads p and q race each other, the result was that an increment was lost See notes for second example

5.6 Synchonized Methods

Methods may be declared synchronized How does it work? For each object: o The object will be given a token called its lock o At each point in time, each lock is owned by one thread or no threads (no sharing!) o A thread has to wait until a lock is free o A thread will own a lock until it decides to relinquish it (temporary but protected ownership) Using the synchronized method x.m(): o A thread will invoke the synchronized method x.m(), and then: If it does not already the lock, it will wait until it is free Once the lock is acquired, the thread begins to execute the method o When a thread leaves an invocation of a synchronized method: If it leaving the synchronized

last invocation for the object itll give up the lock as it leaves (leaves keys behindlike a good tenant) See notes for examples and room metaphor

5.7 Design Rule: Shared Objects

For any object that might be used by more than one thread at the same time: o Declare all methods that access or mutate the data synchronized o ** Exempted: private methods, constructors See notes for AccountManager example

5.8 Waiting

Other than synchronized, we can also use waiting Waiting lets us make a thread wait until a specific condition has arisen

5.9 Type 1: Polling

Uses loops and .yield to keep thread from hogging CPU

5.10 Type 2: wait and notifyAll

Better to have a thread wait until notified about change in condition To wait, the object sends a wait message While waiting, the thread gives up the lock (ownership) To allow other threads to stop waiting, a thread sends a notifyAll message to the object

5.11 Mailbox Example

In this system, producers send a message to the mailbox. The mailbox then stores the message and acts as an access point for the receivers to come get the message. Although lets say the Mailbox is a server with a limited amount of storage (CAP). If the server receives a message when it is at memory capacity the message gets dropped and lost. To prevent this from happening the send method checks the capacity of the message before sending. If it is at capacity it waits. By waiting two dope things happen: 1. Messages wont be dropped and lost 2. The thread doesn't have to poll the mailbox taking up CPU and hogging the thread lock. It can release the lock, sit back and relaxing waiting for someone to tell it to wake up. When a recipient removes a message from the mailbox it makes room for new ones so it notifies all that there is room now. By calling notifyall any threads that

were stuck in a wait can now wake up and use the new room. If another thread beat it to the new open space, it will go back to waiting.

5.12 Deadlocks

While this is fine and dandy it proposes a problem, what if two threads are waiting on each other to do something? They will both be stuck in a wait state thus locking the program. Slides provide a straightforward example of this.

5.13 Summary

6 Testing

6.1 Levels of Testing

1. Unit Testing o Test an individual unit of software (methods or complete classes) 2. Integration Testing o Individual software components are combined and tested as a group 3. System Testing o The system is tested as a whole

6.2 Testing Methods

White Box Testing: o The tester has access to underlying implementation o The tester applies tests to satisfy some criteria Black Box Testing: o Tester has no access to underlying implementation o Focusses on testing the system as a whole to verify requirements have been met

6.3 Test Driven Development

Agile (software development methodology) recommends using a TDD TDD focusses on unit tests o Idea is to write unit tests prior to implementing the feature The test-driven development cycle is: 1. Add a new test 2. Run all tests The new test should fail because we havent implemented the feature yet! 3. Write some code that causes test to pass 4. Run all tests (again) 5. Refactor and re-run tests again Clean up code and apply design principles 6. Repeat 7.4 Advantages of Test Driven Development Encourages more testing, productivity, Makes developers think a bit more about their design Acts as an executable documentation for your code Forces developer to decouple components

6.4 Decoupling

Before you can write a unit test (a test apply to a single software unit, aka. A class) you will need to decouple it from other objects and classes Create fake interfaces in place of other classes (ex. ¡¡interface¿¿ Employee and ¡¡interface¿¿ CheckWriter, etc.) Provide mock implementations (in example, Mock Checkwriter, Mock Employee)

6.5 JUnit

What is JUnit? o Junit is a unit-testing framework for Java o Works well with TDD What is unit testing? o Looking for errors in a subsystem (class or object) in isolation o Ex. Given a class Foo, create class FooTest to test it, looking for particular results to pass / fail

6.6 Assertion and Running a Test

Junit works by providing assert commands to help us write tests Placing assertion calls in the test methods allow us to check for things that we expect to be true else the test will fail In your test class use a method with an @test flag for the Junit test case Assertion Methods

6.7 Junit Summary

Tests need failure atomicity (ability to know exactly what failed) Test for expected errors / exceptions Choose representative test cases from equivalent input classes Avoid complex logic in test methods Use helpers, like @Before

7 MVC and Graphics

7.1 Model View Controller

Model: Holds state information View: presents a visualization of the models state Controller: Interprets UI events (mouse clicks, keyboard input, etc.), and turns UI events into changes to the model (and sometimes view state) MVC encourages the separation of: presentation from representation view from control

7.2 Flow of Information (Idealistic)

Underlying Relationships Often the flow is more complicated because: 1. The underlying GUI system associates events with view objects. (E.g. in AWT/Swing. Events are routed through the GUI component the user directs them at.) 2. The controller may need to know the models state 3. Some events affect only the view and so should not go through the model. (E.g. Scrolling, cursor position, selection.)

7.3 Flow of Information (More Realistic)

Strategy (View using the Controller) o The view uses the controller as a strategy to help it deal with input events o For reusability they usually know each other only via interfaces Observer (Controller and View observing Model) o Controller observes Model so it is aware of relevant changes to state o View also observes Model so that is aware Composite and Faade o Composite can be used by View, Model or Controller o Faade can be used by Model only State Pattern (Used by Controllers and Models) o OO Implementation of state machine o Each state is a different class o All have same interface (strategy pattern) View Controller Model Strategy Observes Controller Recipient Observer Observes Model Observes Model Recipient Composite Can be used Can be used Can be used Facade Can be used State Pattern Can be used Can be used

7.4 Advantages of these MVC Design Patterns

Clean separation of View from Model Clean separation of View from Control o View is platform dependent o By separating controller you can re-use it Clean synchronization (observer pattern keeps everything in sync)

7.5 Case Study: Race Rat Game

See notes.

8 Graphics with libGDX

To be added