1.

There are several problems that occur within elicitation, but the three that are the most prevalent are: understanding between stakeholders and the development team, changing of needs as the market changes, and not properly defining the scope of the project.

A lack of understanding between the parties involved in a piece of software's creation can lead to requirements that are ambiguous, or even unnecessary. Usually, the stakeholders will think of the software from a user's perspective, which is inherently incomplete, so there must be ample communications between them and the development team in order to truly understand what they need, not just what they ask for.

The changing market, and time's progression in general, are the enemy for a development team working on a project. The longer they take, the more stakeholders will find new requirements to add to the project. Additionally, as technology standards keep changing, if a team takes too long to develop some software, then the stakeholders may need something entirely different by the time it is finished.

Finally, if a team does not properly communicate to the client what the limitations are for a product, then the requirements may end up being unrealistic, unfocused, and difficult to implement. The team needs to focus their efforts and create a rudimentary set of limitations to give a frame of reference to the stakeholders and themselves. That way, they will be able to better create the requirements.

2.

Functional requirements are exactly that, functional. They are things a program needs to do for the stakeholders and work correctly. These would be things like "the program needs to have a keyword search functionality" or "the program must be able to calculate integrals for the end user."

Non-functional requirements, on the other hand, are a lot more about how something gets accomplished. These should be measurable, and not vague, but they do not specify something the program does. These would be something like "the program must be able to run on a system with 4 GB of RAM or less" or maybe "the program must be able to run 24/7 for at least three weeks straight." Those examples may be a bit extreme, but it should get the point across well enough.

3.

Scenario-based modeling is when the development team models the requirements around how different people, usually called "actors" will be interacting with the software. These actors can include end-users, system administrators, IT professionals, and many more.

Class-based modeling is when the team focuses more on looking at every piece of the software as an object. The people using the system are objects, functions are objects, and so on. Then they get sorted into classes and labeled as problem or solution.

Behavioral modeling is based around the how the program will react to different events that occur either to it or within it. This allows the team to create use cases and UML diagrams based upon these events.

Flow-oriented modeling is based around the functions of the system, and how the inputs and outputs of each are connected to each other. Breaking it down to a grossly simple level, it is creating a flow chart for the movement of data through a system.

Data modeling is when the team puts together the information domain for the application.

4.

Separation of concerns is the practice of looking at each "concern" (functional requirement, behavior, etc.) and splitting it into smaller pieces. This makes tackling each one less daunting, and it can be distributed to more than one team member. This, ideally, makes the concern easier to create and/or solve.

Functional independence takes the idea of separation of concerns, and it cranks it up to 11. The idea behind it is to separate modules into small parts that do less tasks. They continue to do it, until each piece is practically independent from one another at the base-level, and they are just put together to make the function. Keep in mind that these functions will still need to work well together.

5.

Data-centered architecture is based around a central data repository (go figure), and each software client accesses it independently from the rest. From the way I understand it (and this could be wrong), it works somewhat like how GitHub does.

Data-flow architecture is based around sending data from one module, to the next. Each module does not care how the previous functions, it only cares about the input they receive from it. This allows software to be worked on with less collaboration.

Call-and-return architecture is based around a program using subprograms to achieve different functions. Those subprograms can have subprograms of their own as well. This makes the program easy to scale and add new functionality to it.

Object-oriented architecture is based around the idea of object-oriented programming. With that being said, the components of the system are used to both hold data, as well as functions to access/change the data. Different components communicate with each other through "passing messages" to each other.

Layered architecture is based on assigning components to layers within the system, with outer layers interacting with the layer beneath it. The outermost layers are user-interface type functions, and the innermost functions are the core functions that work with the operating system itself.