`PLEASE COMMENT IF YOU FIND ERRORS OR HAVE BETTER ANSWERS

1.1 Which part to build/test first?

a) Constructing and verifying the functionality of the parts of application at lower levels that higher levels depend on first.

b) Constructing and verifying the functionality of pieces at higher levels that lower levels depend on first.

c) The overall or higher level functionality is ~~not known~~ not tested (or even built). You could verify all the bottom-up features and then have an error on the top level, and now everything below it needs to be rewritten. Thus, you would end up wasting a lot of time on building and testing and not even encounter the really hard problems. The big risks in the project are handled later (instead of handling the difficult design issues early).

1.2 Drivers and Stubs

a) Test drivers are the modules that act as a temporary replacement for a calling module ~~and give the same output as that of the actual product~~. Example: jUnit
From text: A driver is a routine that simulates a call from parent component to child component. Used for bottom up testing

b) Test stubs are programs that simulate the behaviors of software components (or modules) that a module undergoing tests depends on. They are used when testing classes ~~when you don't care about the functionality of a class the class your testing depends on.~~ This allows us to do top-down testing (see 1.1).
From text: A stub is a routine that fakes behavior of a child component. Used for top-down testing

c) A testing framework that allows you to mock the behavior of objects. For example, if I'm testing a bike object and it depends on the wheel object, I can program what functionality I want out of a fake wheel object so that I don't have to worry about the functionality of the wheel. This allows me to verify that the bike object works.

d)
   i) B b = new Mock(B.class);
   ii) when(b.doesSomething).then(doSomethingElse);
   iii) A.doTheThing; // assume that A uses b
   iv) Verify that A works ok.

1.3 Compile/Run/Debug Cycle

a) If the compile/run/debug time takes too long, debugging can take too long You could also spend a lot of time sitting around doing nothing if the compile/run takes a long time. For example, every time you find a bug you fix it, then recompile/run the program. It is better to find them all at once so you can spend more time writing code than finding problems.

b) Automated testing, where you don't need to test the code yourself, other code verifies its functionality.
From Piazza:
Use mock objects. Ex: If you are experiencing arithmetic errors (and you know the front end is sending the right information to the server) during backend type processing when coding an app, there is no reason why you should have to boot up the frontend client just to debug a backend issue. You can use tools like Postman, etc. Isolate the problem, if your application needs to perform some server call, or do A,B,C,D that takes 2 minutes, simulate the input and iron out the bugs that way.

1.4 - Class Diagram

See the document Scan_Dec_7_2018.pdf provided by a student

.

1.5 - Sequence Diagram

Answer Here **Hi Noah!!**

1.6 Dependency Injection

```
public class TestClass {
        Public static void main(String[] args) {
                IVolleyComm c = new VolleyComm();
                GameLogic g = new GameLogic(c);
                g.playGame();
        }
}

interface IVolleyComm {
        Void sendToServer();
}

class VolleyComm implements IVolleyComm {
        Void sendToServer() { }
}

class GameLogic {
        IVolleyComm i;

        GameLogic(IVolleyComm i) {
                This.i = i;
        }

        Void playGame() {
                i.sendToServer();
        }
}
```

Another example

1.7 Visitor Pattern

Slide 8

2.1 Testing techniques

a) Inputs: 18,65 Nope: You need more! Added after Mitra's Comment: For boundary testing, you need stuff outside boundaries. So 17 and 66 would be other inputs. You should also have a typical value input, say, 40, also 19 and 64, to test one near bounds, while still within.

Edit :"Boundary testing or boundary value analysis, is where test cases are generated using the extremes of the input domain, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values." So add NaNs and negative numbers to the list of values to test

b) inputs: 20, 21,19 (Explain!) Added after Mitra's Comment: 21 is used to test the doSomething() potion of code. 20 is used to test the doSomethingElse() portion of code and 19 is used to see what happens if a value under 20 is inputed. The code ends here so unkown behavior might occur.
Test case 20: if (c >= 20) → TRUE → doSomething(); → if (c==20) → TRUE → doSomethingElse();
Test case 21: if (c >= 20) → TRUE → doSomething(); → if (c==20) → FALSE
Test case 19: if (c >= 20) → FALSE

Decision	**True**	**False**
c >= 20	When c = 20 or 21	When c = 19
c == 20	When c = 20	When c = 21 (not c = 19, it doesn't reach this decision)

2.2 Automated testing

1) boolean oracleForFFN(Graph g, int v, List list) CORRECT!
2) ~~return findFriendlyNodes(g, v).equals(expected);~~

```
boolean oracleForFFN(Graph g, int v, List list){
 List<Node> nodes = new List<Node>();
 for (node n in g){
  if (n.getNumberOfNeighbors() > v){
   nodes.add(n);
   if (!list.contains(n)) {
    return false;
   }
  }
 }
 if(nodes.size() != list.size()){
  return false;
 }
 return true;
}
```

~~Sigh! No.~~
~~Here, write pseudo code to show how you would check that the "list" is the correct answer to the question. You DO NOT HAVE "expected".~~
SM: great! (1) name should not be findFriendlyNodes -- that is the S.U.T.

3) Regression testing is important, as it will tell you if the old code will work after a new change.

   If updates are rolled out to customers with errors to EXISTING features, then customers lose trust with your software development practices and it is a nightmare AND expensive to try and fix the problem.

2.3 How to and difficulty of automating tests

1) Inputs can be generated randomly, and the test case itself can calculate what the expected output should be. This is easy to implement but is unlikely to be to the same quality as manually generated test cases. Ex. A method that tests the calculation of the volume of a cube can be passed 8 points. The test case can calculate the volume and compare it to the actual method.

SM: We can create millions of random inputs (using random number generators). That can be truly automated! We will not be able to generate EXPECTED outputs. That's why we will use our ORACLE.

2) SM: jUnit is a driver! We can also create scripts using any scripting language or even HLL to call S.U.T (System under Test). Also, there are ways (called data driven testing) to read inputs from files and call our tests for each input.
3) SM: We will not have to compare expected and actual outputs. Instead, our oracle will check if our result is correct.

   You can test this with an oracle, so it would be relatively easy to implement. For example, if you have a falling object in a physics engine, you can give the expected position at a certain time. The oracle would compute what the output should be and compare it to what the actual method computed.

2.4 Importance of automation

It is more efficient and quicker to have the testing automated. Manual testing is also error-prone (i.e. aside from being expensive and time-consuming). Also, regression testing necessitates frequent re-testing. Hence, it is necessary to automate testing as much as possible. You can make sure the newly written code does not break other parts of the program.

2.5 Black Box Test Case Generation

Boundary value testing: test with inputs around the minimum, maximum, and nominal values. If there are multiple variables, vary one at a time in this manner.

Equivalence class testing: determine groups of inputs that will cause the S.U.T to behave similarly, and use one input/set of inputs from each group to theoretically test for all inputs/input sets in that group.

Random testing: assuming an even distribution of input values, determine how often each result should occur (percentage). Randomly generate a large number of inputs, recording the result for each, and compare the frequency of the actual results to the expected ones.

2.6 Code Coverage Techniques (White Box)

Statement coverage: given an input or set of inputs, determine the percentage of the application code executed, striving for 100% execution. Attempt to find the minimum amount of different inputs needed to cover all lines of code.
Decision/Branch coverage: make sure each decision has been tested as both true and false
Condition coverage: for decisions that contain multiple expressions, make sure that every sub-expression has been independently tested as both true and false. This is essentially a beefed-up version of decision coverage.

3.1 Hospital Ethics (ETHICS NOT ON EXAM)

a. By being able to recognize ethical issues and evaluate the situation (don't be an idiot?) Smell test or Des Moines Register Test?
b. It violates the Judgement principle, because you have not revealed a conflict of interest
c. Five Approaches:
    i. Common Good: My lack of disclosure violates the communities' trust
    ii. Utilitarian: I am a stakeholder, I am positively affected by this. Not disclosing this is not morally right

iii. Rights: I might be biased in my choice of A-star systems, since I am a co-owner. This violates to right to a fair trial.
iv. Virtue: Will you be able to, while basking in your newfound wealth, be able to look at yourself in the mirror by misleading the hospital?
v. Fairness/Justice: Are the benefits and burdens more equally distributed between everyone (hospital faculty, patients, stakeholders) compared to the other proposals?

4.1

~~The SCRUM process model involves setting a list of features in the sprint backlog, using sprint to implement desired features in a short amount of time (2 weeks) and meeting every day to ask questions and give the individuals status (usually 15 minutes).~~

~~Disadvantages~~

- ~~There is a temptation to add more features since there is no definite end date~~
- ~~Does not work well with novices since time needs to be spent teaching them~~
- ~~If timelines are wrong, the entire schedule gets thrown off~~

I would describe using the main ideas in each of the following:

(1) Scrum Basics

Scrum is a process model. Defined by a series of "Sprints", where each Sprints are 2-4 weeks, maximum 1 month. Products are defined, coded, tested in the period of these sprints.

(2) Scrum Roles

There are three main Scrum Roles:

Product Owner :

Defines features of the product, and focuses on marketability of the product. Accepts/rejects work results.

Scrum Master:

Takes on the role of management, enacting scrum values & practices. Allows the team to work together, and facilitates it by removing external impediments and encouraging inter-personal work.

Team member:

Can have different functions(coder, tester, user experience designer, etc), which are usually full time. Membership is static between sprints

(3) Scrum Artifacts:

Product Backlog:

Requirements(user stories) such that it provides value to customer or user. The priority is set by product owner.

Sprint Backlog:

List of tasks for sprint, selected from product backlog. They include estimated time to completion.

Scrum Board:

Visualization of scrum progress.User stories and tasks are written on post its, and are moved among todo, in progress, done.

(4) Scrum Ceremonies

Sprint Planning: Team selects items from product, and commit to completing.

Daily Scrum: 15 minute, stand up where everybody is invited.
People in the scrum are the only ones invited.

Questions to be answered: What did you do, what will you do, what is in your way?

Sprint Retrospective: After every sprint, team members gather to discuss:
Start, continue, stop doing.

Sprint Review: Invite the whole world, and a mini-demo of new features.

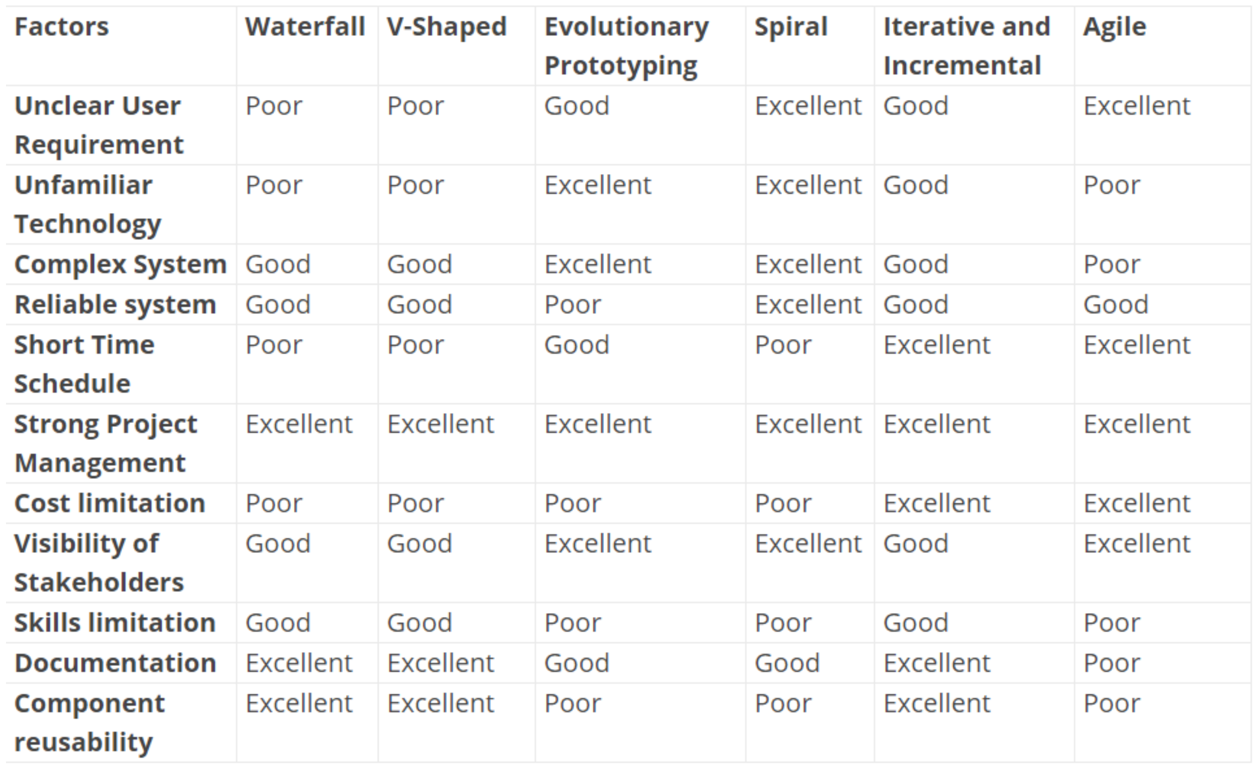
Regarding disadvantages (see slide#18 of some process models).

4.2

1. Agile - Longform process where requirements change constantly, the customer is heavily involved and features are added in sprint(?). Benefits long, ongoing projects with no definite end date and constantly changing requirements.
2. Waterfall - A sequential process where each fundamental activity of a process is represented as a separate phase, arranged in linear order. Useful when requirements are well understood and unlikely to change.
3. Spiral - Risk driven where the process is represented as a spiral where risk assessment is an added features. Similar to waterfall. Useful for high risk situations where the requirements are ambiguous.

4.3 Choosing a Process Model

Depends on many factors such as how clear the requirements are, the complexity of the system, and how long the time schedule is. First determine which factors are most important, then evaluate which process model benefits those factors the most.



Factors	Waterfall	V-Shaped	Evolutionary Prototyping	Spiral	Iterative and Incremental	Agile
Unclear User Requirement	Poor	Poor	Good	Excellent	Good	Excellent
Unfamiliar Technology	Poor	Poor	Excellent	Excellent	Good	Poor
Complex System	Good	Good	Excellent	Excellent	Good	Poor
Reliable system	Good	Good	Poor	Excellent	Good	Good
Short Time Schedule	Poor	Poor	Good	Poor	Excellent	Excellent
Strong Project Management	Excellent	Excellent	Excellent	Excellent	Excellent	Excellent
Cost limitation	Poor	Poor	Poor	Poor	Excellent	Excellent
Visibility of Stakeholders	Good	Good	Excellent	Excellent	Good	Excellent
Skills limitation	Good	Good	Poor	Poor	Good	Poor
Documentation	Excellent	Excellent	Good	Good	Excellent	Poor
Component reusability	Excellent	Excellent	Poor	Poor	Excellent	Poor

Source: https://melsatar.blog/2012/03/21/choosing-the-right-software-development-life-cycle-model/