50.003 Elements of Software Construction
Finals Revision

Tey Siew Wen

12 Oct 2019

Preface

As there are a lot of contents to be revised for finals, I've split up the original document into individual sections instead.

Credits

- UMLDiagrams.org - UML Use Case Diagram
- Journal of Systems & Software - UML Misuse Case Diagram
- LucidCharts - UML Class Diagrams
- LucidCharts - UML Sequence Diagram
- Guru99 - Equivalence Class Partitioning & Boundary Value Testing
- Tester Stories - Control Flow Graphs
- StackOverflow - OBJ04-J's explanation on the `final` term
- Jenkov Java Concurrency Tutorials for some general concepts
- HowToDoInJava for CountDownLatchExample.
- JavaEUSpecialists for CyclicBarrierExample & PhaserExample *(i renamed and changed some stuff for easier readibility)*
- Netjs for Phaser Explanation
- 50.003 Elements of Software Construction Slides - Syllabus and some content
- Prof Sudipta - Comments on Finals
- Lyqht - Compilation

not testing for 2019 ESC Finals:

- Software engineering process e.g. water fall, agile, time-based
- testing tools eg. junit, selenium, evosuite, klee

Contents

1 Concurrency

1.1 Main Issues of Concurrency

3 Main Issues:

1. **Visibility**

   - Visibility is solved if all cores are sharing all the memories. Thus, updates to a variable will immediately be visible in the shared memory and also to an arbitrary thread running on an arbitrary core.

2. **Atomicity**

   - Making statements atomic alone does not solve the issue of atomicity. Often according to your requirement, you need to make multiple statements (a code block) atomic.
   - e.g. consider a stack class that is trying to pop an item. You might need to check the empty stack and the pop together atomically. If only single statements are atomic, this only guarantees that the check and the updates are atomic in isolation, but not together.
   - To solve such problem, we will still need locks.

3. **Execution Ordering**

   - Unless the hardware implements a completely deterministic scheduler known to the programmer, the execution order problem will persist. In practice, it is almost impossible to implement a completely deterministic scheduler.
   - But, using locks can help to restrict the number of possible execution orders.

1.2 Basics of Concurrency - Multi-threading

1.2.1 Starting up a thread on Java

1. Make an object class that extends Thread.

2. For Thread objects, it is mandatory to implement the `run()` method. In this method, you write the code for the task that you want to thread to be doing e.g. arithmetic operations, insertion into a list etc.

3. In a main program, create an instance of the object Thread and start it. This makes the object-Thread start running its `run()` method.

4. The main program should only resume when the threads finish their work. To make the main program wait, use `join()`.

```java
public objectThread extends Thread() {
    public void run() {
        System.out.println("Thread is running.")
    }
}

public void main String[] args {
    objectThread thread1 = new objectThread();
    objectThread thread2 = new objectThread();

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();
}
```

1.2.2 Stopping a thread

There are times when we want to stop a thread under certain conditions or just for fun. Unfortunately, we cannot destroy a thread, but we can interrupt its work using `interrupt()`.

- We can use a try/catch block to handle the `InterruptedException e`.
- We can also check if the thread is interrupted with `Thread.interrupted()`, which returns a boolean.

P.S. Why we cannot destroy threads is because of the way that the thread library scheduler works. The thread still has to exist in the thread table even if it is no longer running so that it doesn't screw up the scheduling. For more details, please ask your friendly CSE / ESC / Google profs. There are methods such as `destroy()`, `stop()` for Java threads but they are deprecated.

TLDR: Only do interruption to stop a thread.

1.2.3 Runnable Interface

- Alternative for implementing & starting a thread
- Allows the object to inherit from other classes as well.
- Runnable allows better separation of the actual computation and thread control.

```java
class RunnableExample {
    public void run() {
        System.out.println("Thread is running.")
    }
}
new objectThread(RunnableExample).start();
```

OR

```java
new objectThread(new Runnable() {
    public void run() {
        System.out.println("Thread is running.")
    }

}).start();
```

1.3 Basic Thread Safety

> "A class is thread-safe if no set of operations performs sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an **invalid** state."

What constistutes a valid state depends on the specs such as:

- pre-conditions/post-conditions
- assertions

How to build thread-safe classes?

- From scratch
- By extending the class
- Through client-side locking
- Through composition

Steps to build thread-safe classes from scratch

1. Identifying states A
    - An object's state includes all of its mutable variables.
    - If the object has fields that are references to other objects, its state will encompass fields from those as well.
2. Identifying Requirements
    - Constraint on valid values or state transitions for state variables can create atomicity & encapsulation requirements.
3. Designing Policy
    - Update related state variables in a single atomic operation
    - For each mutable variable that may be accessed by more than one thread, all assesses to that variable must be performed with the same lock held.
    - Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.
    - For every invariant that involves more than one variable, all the variables involved in that invariant must be guarded by the same lock.
4. Implementing Policy
    - Make sure every access of any variable is guarded by the lock according to the policy.
    - Make sure access of the related variables in the same method is in synchronized block.
    - Add waiting (and notify) to handle pre-conditions.
    - Try to use `final` for variables as much as you can.

1.3.1 Locking

Locking is not just about mutual exclusion; it is also about memory visibility. To ensure that all threads see the most up-to-date values of shared mutable variables, the reading and writing threads must synchronize on a common lock.

Lock Contention

- Two factors influence the likelihood of contention for a lock
  - i. How often that lock is requested
  - ii. How long it is held once acquired
- There are three ways to reduce lock contention
  - i. Reduce the duration for which locks are held
  - ii. Reduce the frequency with which locks are requested
  - iii. Replace exclusive locks with coordination mechanisms that permit greater concurrency

1.4 Requirements of Multi-threaded programs

1. No Race Condition
2. No Visibility Issue
3. No Execution Ordering Problem
4. No Deadlocks
5. Efficiency (most basic!)

1.4.1 The Race Condition

Imagine playing Overcooked, where you and your friends would often scream and shout at each other over the wrong order of ingredients and procedure of matters. There are also times where you and your friends both need to access the same ingredient or cooking pot at the same time. None of you communicated about this issue, so the item of contention just becomes first come first serve. This is exactly the Race Condition when there is no global order of execution of threads, and the threads each hold onto a resource that other threads need.

1.4.2 Visibility

Implement variables as `volatile`, so that any update to the variable will be updated in the main memory and be seen by all threads. Also, the use of `volatile` actually does not only affect the `volatile` variable only, it affects the other variables of the same object as well under certain conditions.

1.4.3 Deadlocks

- A program that never acquires more than one lock at a time cannot experience lock-ordering deadlocks.
- Strive to use open calls (calling a method with no locks held) throughout your program.
- A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order.

Non-blocking algorithms

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread.

Non-blocking algorithms are immune to deadlock (though, in unlikely scenarios, may exhibit livelock or starvation).

1.5 Java Synchronizers

1.5.1 CountdownLatch

The CountDownLatch class allows one or more threads to wait until a set of operations in other threads complete. CountDownLatch works by having a counter initialized with number of threads, which is decremented each time a thread complete its execution. When count reaches to zero, it means all threads have completed their execution, and thread waiting on latch resume the execution.

1. `CountDownLatch latch = new CountDownLatch(numberOfThreads);`
2. To inform the main thread that one of the service threads has completed its task, we decrement the counter with `latch.countDown();`
3. The main thread can resume its work with `latch.await();`

A more concrete example of a program that relies on service threads and a main thread needs to wait for several threads to finish their tasks:

```java
public static void main (String[] args) {
    CountDown Latch latch = new CountDownLatch(numberOfTasks);

    // initiatite the tasks
    services = new ArrayList<ToDoTask>();
    for (int i = 0; i < numberOfTasks; i++) {
        services.add(new ServiceWorker(i, latch));
    }

    // initiate the thread pool to take on the tasks
    Executor executor = Executors.newFixedThreadPool(services.size());
    for(ToDoTask w : services) {
        executor.execute(w);
    }

    // Check
    latch.await();
    for(ToDoTask w: services) {
        if(!w.isServiceUp()) {
            return false;
        }
    }

    return true;

    }
}
```

Here's the methods for ToDoTask object class.

```
public ToDoTask(int i, CountDownLatch latch)
    {
        super();
        this._latch = latch;
        this._index = i;
        this._serviceUp = false;
    }

    @Override
    public void run() {
        try {
            doSomeWork();
            _serviceUp = true;
        } catch (Throwable t) {
            t.printStackTrace(System.err);
            _serviceUp = false;
        } finally {
            if(_latch != null) {
                _latch.countDown(); // decrement latch counter.
            }
        }
}

public doSomeWork() {
    // some working code
}
```

What if you are not certain of how many batches of tasks you actually need to complete? You would need to create a new CountDownLatch for every batch of tasks that you receive, and in this code, this is not accounted for. So, we turn to CyclicBarrier for resolving this issue.

1.5.2 CyclicBarrier

For different batches of the same number of tasks, the CyclicBarrier can be reused indefinitely.

Assuming the following variables & the method `doTask` are declared as such for the CyclicBarrier main program:

```java
import java.util.concurrent.*;

protected final static int TASKS_PER_BATCH = 3;
protected final static int BATCHES = 5;

protected final void doTask(int batch) {
    System.out.printf("Task start %d%n", batch);
    int ms = ThreadLocalRandom.current().nextInt(500, 3000);
    try {
        Thread.sleep(ms);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    System.out.printf("Task in batch %d took %dms%n", batch, ms);
    }
}
```

CyclicBarrierExample program code:

```java
public class CyclicBarrierExample {
    public static void main(String[] args) {
        CyclicBarrierExample eg = new CyclicBarrierExample();
        ExecutorService pool = Executors.newFixedThreadPool(TASKS_PER_BATCH);
        CyclicBarrier barrier = new CyclicBarrier(TASKS_PER_BATCH);

        for (int batch = 0; batch < BATCHES; batch++) {
            for (int i = 0; i < TASKS_PER_BATCH; i++) {
                int batchNumber = batch + 1;
                pool.submit(() -> eg.task(barrier, batchNumber));
            }
        }
        pool.shutdown();
    }

public void task(CyclicBarrier barrier, int batch) {
    boolean interrupted = Thread.interrupted();
        while (true) {
            try {
                barrier.await(); // wait for barrier to be lifted.
                break;
            } catch (InterruptedException e) {
                interrupted = true;
            } catch (BrokenBarrierException e) {
                throw new AssertionError(e);
            }
        }
    if (interrupted) Thread.currentThread().interrupt();
```

```
        doTask(batch);
    }
}
```

1.5.3 Phaser

Essentially, Phaser = CountdownLatch + CyclicBarrier.

What sets Phaser apart is it is **reusable** (like CyclicBarrier) and **more flexible in usage**. In both CountDownLatch and CyclicBarrier number of parties (thread) that are registered for waiting can't change where as in Phaser that number can vary.

Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or by specifying initial number of parties in the constructor). Tasks may also be optionally deregistered upon any arrival (using `arriveAndDeregister()`).

We can reuse the phaser for the batches, like with the CyclicBarrier. The Phaser also knows which phase it is in, thus we do not need to pass along the batch number.

```java
public class PhaserExample extends LockStepExample {
    public static void main(String[] args) {
        PhaserExample eg = new PhaserExample();
        ExecutorService pool = newFixedThreadPool(TASKS_PER_BATCH);
        Phaser phaser = new Phaser(TASKS_PER_BATCH);

        for (int batch = 0; batch < BATCHES; batch++) {
            for (int i = 0; i < TASKS_PER_BATCH; i++) {
                int batchNumber = batch + 1;
                pool.submit(() -> eg.task(phaser));
            }
        }
        pool.shutdown();
    }
```

The task() method is also just reduced to a one-liner: `phaser.arriveAndAwaitAdvance()`.

```java
public void task(Phaser phaser) {
    phaser.arriveAndAwaitAdvance();
    doTask(phaser.getPhase());
}
```

Advantage: if our Phaser blocks a thread in the common fork-join pool, another will be created to keep the parallelism at the desired level.

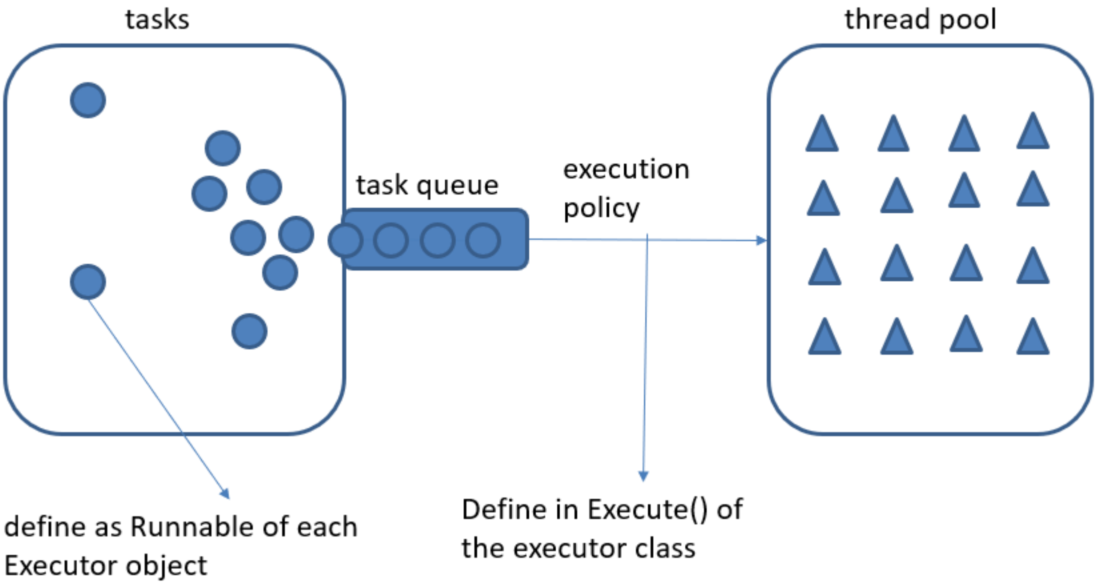
1.5.4 Summary

CountDownLatch	Cyclic Barrier	Phaser
• Created with a fixed number of threads	• Created with a fixed number of threads	• Number of threads need not be known at Phaser creation time. They can be added dynamically.
• Cannot be reset	• Can be reset	• Can be reset and hence is, reusable.
• Allow threads to wait with `await()` or continue with its execution `countdown()`	• Does not provide a method for the threads to advance. The threads have to wait till all the threads arrive	• Allow threads to wait with `arriveAndAwaitAdvance()` or continue with its execution `arrive()`

1.6 Performance

1.6.1 Thread Pools with the Executor

Executor provides a standard means of decoupling task submission from task execution. The `Runnable` is the task itself. The method `execute` defines how it is executed.



The queue may grow unbounded! When a bounded task queue fills up, the saturation policy comes into play. default: throw `RejectedExecutionException`

Advantages

- Reusing an existing thread; reduce thread creation and teardown costs.
- No latency associated with thread creation; improves responsiveness.
- By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or crashes due to competition among threads for resources

Disadvantages

- Context switching: requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread
  - CPU time spent on JVM/OS
  - Cache misses
  - Costs about 5,000 to 10,000 clock cycles
- Memory synchronization:
  - Memory barriers inhibit compiler optimization

Execution Policy

- In what thread will tasks be executed?
- In what order should tasks be executed (FIFO)?
- How many tasks may execute concurrently?
- How many tasks may be queued pending execution?
- If a task has to be rejected because the system is overloaded, which task should be selected and how the application be notified?
- What actions should be taken before or after executing a task?

Task Implementation
Thread pools work best when tasks are homogeneous and independent.

- Dependency between tasks in the pool creates constraints on the execution policy which might result in problems (deadlock, liveness hazard, etc.)
- Long-running tasks may impair the responsiveness of the service managed by the Executor.
- Reusing threads create channels for communication between tasks – don't use them.

Task should not be too big or small:

- Too small: the ratio of useful work vs overhead becomes small
- Too big: Number of tasks available at a time is upper bound on achievable speedup

Implementations

1. Creating the Thread Pool
   - `newFixedThreadPool`
     - Fixed-size thread pool; creates threads as tasks are submitted, up to the maximum pool size and then attempts to keep the pool size constant
   - `newCachedThreadPool`
     - Boundless, but the pool shrinks and grows when demand dictates so
   - `newSingleThreadExecutor`

    - A single worker thread to process tasks, sequentially according to the order imposed by the task queue

- `newScheduledThreadPool`
    - A fixed-size thread pool that supports delayed and periodic task execution.

2. Terminate the executor's service:

- `shutdown()`
    - will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run
- `shutdownNow()`
    - will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads.
    - Note that if your tasks ignore the interruption, `shutdownNow()` will behave exactly the same way as `shutdown()`.

1.6.2 Optimal CPU Utilization

Given these definitions:

- N = number of CPUs
- U = target CPU utilization
- $\frac{W}{C}$ = ratio of wait time to compute time

The optimal pool size is:

$$M = N * U * \left(1 + \frac{W}{C}\right)$$

The number of CPUs can be obtained by: `Runtime.getRuntime().availableProcessors()`.

1.6.3 Other Factors

Other resources that can contribute to sizing constraints are memory, file handles, socket handles, database connections, etc.

- Add up how much of those resources each task requires and divide that into the total quantity available

Alternatively, the size of the thread pool can be tuned by running the application using different pool sizes and observing the level of CPU and other resource utilization.

1.7 Parallelization Patterns

- Loop Parallelism: Given a loop, each thread/process execute part of the loop.
- Master/Worker: A master thread/process divides a problem into several sub-problems and dispatches them to several worker processes.
- Fork/Join

    - Tasks are created dynamically
    - Tasks can create more tasks
        - Manages tasks according to their relationship
        - Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation

1.8 Concurrency-related Libraries

1.8.1 Older libraries (JDK 2)

- Usually we don't use such libraries anymore, but for legacy sake, we need to learn about how they work
- Usually involves locking a collection and are built with synchronized methods, hence making it very thread-safe, but this results in compromise of performance.

  e.g. Vector is considered a "legacy" collection class. "Modern" collection classes use Iterator.

Problems with locking a collection

- There is usually a hidden Iterator, so you can't modify/override the methods. And if you modify the collection during the iteration, a `ConcurrentModificationException` will be thrown. This is known as ***fail-fast***.
- If the collection is large or the task performed is lengthy, other threads could wait a long time.
- It increases the risk of problems like deadlock.
- The longer a lock is held, the more likely it is to be contended.

1.8.2 Slightly More Recent Libraries (JDK 5)

Synchronized collections in jdk2 are replaced with Concurrent collections in this version of jdk. This offers dramatic scalability improvement with little risk.

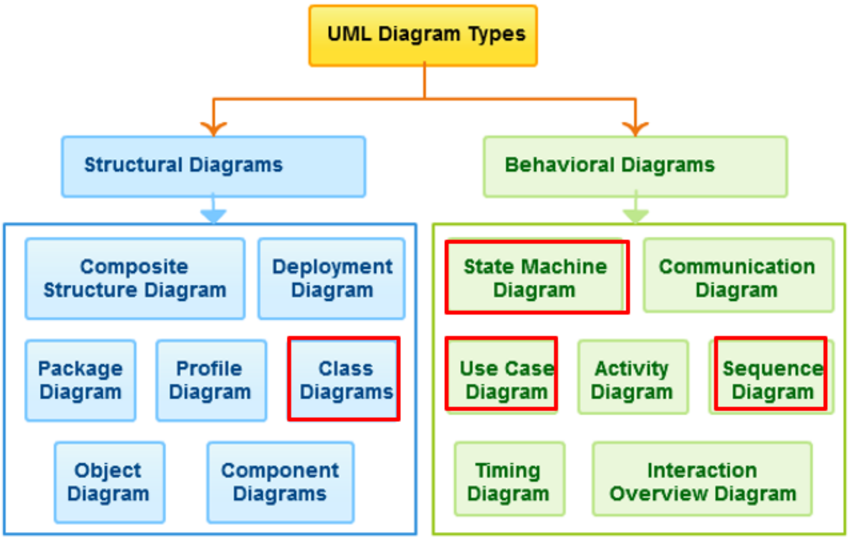
ConcurrentHashMap

- It is a HashMap designed for concurrency.
- It uses a finer-grained locking mechanism called lock striping.
- Uses an array of 16 locks. Each lock guards 1/16 of the hash buckets.
- Bucket N is guarded by lock N mod 16.
- The iterators returned by ConcurrentHashMap are weakly consistent (i.e., it is OK to modify the collection while iterating through it) instead of fail-fast.
- It does not support client-based locking.

CopyOnWriteArrayList

- It is a concurrent replacement for a synchronized list that offers better concurrency in some common situations.
- A new copy of the collection is created and published every time it is modified.
- All write operations are protected by the same lock and read operations are not protected.
- More efficient than alternatives when traversal operations vastly outnumber mutations.
- Useful when you don't want to synchronize traversals (the process of visiting each node in a data structure, exactly once.), but need to prevent interference among concurrent threads.

2 UML



- You will be given a requirement, then you have to model each of the diagrams below.
- It is important that you finish all the UML exercises for the telephone network. These were not part of the PSET. Thus, I am assuming that many did not do it. This is probably the time to do. Do not look at the answers, try to do it yourself. As per as UML is concerned, there are way too many correct answers. While grading, we aim to find things that are WRONG, rather than matching through a pre-computed correct answer.
- Ones highlighted in red are those that are taught in class.

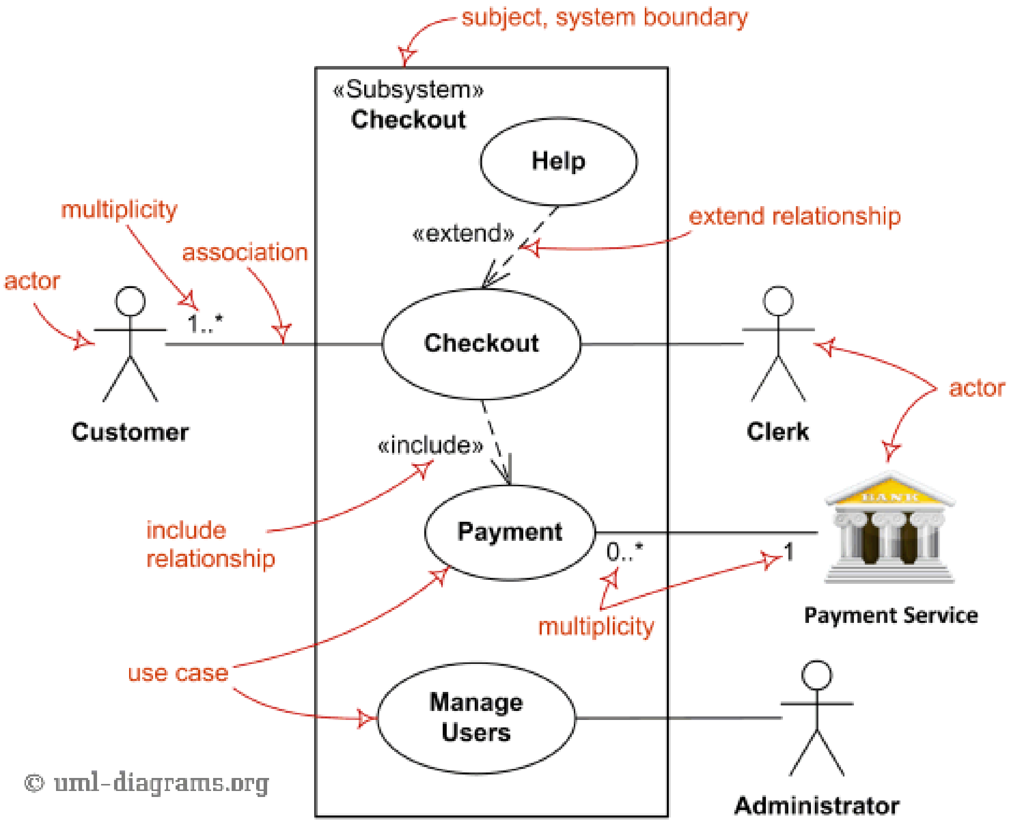
2.1 Use Case Diagram

A use case shows a set of use cases and actors and their relationships and represent system functionality, the requirements of the system from the user's respective.

- Use Case
  - ○ A description of a set of sequences of actions, including variants, that system performs that yields an observable value to an actor.
  - ○ Should ideally begin with a verb.

- Think of the end goal of a user. They don't want to ”login” or ”sign up.” That's not a use case. The use case is more like ”make a purchase.”

- Actors
  - People or systems that provide or receive information from the system; they are among the stakeholders of a system, which could be human beings, other systems, timers and clocks or hardware devices.
  - Primary Actors: stimulate the system and are the initiators of events (active).
  - Secondary Actors: only receive stimuli from the system (passive).



example: of use case diagram

The customer makes many checkouts and for any payment he makes, there is only one payment service in charge of it. While checking out, he might require help and by checking out, he would need to make a payment.

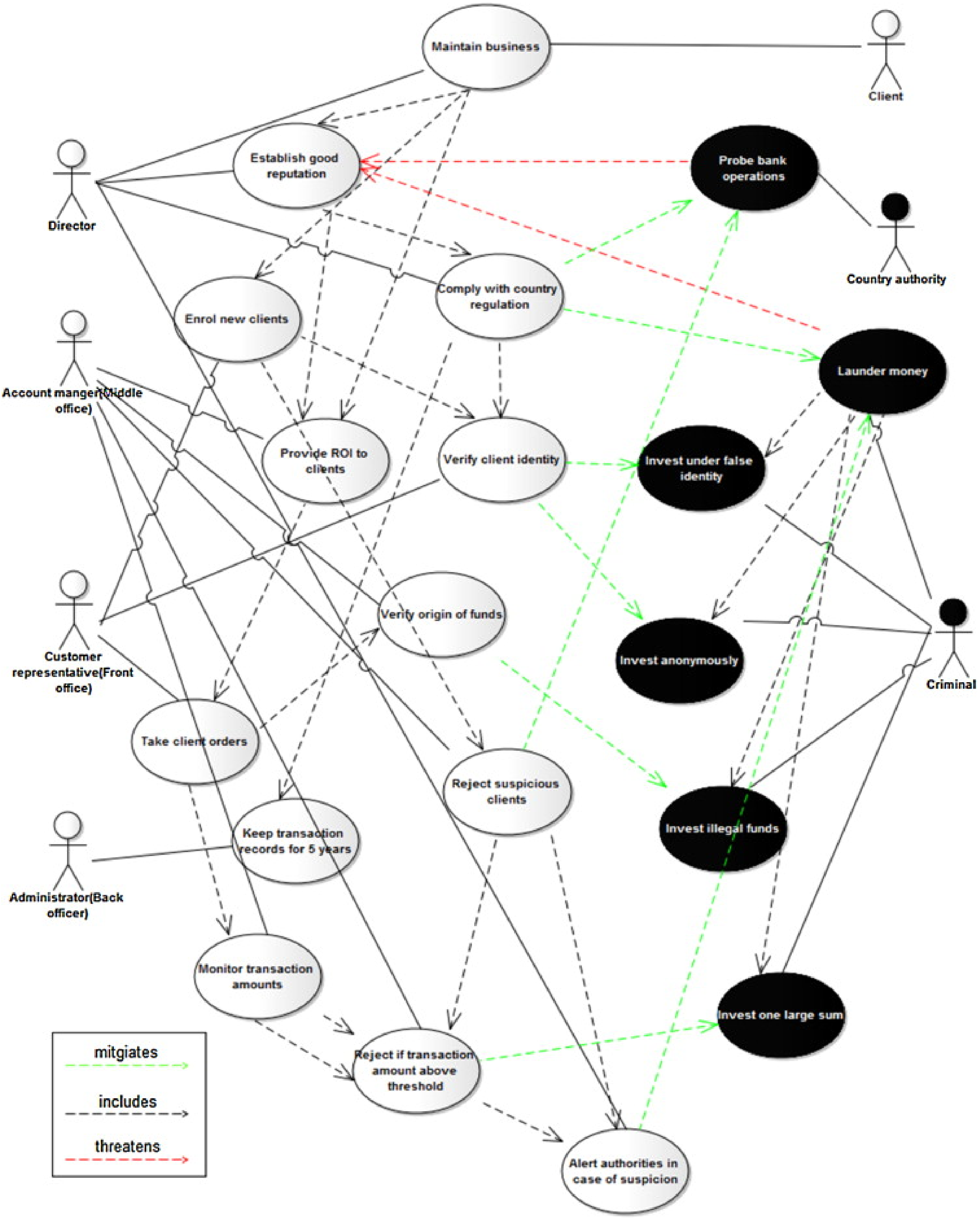
!! Note: Usually multiplicity is not required for use case diagrams, but mandatory in class diagrams.

2.1.1 Limitations

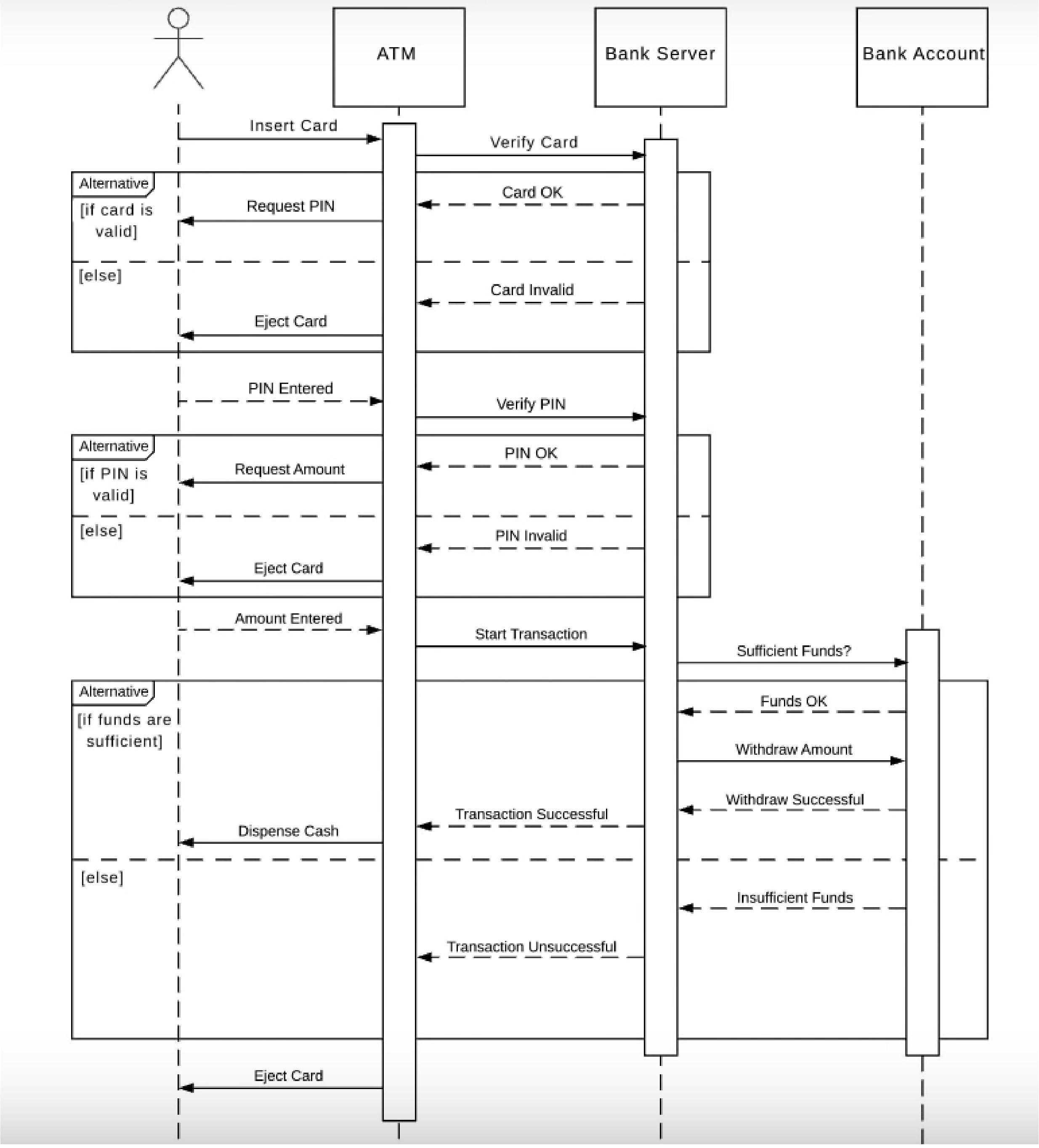
- Use cases only capture how a system is used
- Use cases do not consider adversarial scenarios
  - Abusing system functionality
  - Hacking and malfunctioning
  - In general, any non-functional requirement, especially security

- Clients and customers are unlikely to provide misuse cases, the onus is on the software engineers. Hence the need for misuse case diagrams arises.

example: misuse diagram


Maintain business
Client
Director
Establish good reputation
Probe bank operations
Country authority
Enrol new clients
Comply with country regulation
Account manger(Middle office)
Launder money
Provide ROI to clients
Verify client identity
Invest under false identity
Customer representative(Front office)
Verify origin of funds
Invest anonymously
Criminal
Take client orders
Reject suspicious clients
Invest illegal funds
Keep transaction records for 5 years
Administrator(Back officer)
Monitor transaction amounts
Invest one large sum
Reject if transaction amount above threshold
Alert authorities in case of suspicion
mitgiates
includes
threatens

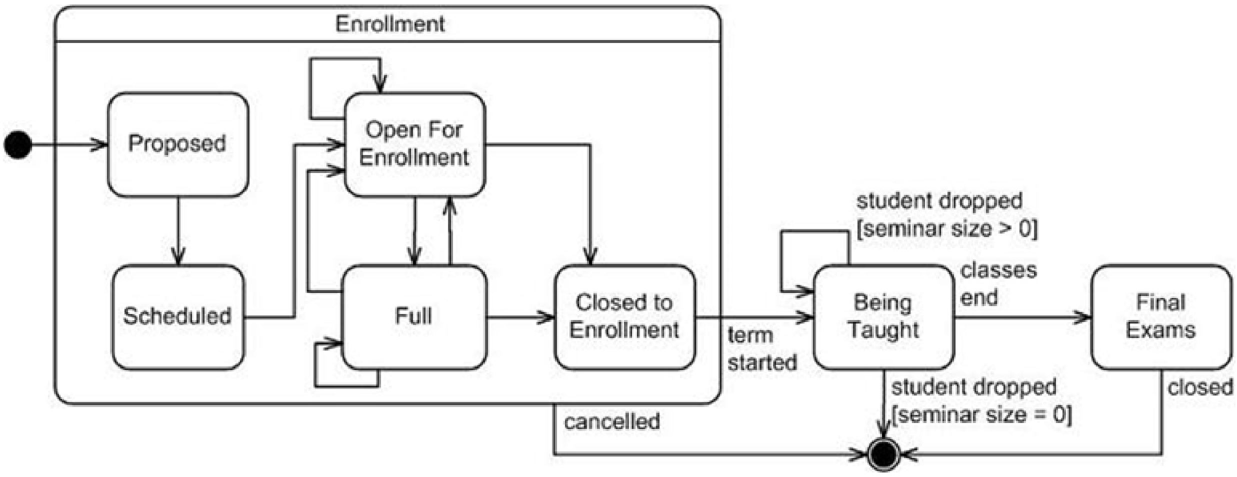
2.2 Sequence Diagram



2.3 State Diagram

If I ask to draw state diagram, I will not tell you the number of states or the names of the states. Thus, you need to pay attention on how to figure out states in a state machine from arbitrary requirement. - Sudipta

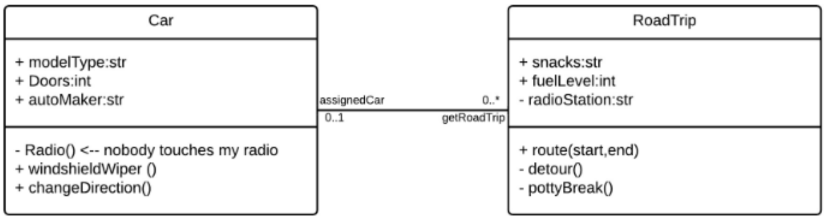
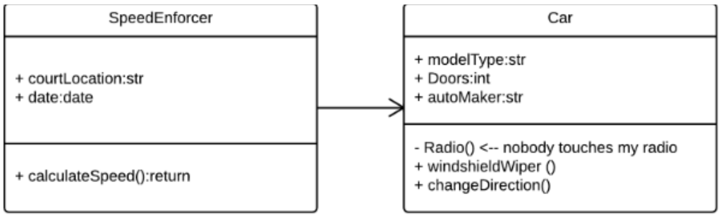
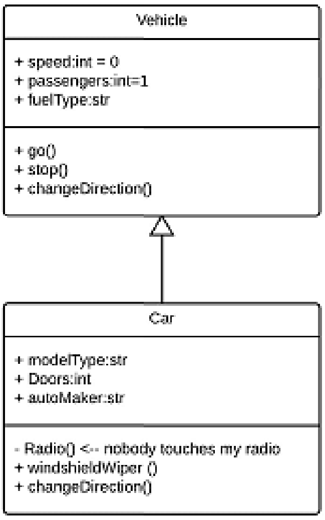
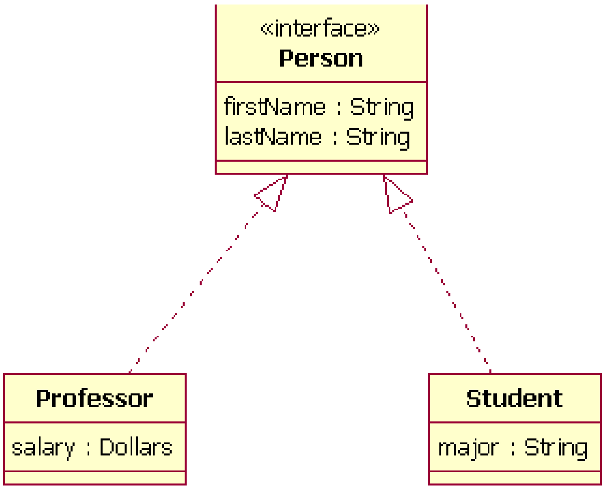
example: state diagram



2.4 Class Diagram

Understand the concepts of multiplicity and association clearly. These two are the most complex concepts as per as class diagram is concerned. - Sudipta

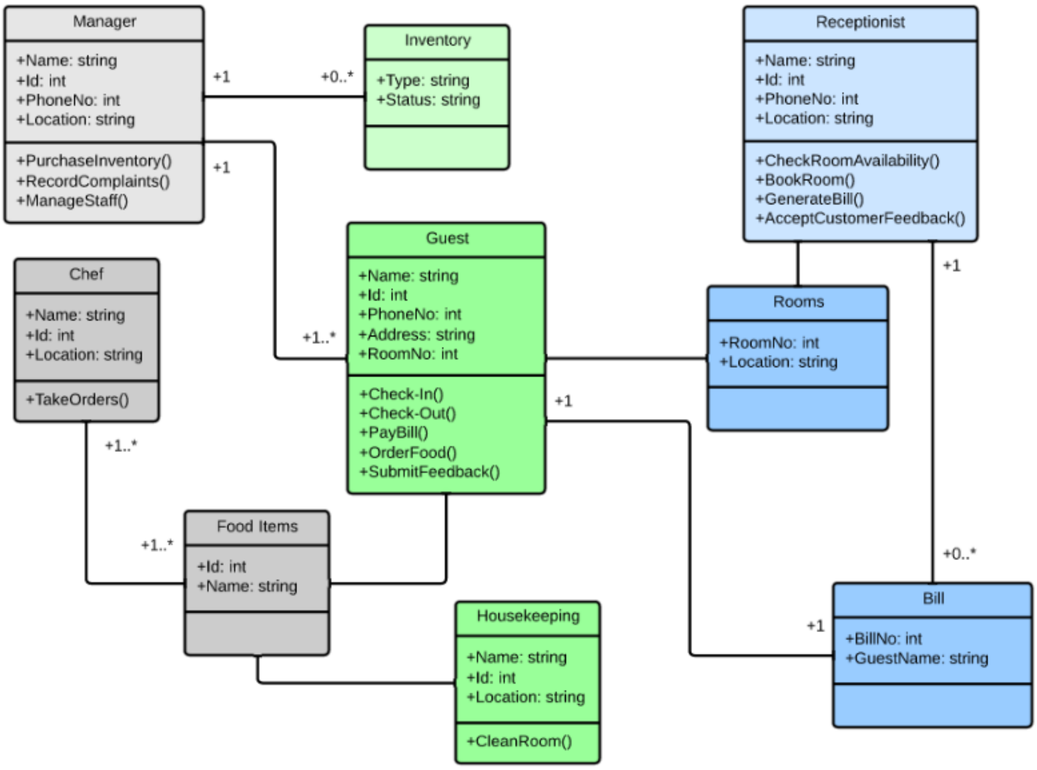
2.4.1 Association

Type of Association	*example:*
Bidirectional	 Car + modelType:str + Doors:int + autoMaker:str - Radio() <-- nobody touches my radio + windshieldWiper () + changeDirection() assignedCar 0..1 0..* getRoadTrip RoadTrip + snacks:str + fuelLevel:int - radioStation:str + route(start,end) - detour() - pottyBreak()
Unidirectional	 SpeedEnforcer + courtLocation:str + date:date + calculateSpeed():return Car + modelType:str + Doors:int + autoMaker:str - Radio() <-- nobody touches my radio + windshieldWiper () + changeDirection()
Inheritance	 Vehicle + speed:int = 0 + passengers:int=1 + fuelType:str + go() + stop() + changeDirection() Car + modelType:str + Doors:int + autoMaker:str - Radio() <-- nobody touches my radio + windshieldWiper () + changeDirection()
Implement Interface	 «interface» Person firstName : String lastName : String Professor salary : Dollars Student major : String

2.4.2 Multiplicity

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Three only
0..5	Zero to Five
5..15	Five to Fifteen

example: extensive multiplicity



Member Access Modifier Symbols:

- Public (+)
- Private (-)
- Protected (#)
- Package (~)
- Derived (/)
- Static (underlined)

3 Refactoring

3.1 Comments

- Used to document
  - Application programmer interfaces (APIs)
  - Choices of data structure and algorithms
- Things that can be included:
  - General description of what it does ***ONLY IF*** *your code cannot be self-explanatory.* e.g. your variables and method names by right should already speak for themselves what they do. Unless coz of legacy code that you can't override and have to use them.
  - Precondition (@param), Postcondition (output)

```java
// precondition: true
// postcondition: if the stack is empty, throws an exception;
// otherwise, remove the top element in the stack and leave the rest
public synchronized E pop() {
E obj;
int len = size();
obj = peek();
removeElementAt(len - 1);
return obj;
}
```

- Don't overcomment & don't explain how the code works e.g.:
  - Talk about how potential changes to be made in a different method

3.2 Code Smells

Its presence is *not conclusively* a sign of bad code, and depends on application, programming language etc. They are just signs that the code might be rottening and likely to attract maggots (bugs) in near future.

Do note that Removing one code smell may introduce other code smells.

3.2.1 Types of Code Smells

- **Repeated Code**
  - the method should have a single, clear objective.
  - Avoid blackhole classes: classes that start off small but are eventually stacked with too much responsibilities.
- **Long Message Chains:** e.g. A.B().C().D().E()
- **Speculative Generality:** Implement methods/interfaces in advance despite the lack of need for them now
- **Refused Bequest:** Bad Inheritance, where unnecessary methods are inherited.

- **Shotgun Surgery:** Adding a simple feature may require change all over the code
- **Feature Envy**
  - If one class (A) is interacting with another class (B) so frequently to the point that A is like doing the job of B & A is more interested in the job of B than its own.
  - Either B should take care of its own job or just join the classes together.
- **Data Class:** classes that just hold data without much responsibility
  - removing this might lead to data clumps and vice versa.
- **Data Clump:** (Opposite of Data Classes)
  - Often due to poor program structure or "copypasta programming".
  - To check whether or not some data is a data clump, just delete one of the data values and see whether the other values still make sense. If this isn't the case, this is a good sign that this group of variables should be combined into an object.
    - If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields (this will increase dependency between the two classes, might be good/bad depending on the situation)
    - Improves understanding and organization of code. Operations on particular data are now gathered in a single place, instead of haphazardly throughout the code.
    - Reduces code size.

example: of Data Clump class

```
public class DataClumpSmell {
    public DataClumpSmell() {
    }

    public void doSomething1(int x, int y, int z) {
    }

    public void doSomething2(int x, int y, int z) {
    }

    public void doSomething3(int x, int y, int z) {
    }

    // do other things without x, y and z
}
```

3.3 Secure Coding Standard

A set of rules which are meant to provide normative requirements for code. Each rule is associated with a metrics for severity (low, medium, and high), likelihood (unlikely, probably, and likely) and remediation cost (high, medium, and low).

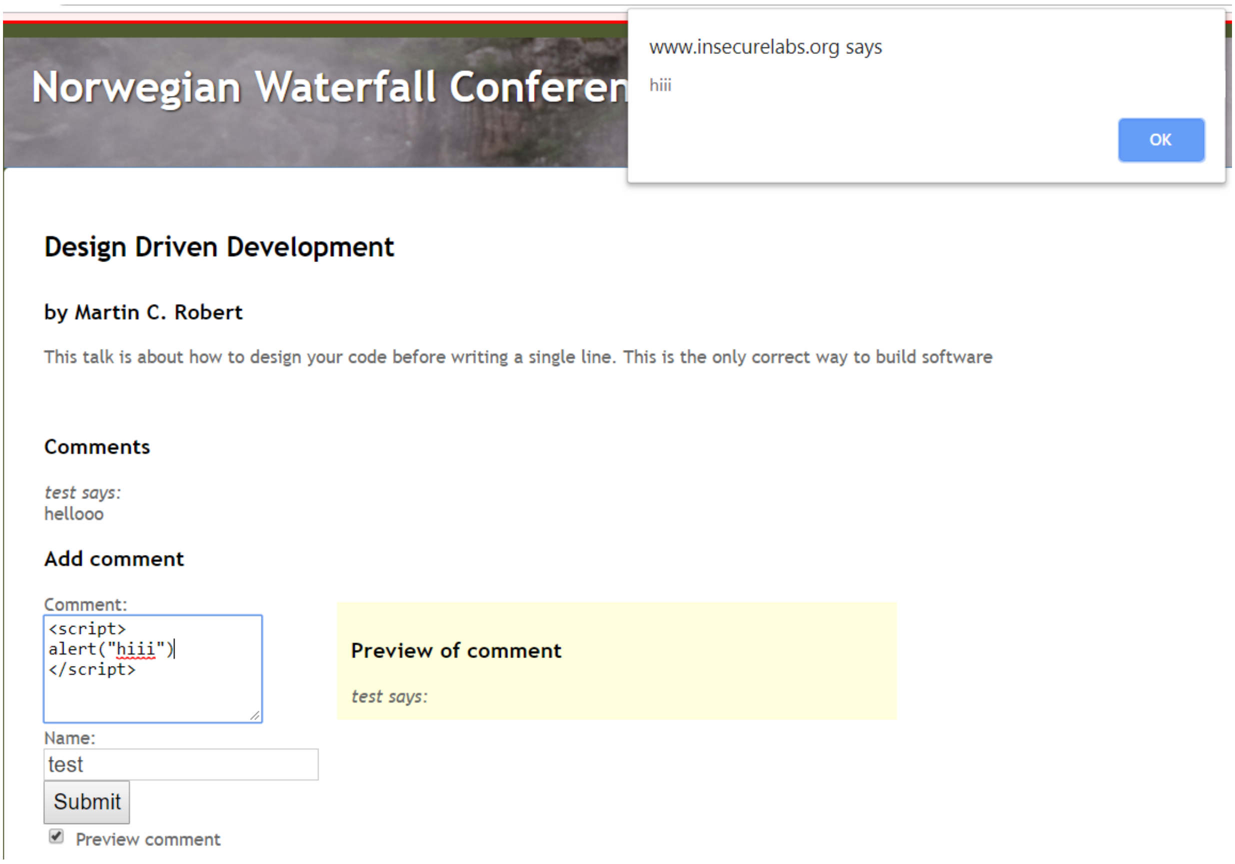
Conformance to the rule can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Types of Misuse we are taught are: XSS, XML, SQL injection.

3.3.1 XSS Injection

Also known as Cross Site Scripting. Given a website serves HTML pages to users who request them, the attacker is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website. Usually he does so by adding a `<script> ... </script>` in the html. The victim is a normal user of the website who requests pages from it using his browser.

example: of XSS Attack



www.insecurelabs.org says

hiii

Norwegian Waterfall Conferen

Design Driven Development

by Martin C. Robert

This talk is about how to design your code before writing a single line. This is the only correct way to build software

Comments

test says:
hellooo

Add comment

Comment:

```
<script>
alert("hiii")
</script>
```

Name:
test

Preview of comment

test says:

e.g. In this text area input element, usually visitors would input a harmless string. However, attackers can write direct code in it to carry out script attacks. This is possible if the code does not carry out sanitization / validation of input data.

try out XSS injection attacks at this site: http://www.insecurelabs.org/

3.4 Prevention

One strategy for avoiding XSS may include forbidding `<script>` tags in inputs. But, it is insufficient for complete input validation and sanitization, as the "sanitized" input may go through normalization afterwards.

Non-compliant code ***example***:

```java
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(in);
if (matcher.find()) {
    // Found black listed tag
    throw new IllegalStateException();
} else {
    // . . .
}
// Normalize
in = Normalizer.normalize(in, Form.NFKC);
```

Hence the strings should be normalized before validating them.

Compliant code // ***example***:

```java
in = Normalizer.normalize(in, Form.NFKC);

Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(in);
if (matcher.find()) {
    // Found black listed tag
    throw new IllegalStateException();
} else {
    // . . .
}
```

Also, Non-valid ASCII characters should be removed first before normalizing the code, if not these characters may slip unnoticed through the normalizer and be run.

So this meant:

1. Remove Non-valid ASCII characters
2. Normalize String that has been stripped
3. Validate String

```java
public class XSSFixed {
    public static void main(String args[]) {
    // assume "s" is the input that may be susceptible to XSS attacks
    String s = "\uFE64" + "script" + "\uFE65";

    // Deletes all non-valid characters
    s = s.replaceAll("[^\\p{ASCII}]", "");
    s = Normalizer.normalize(s, Form.NFKC);
    Pattern pattern = Pattern.compile("<script>");
    Matcher matcher = pattern.matcher(s);
    if (matcher.find()) {
        System.out.println("blacklisted tag");
    } else {
        // . . .
    }
    }
}
```

3.4.1 XML Injection

Consider an online store which stores purchase orders in XML with the following non-compliant code.

```java
private void createXMLStream(BufferedOutputStream outStream, String quantity)
throws IOException {
    String xmlString = "<item>\n<description>Widget</description>\n"
                + "<price>500.0</price>\n" +
                "<quantity>" + quantity + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
}
```

User selects "iPhone X" and inputs in the quantity field the following: `1</quantity><price>1.0</price><quantity>1`

The following order is thus generated.

```xml
<item>
    <description>iPhone X</description>
    <price>999.0</price>
    <quantity>
        1</quantity><price>1.0</price><quantity>1
    </quantity>
</item>
```

XML (SAX) parser (org.xml.sax and javax.xml.parsers. SAXParser) interprets the XML such that **the second price field overrides the first**, leaving the price of the item as $1.

example: Compliant code that would sanitize untrusted data passed across a trust boundary would be:

```java
private void createXMLStream(BufferedOutputStream outStream, String quantity)
throws IOException {
    // Write XML string if quantity contains numbers only.
    // Blacklisting of invalid characters can be performed in conjunction.
    if (!Pattern.matches("[0-9]+", quantity)) { /*Format violation*/ }
    String xmlString = "<item>\n<description>Widget</description>\n"
                   + "<price>500.0</price>\n" +
                   "<quantity>" + quantity + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
}
```

3.4.2 SQL Injection

An SQL injection vulnerability arises when the original SQL query can be altered to form a different query.

An SQL command to authenticate a user might take the form:

```sql
SELECT * FROM db_user WHERE username='<USERNAME>' AND
password='<PASSWORD>'
```

If it returns any records, the username and password are valid.

Case 1: With a valid username and any password, the user will be authenticated. The user inputs the following username: `validuser’ OR ‘0’=’0`, where validuser is a valid username.

The SQL command becomes `SELECT * FROM db_user WHERE username=’validuser’ OR (’0’=’0’ AND password=<PASSWORD>)`

Case 2: With any username, the user will be authenticated. The user inputs the following password: `‘ OR 0=’0`

The SQL command becomes `SELECT * FROM db_user WHERE username=‘’ AND password=‘’ OR 0=‘0’`

example: of SQL Injection Compliant Code

```java
public class SQLInjectionCompliant {
    String hashPassword(char[] password) {
        // create hash of password
        return null;
    }

    public void doPrivilegedAction(String username, char[] password) throws SQLException {
        Connection connection = getConnection();
        if (connection == null) {
            // Handle error
        }
        try {
            String pwd = hashPassword(password);
            if (username.length() > 8) {
                // do sth
            }
            String sqlString = "select * from db_user where
                                username=? and password=?";
            PreparedStatement stmt = connection.prepareStatement(sqlString);

            // Use the set*() methods of the PreparedStatement class
            // to enforce strong type checking.
            // This mitigates the SQL injection vulnerability
            // because the input is properly escaped by
            // automatic entrapment within double quotes.

            stmt.setString(1, username);
            stmt.setString(2, pwd);
            ResultSet rs = stmt.executeQuery();

            if (!rs.next()) {
                throw new SecurityException("User name or password incorrect");
            }
            // Authenticated, proceed
        } finally {
            try {
                connection.close();
            } catch (SQLException x) {
                // forward to handler
            }
        }
    }

    private Connection getConnection() {
        // TODO Auto-generated method stub
```

```
                return null;
            }
        }
```

3.4.3 Objection Orientation

Object-orientation allows us to encapsulate data and preserve invariants (sometimes security policies) among class members.

- OBJ00-J. Limit extensibility of classes and methods with invariants to trusted subclasses only
- OBJ01-J. Declare data members as private and provide accessible wrapper methods
- OBJ02-J. Preserve dependencies in subclasses when changing superclasses
- OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code
- OBJ05-J. Defensively copy private mutable class members before returning their references
- OBJ10-J. Do not use public static non-final variables

OBJ01-J

- Data members of a class must be declared private.
- Using wrapper methods enables appropriate monitoring and control of the modification of data members.

OBJ02-J When developers modify a superclass (during maintenance, for // ***example***:), the developer must ensure that changes in superclasses preserve all the program invariants on which the subclasses depend.

> In other words, don't anyhow add new methods in the parent class that will affect functionalities of the children class. Otherwise, at least throw an Exception in the new method so that whoever is maintaining the child class will notice when they run it OR update the child method appropriately.

OBJ04-J Mutable classes allow code external to the class to alter their instance or class fields. Just declaring variables as `final` does not ensure that it is safe from modification, because:

- For a normal object, assigning it as `final` means you cannot assign another object to this reference, but it does not stop you from invoking methods of this object.

example: of class object variable

```
final ObjectA objA = new ObjectA();
objA.setXXX() // legal
```

- For basic type variables(int, float, boolean etc), they don't have reference, pointer, and object method, so their value cannot be changed.

example: of primitive variable

```
final int IntegerA = 123;
IntegerA = 321; // not legal
```

example: of class object variable despite being related to primitive variables

```
final Integer A = new Integer(123);
a.someIntegerMethod(); // legal
```

Hence need to provide means for creating copies of mutable classes so that **disposable instances of such classes can be passed to untrusted code**.

example:

```
public final class MutableClass {
        private final Date date;
        public MutableClass(MutableClass mc) {
this.date = new Date(mc.date.getTime());
    }
    public MutableClass(Date d) {
this.date = new Date(d.getTime());
    }
    public Date getDate() {
return (Date) date.clone();
    }
}
```

4 Designing Test Cases

Black Box Tests	**White Box Tests**
Based on Specification	Based on Code
Covers as many specification as possible, hence better at finding whether code meets the specification or some specs are not implemented	Covers as much implemented behavior as possible, hence better at finding crashes, out-of-bound errors, file handling errors etc.

4.1 Equivalence Class Partitioning & Boundary Value Testing

Equivalent Class Partitioning is a black box technique (code is not visible to tester) which can be applied to all levels of testing like unit, integration, system, etc.

In this technique, you divide input conditions into groups (classes).

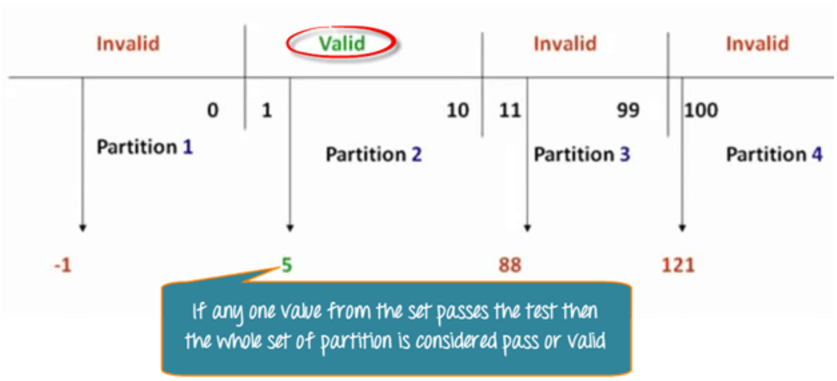
- Inputs in the same class should behave similarly in the program.
  - The hypothesis behind this technique is that if one condition/value in a partition passes all others will also pass. Likewise, if one condition in a partition fails, all other conditions in that partition will fail.

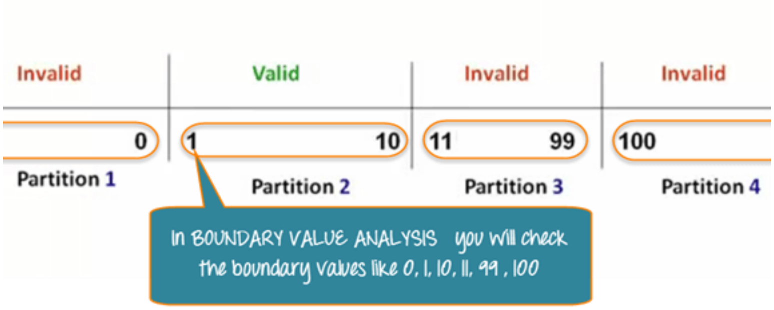
- Then we pick only one value from each partition for testing.

Boundary Analysis testing is used when practically it is impossible to test a large pool of test cases individually.

Using both together:

1. In Equivalence Partitioning, first you divide a set of test condition into a partition that can be considered.
2. Thereafter in Boundary Value Analysis, you test boundaries between equivalence partitions.


Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
-1
5
88
121
If any one value from the set passes the test then the whole set of partition is considered pass or valid


Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99, 100

This form of testing is used to reduce a very large number of test cases to manageable chunks, and is appropriate for calculation-intensive applications with a large number of variables/inputs.

4.2 Fault-based Testing based on Dirty / Failure Test Cases

- Can something cause division by zero?
- What if the input type is wrong
- What if the customer takes an illogical path through your functionality?
  - e.g. Enter invalid email address or URL
- What if mandatory fields are not entered?
  - e.g. Do not enter recipient's email to send email
- What if the program is aborted abruptly or input or output devices are unplugged?
  - e.g. Quit a game suddenly in the middle, does it save the state?

4.3 All coverage metrics

- **Node Coverage:**
  Achieved when the paths identified have a test that goes to every node in the graph.

- **Link Coverage:**
  Achieved when the paths identified have a test that goes along every link, or line, in the graph. In many cases, node coverage will take care of this.

- **Loop Coverage:**
  Achieved when the numerous paths identified have tests that explores the interaction between sub-paths within a loop. Statement coverage: Achieved when the flow of control reached every executable statement of source code at least once.

- **Path Coverage:**
  Achieved when all linearly independent paths in the program are *executed at least once*.

  - A path is defined as the number of nodes executed.
  - A linearly independent path can be defined in terms of what's called a *control flow graph* of an application.
  - If there is a while loop happening, depending on the input, there can be a lot of paths such that it is impossible to achieve path coverage.

example: code that has a loop and impossible to achieve path coverage.

```
int f1(int x) {
    int y;
    while (x < 0) {
        if (x >= 0)
            y = 1/x;
        else
            y++;
        x ++;
    }
}
```

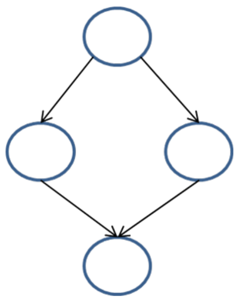
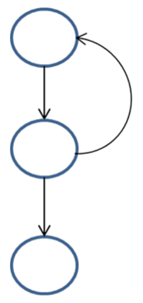
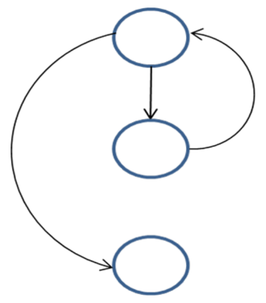
example: code that has a loop but achieves path coverage.

```
int f2(int x) {
    int y;
    int i = -10;
    while (i < 0) {
        if (x >= 0)
            y = 1/x;
        else
            y++;
        i++;
    }
}
```

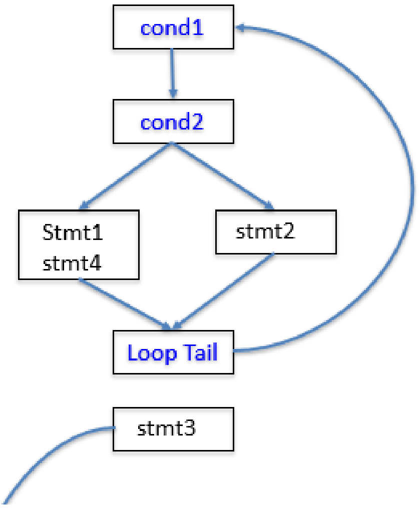
4.3.1 Control Flow Graph

For advanced white-box testing, it is often important to capture the control flow of the program.

example: basic control flow graphs

If	Until	While
		

example: a slightly more complex control flow graph



```
while (cond1)  {
    if (cond2)
        //stmt1
        //stmt4
    else
        //stmt2
}
//stmt3
```

4.4 Test Generation

You might be asked to:

- insert or reduce instrumentation code (where and why).
- reason whether it is ok to use random fuzzing, genetic algorithm or symbolic execution for making test inputs in terms of:
  - how probable it is to find the bug
  - performance of the method

4.4.1 Instrumentation

Instrumentation code helps to check if a certain path / condition is executed. But if there is too much, it will become a **performance overhead**.

> Try making a HTTP get request from your database for all your images and printing to console the Base64 string of every image you got for testing purposes :) You will get why reducing the overhead is important. (don't be like me...)

4.5 Genetic Algorithm

Might be given a model to map the stuff to Genetic Algorithm's operators.

- Fitness Selection
- Crossover (based on which individual is fitter)
- Mutation for randomized output.
- Chromosome/ Individual Initiation

can be used to generate randomized input such as:

- Strings e.g. Palindrome:
  - Chromosome: A char array of n letters
  - Crossover: for 2 inputs, randomly exchange letters to obtain child
  - Mutation: Randomly flip a letter
  - Fitness function: If the string has achieved palindromic property for ¿= 1 pair of letters
- Integers e.g. weird specific integer (!! symbolic execution better):
  - Chromosome: a bitvector representing integer 'x'
  - Crossover: for 2 inputs x1, x2, randomly exchange bits to obtain child
  - Mutation: Randomly flip a bit fo ran input x
  - Fitness function: both statement and branch coverage works

4.5.1 Symbolic Execution

It is good for finding out bugs which require specific values of the parameters to be found out.

example: method where symbolic execution test generation is appropriate

```
int fun3(int x, int y) {
    int y;
    if (x == 231301)
        y = 5/0;
}
// bug hidden under weird condition branch
// only 2 tests required: x = 231301 and x != 231301
```