150 Shades of EIST:

1. *Software Development is a problem solving activity. It involves dealing with complexity and change.*
2. *Complex problems are hard to understand and analyze. Moreover it is difficult to find a starting point when dealing with them.*
3. *First of all the problem and its requirements are often ambiguous / unclear. The problem domain and solution domain of non-trivial domains are complex. Software has many hidden surprises and is very flexible. Furthermore the project, its parts as well as the process are difficult to manage because they are subject to change.* ***Change is the only constant.***
4. *We try to reduce complexity and change: We deal with complexity by using methodologies such as Modeling, Notations, Analysis and Design, Implementation and Testing. Furthermore we define a process of System development. The Release Management, Delivery, Software Life Cycle Modeling, Rationale Management, and Project Management help us when dealing with change.*
5. *A phenomenon is an object as we perceive it in the real world. A concept describes the common properties of a set of phenomena (Name, Purpose, Members). The process of abstracting is grouping phenomena into concepts. When Modeling we try to abstract complex parts of the application domain so as to simplify them. We reduce information in the process. A notation is only a set of rules for describing models and views.*
6. *A Model is an abstract description of an entire System and can be represented as text and/or Diagrams (UML). A view only depicts selected parts of a model.*
7. *Textual and Visual Models (UML: Use case, Class, Communication, Activity, Sequence, etc.)*
8. *In Software Development we develop solutions to given problems.*
9. *A Software life cycle is a set of activities and relationships to each other to support the development of a software system.   
   A Software life cycle model is an abstraction of this Software life cycle.*
10. *Requirements Elicitation, Analysis, System Design, Object Design, Implementation, Testing, Delivery and Maintenance.*
11. *The process of Tailoring involves Naming, Cutting and Ordering. We use it to adjust our software lifecycle to our requirements. Naming is adjusting the names of activities, Cutting is removing unimportant parts of the project and Ordering is ordering the activities.*
12. *We can either control Software Development through Organizational Maturity or Agility. We describe this process with two different Models: Defined Control Process Model and Empirical Control Process Model.*
13. *In a Defined Process Control Model we work with a well-defined set of inputs and activities. We do not accept deviations and have Preconditions imposed on the model. If these are not satisfied it can lead to chaos. We use this Model when we have no change and a predictable output (aka never). (e.g. Waterfall Model)*

*In an Empirical Process Control Model, we have a less well defined set of inputs and not the same outputs are generated every time. Deviations can be explored. Control and risk management need to be exercised. (e.g. Scrum)*

1. *We can use UML for Communication, Analysis, Design and Archivation.*
2. *The Application or Problem Domain is the problem that we have to solve as well as its environment. The Solution Domain is the solution we develop – it processes application domain knowledge.*
3. *Abbot’s technique is a technique to map Text (Problem Statement) to parts of a Class Diagram.*
4. *During Requirements Elicitation we employ the Use Case diagram. It consists of actors, who interact with the system, Use Cases, which represent functionality provided by a system as well as its event flow – Example: User (Actor) can directly access the Functionality Steer (Use case) which then triggers the Functionality Change Direction (Use case).*
5. *See 16.*
6. *The <<extends>> and <<includes>> relationships exist. The former is used when describing very specific extensions of other use cases, e.g. “No Money” <<extends>> “Purchase Tickets”. The latter is used when a use case is part of many other use cases, in which case the others <<include>> this use case, e.g. “Purchase Single Ticket” <<includes>> “Collect Money”.*
7. *Boundary Objects, Control Objects and Entity Objects. Piston, Loop, Circle with a line below.*
8. *Entity Objects represent the persistent information tracked by the system. We track application domain knowledge/objects.  
   Boundary Objects display the interaction between the user and the system.  
   Control Objects represent the control tasks by the system, (e.g. relaying Application Domain Knowledge to the user).*
9. *Actors are users/systems that interact with the system. Classes are blueprints that define parts of the system. Objects are instantiations of these blueprints.*
10. *We can describe the Dynamic Model with Sequence Diagrams, State Diagrams, (Communication Diagrams and Activity Diagrams?).*
11. *Activities in state chart diagrams are candidates for public operations in classes. Activity lines in sequence diagrams are candidates for objects. Flow of events in sequence diagrams. Find sender and receiver of events (heuristics).*
12. *Something that happens at a point in time. Can be related or unrelated to other events. Often used to describe the instance of an event class as well as the values of the attributes of an event class.*
13. *Its elements can be organized: By System Hierarchy (Enterprise, Systems, components and classes), System Model (Functional, Object and Dynamic), Domain (Application or Solution), Stakeholders and different view-points.*
14. *Requirements Elicitation is the system in terms understood by customer/user. Analysis is the definition of terms as understood by a developer. Requirements Engineering is the combination of the two activities.*
15. *User Interaction, Error handling, Handling of Environmental Conditions, Functionality.*
16. *Use case model is created in Requirements Elicitation. Analysis expresses this model in application domain objects. System Design expresses this in appropriate Subsystem. In Object Design the System required from the Use Case Model is realized with Solution Domain Objects. In the Implementation phase we implement them with source code. Last but not least, in the Testing phase we validate the model. (Questionable Slide we implement the Solution Domain Objects in code, and validate the code).*
17. *URPS: Usability, Reliability (Robustness, Safety), Performance (Response Time, Throughput, Availability, Accuracy), Supportability (Adaptability, Portability, Maintainability).*
18. *Constraints: Requirements imposed by external Sources on the System that the System has to fulfill. Unrelated to functionality and URPS. (e.g. Packaging Requirements, Interface Requirements and Legal requirements).  
    Requirements: Functionality and URPS that the system has to offer. (e.g. the Car must be able to drive, or: The car must be able to reach a speed of 100km/h.*
19. *Scenario: Describes the* ***use of the system*** *as a series of interactions between a* ***specific*** *user and the system. / Concrete, focused, informal description of a single feature of the system used by a single actor  
    Use Case: Describes a* ***set*** *of scenarios between a* ***generic*** *end user(actor), interacting with the system.  
    User Story: Describes a* ***functional requirement*** *from the* ***perspective******of******an******end user****.*
20. *As-is scenario: scenario of a current situation 🡪 Reengineering Projects  
    Visionary scenario: scenario of a future system 🡪 Reengineering Projects  
    Evaluation scenario: scenario of a user task, for the purpose of evaluating the system   
    🡪 Demonstration and Acceptance Tests  
    Training scenario: scenario where a user is introduced to the system 🡪 System delivery test*
21. *A Problem Statement describes the Problem that the customer needs a Solution for. This is issued during the Requirement Elicitation by/with the customer.*
22. *Correctness, Clarity, Completeness, Consistency, Realism, Traceability.   
    Correctness 🡪 Requirements fit the clients needs  
    Clarity 🡪 Requirements are not ambiguous  
    Completeness 🡪 All requirements have been captured  
    Consistency 🡪 No contradicting requirements  
    Realism 🡪 Requirements are fulfillable  
    Traceability 🡪 The Systems behavior can be traced back to the functional requirements (?)*
23. *Description of System Structure, Development Methodology, Development Environment (language), Reusability 🡪 Should not be restrained by the client*
24. *Actor 🡪 Boundary 🡪 Control 🡪 Entity*
25. *Sequence diagrams represent behavior in terms of interaction and order. They also help identify the relationship between objects over time. .*
26. *Statechart diagrams help to identify changes to an individual object over time.*
27. *Communication Diagrams visualize the interactions between objects as a flow of messages. Messages can be events or calls to operations. Describe the static structure as well as dynamic behavior. Activity Diagrams have no information (?)*
28. *Identify Design Goals, Subsystem Decomposition, Identify Concurrency, Hardware/Software Mapping, Persistent Data Management, Global Resource Handling, Software Control, Boundary Conditions*
29. *Nonfunctional Requirements 🡪 Design Goals  
    Functional Model 🡪 Subsystem Decomposition  
    Object Model 🡪 Hardware/Software Mapping, Persistent Data Management  
    Dynamic Model 🡪 Identification of Concurrency, Global Resource Handling, Software Control  
    Hardware/Software Mapping 🡪 Boundary Conditions*
30. *Nonfunctional Requirement 🡪 Govern the system design activities, complemented by stakeholder goals regarding Design methodology, Design metrics and Implementation goals*
31. *Client: Low cost, high productivity, backward compatibility, traceability of requirements, rapid development, flexibility, Runtime Efficiency, Reliability  
    Developer: Well-defined interfaces, Reusability, Adaptability, Modifiability, Readability, Minimum # of errors, Portability, Good documentation, Reliability  
    End User: Functionality, User-friendliness, Usability, Ease of learning, Fault tolerant, Robustness, Runtime Efficiency, Portability, Good documentation, Reliability*
32. *Functionality 🡨🡪 Usability 🡪 Tum-online, many possibilities, hard to get started with  
    Cost 🡨🡪 Robustness 🡪 My self-programmed applications are less robust, as not as much time/resources were invested in order to fix bugs etc.  
    Efficiency 🡨🡪 Portability 🡪 In order to make Games run on smart phones the requirements have to be toned down compared to other games, as the hardware is not as capable  
    Rapid development 🡨🡪 Functionality 🡪 Our ERA Project had to be finished quickly, thus we did not include as many tests  
    Cost 🡨🡪 Reusability 🡪 My self-programmed projects do not have much reusability value as I developed them quickly, due to a lack of time and interest (cost/resource)  
    Backward Compatibility 🡨🡪 Readability 🡪 In order to keep up the backward compatibility some functions in programs have to be left the way they are, which prevents them from being updated and becoming more readable*
33. *A subsystem is a system that is part of another system. A Service is a method offered by an interface. A Subsystem interface is a collection of methods offered by a subsystem. An API is a specification of a Subsystem interface in a specific language. A Subsystem interface object is a set of public operations provided by a subsystem, it describes all the services offered 🡪 can be realized with the façade pattern.*
34. *Cohesion measures the dependency and interaction between classes/objects. Coupling measures the dependency and interaction among subsystems. What we want to achieve is High cohesion within a subsystem and low Coupling between subsystems. Why? If high cohesion does not exist in a subsystem it should be split up as it does not really belong together. Low Coupling prevents failure within a Subsystem from spreading out to all other subsystems.*
35. *High cohesion is achievable if most interactions take place within subsystems. Low Coupling is achievable if the calling class does not need to know about the internals of the called class. (Införmatiön hiding). 🡪 Try to aim for this kind of design*
36. *Subsystem decomposition: Decomposing the subsystem into multiple smaller subsystems, so as to increase cohesion within subsystems while maintaining low coupling between subsystems.  
    Architectural Style: A Pattern for subsystem composition  
    Software Architecture: Implementation of an Architectural Style  
    Layered Systems: Hierarchical implementation of a subsystem. Only lower layers can be accessed. (Differentiation closed/open)  
    Hierarchical 🡪 reduces complexity.  
    MVC Pattern: Program split into Model, View and Controller Subsystem. View displays information, Model tracks application domain knowledge, Controller relays input to the model and updates the view. (Alternatively: Model updates the view).  
    Client/Server Architectural style: The server offers an interface which can be accessed by multiple clients. 🡪 Can/Should be run on different hardware architectures.*
37. *See 49.*
38. *Layer = Virtual Machine, a type (e.g. class, subsystem), Tier is an instance (e.g. object, hardware node)*
39. *Functional and Object-oriented decomposition. Object-oriented composition, because otherwise source code is hard to understand, complex and impossible to maintain.*
40. *See 49.*
41. *See 49.*
42. *Consists of Filters and Pipes: A filter is a subsystem that does a processing step. A pip is a connection between two processing steps. Every filter has an input pipe and an output pipe. Example of a Pipes and Filters architecture: Unix.*
43. *Components (Subsystems): Computational units with a specified interface.  
    Connectors (Communication): Interactions between components.*
44. *Thread of control: A path through a set of state diagrams where only a single object is active.  
    Concurrency involves Thread splitting: Object does not wait after sending an event to another object 🡪 Thread splits up. Objects are inherently concurrent if they can receive at the same time without interacting, they can then be assigned to different Threads of control.  
    Careful of race conditions.*
45. *Physical concurrency (Hardware) and Logical concurrency (Software).*
46. *Externally imposed hardware and software constraints. Control Objects -> Processor, Entity Objects -< Memory, Boundary Objects -> Input / Output devices.*
47. *Component Diagram 🡪 dependencies between components at design, compilation and run time (top level view of the system design in components and connectors)  
    Deployment Diagram 🡪 distribution of components on concurrent processes at run-time  
    (graph of nodes and connections)*
48. *File System and Databases.*
49. *Global Resource handling means managing access control and the rights for different classes of actors. We can manage it by defining access rights.*
50. *Access Matrix, Access Control List, Capability.*
51. *Implicit control / Explicit Control 🡪 Centralized control (Procedure-driven, Event-driven) resides with one object (sequence diagram fork diagram), Decentralized control (sequence diagram stair diagram).*
52. *Initialization, Termination, Failure. Use-cases.*
53. *Prepare for the implementation of the system model based on design decisions.*
54. *Reuse, Interface specification, Object model restructuring, Object model optimization*
55. *Reuse of design knowledge, existing classes, existing interfaces  
    Object design gap:   
    Composition, new class created by aggregation of existing classes: Delegation  
    Inheritance, new class is created and functionality reused*
56. *Delegation/Composition 🡪 Black-box, Implementation/Specification Inheritance 🡪White-box*
57. *Specification and Implementation Inheritance*
58. *Generalization and Specialization.*
59. *Structural Patterns: Reduce coupling, Introduce abstract classes for extensibility, Encapsulate complex structures  
    Behavioral Patterns: Allow choices between algorithms, assignment of responsibilities to objects, model complex  
    Creational Patterns: Allow abstraction from complex instantiation processes, make system independent from creation*
60. *Patterns and Relationships between Classes / Objects.*
61. *Reducing complexity, solve certain problems in a reusable/elegant manner.*
62. *See 73.*
63. *Later*
64. *Later*
65. *A methodology focused on creation of domain models (abstract representation of the knowledge and activities for a particular application domain)  
    🡪 Reuse standardized models, knowledge, collaboration between developers, terminology, best practices*
66. *Communication and knowledge management, analysis of requirements and design changes, complete representation*
67. *Model transformation 🡪 redesign of a model. We can reuse and improve old models by transforming the according to our requirements and needs.*
68. *Forward Engineering, Backward Engineering, Model refactoring, Source Code Transformation (Model Space and Source Code space)*
69. *A change made to the internal structure of code, to make it easier to understand, modify it without changing its observable behavior. (same for models I guess)*
70. *Contracts are constraints on a class that enables class users, implementors and extenders to share the same assumptions about the class.*
71. *Precondition (Exception, Predicate), Postcondition (Predicate, Exception, Specific Exception) and invariant (checked at the same time as postconditions).*
72. *Object-oriented Database, Relational Database, NoSQL Database. Extra parent table or vertical mapping in which parent columns appear as redundancy.*
73. *We transform objects to JSON strings.*
74. *A pattern describes a reoccurring problem and the core of the solution. The solution is almost never implemented in the same way. Meanwhile an algorithm solves a problem using a finite sequence of well-defined inputs, instructions and outputs.*
75. *Analysis Object Model: Attributes and methods without visibilities or signature 🡨🡪 Object Design Model*
76. *A software life cycle is a set of activities and relationships that describe the process of software development. A software life cycle model is a model of this process.*
77. *The selected activities, the dependencies between them and how they should be scheduled.*
78. *See 92.*
79. *Waterfall model, V-Model.*
80. *Engineering Stage (Inception, Elaboration), operated in smaller more unpredictable teams and Production Stage (Construction, Transition), operated in larger more predictable teams.*
81. *They do not deal well with frequent change.*
82. *Incremental 🡪 V-Model, Waterfall Model  
    Iterative 🡪 Spiral Model, Unified Process  
    Adaptive 🡪 Scrum*
83. *Defined Control Process Control Model: Well-defined input with the same output. Requires that every piece of work is well understood before starting the process. Change must be ignorable, output predictable 🡪 may lead to undefined behavior otherwise (Waterfall)  
    Empirical Process Control Model: Imperfectly defined process, not all pieces of work well understood. Control and risk management exercised through frequent inspection.*
84. *Entities: Product Backlog, Sprint Backlog, Potentially shippable Product Increment.  
    Activities: Project Kickoff Meeting, Sprint planning Meeting, Daily Scrum Meeting, Sprint, Sprint Review Meeting.  
    Roles: Scrum Master, Product Owner, Developer.*
85. *I did.*
86. *Scrum Master: Responsible for enacting scrum values and practices, Main job is to remove impediments, Typically filled by a project manager or team leader, should moderate and coach the team  
    Product Owner: Knows what needs to be build and in what order, responsible for the product, value and prioritization, typically a product manager.  
    Developer: Responsible for realization of the potentially shippable product increment*
87. *Effective use of time and money, divided into easily manageable sprints, Developments coded are already coded and tested during sprint review, works well for fast moving projects, much interaction between team members, very adaptable to feedback.  
    The framework needs experienced team members succeed. Daily Meetings may be frustrating, members leaving/being absent in the midst of development has a very negative impact, as teams are cross-functional.*
88. *A set of management disciplines within a software engineering process to develop a baseline. Encompasses disciplines and techniques of initiating, evaluating and controlling change to software products during and after a software project. 🡪 Fixes needs for coordination*
89. *Configuration Manager, Change Control Board Member, Developer and Auditor*
90. *Configuration item identification, Change Management, Promotion Management, Branch Management, Release Management, Variant Management*
91. *Config. Item: An aggregation of hardware, software or both for configuration management and treated as a single entity in management activities. Not only source files but all types of artifacts.  
    Version: An initial (intern) release or re-release of a configuration item associated with a complete compilation or recompilation of the item. Different functionality.  
    Baseline: A specification or product that has been reviewed and agreed to by responsible management.  
    Release: A Publication of a new Build/Version of the System for the customer(s)  
    Promotion: Distribution of a version to another developer  
    Revision: Change to a version that corrects only errors in the design/code but does not affect functionality*
92. *I did.*
93. *Programmer’s Directory is completely under control of one programmer. Master’s Directory is a central directory of all promotions. Software Repository contains externally released baselines.*
94. *Change Management is the handling of change requests. Change Policy ensures that each promotion or release conforms to commonly accepted criteria.*
95. *Version Control System. Monolithic Architecture, Repository Architecture. Peer-to-Peer Architecture.*
96. *Monolithic architecture: Simple local database, keeps all the changes to files under revision control  
    Repository Architecture: Single server contains all versioned files. Programmers check out files from the server to their computer, they change them and check them back into the server.  
    Peer-to-Peer: Each programmers directory fully mirrors master directory. Programmers can work offline on their own work. Can back up in case of failure.*
97. *Git add 🡪 adds files to set of changes for the next commit (staging area)  
    Git commit 🡪 update local repository with selected changes  
    Git push 🡪 move version from local repository to remote repository  
    Git fetch 🡪 move version from remote repository to local repository  
    Git merge 🡪 merge changes from local repository to programmers directory  
    Git pull 🡪 fetch and merge  
    Git clone🡪 creates a copy of the master repository (fetch & initial checkout)*
98. *Commit related changes, commit and push often, do not commit half done work, test before commit, write meaningful and understandable commit messages, do not use vcs as backup, keep working copy up to date, do not change history - COORDINATION*
99. *Product baseline, Developmental baseline, Functional baseline*
100. *Master branch 🡪 external releases, Development branch 🡪 central directory of all promotions, possible release version, Feature Branch 🡪 managed by one developer*
101. *Avoid unexpected failures due to big/late integration processes. Easy to understand integration processes.  
     Software dev. Technique where work is frequently integrated*
102. *Verify that previously developed software still works after updating it.*
103. *Continuous Integration: Technique where each developer promotes and integrates their work frequently.  
     Continuous delivery: Teams keep producing valuable software in short cycles and ensure that it fulfills baselines for release.  
     Continuous deployment: Every change that passes automated tests is deployed automatically.  
     Continuous software engineering: Organizational capability to develop release and learn from software in short cycles.*
104. *Failure: Deviation of observed behavior from specified behavior  
     Error: System is in a state that further processing can lead to failure  
     Fault: The mechanical or algorithmic cause of an error (= bug)  
     Validation: Activity of checking for deviations between observed behavior and its specification (=expected behavior?)*
105. *Fault avoidance, detection and tolerance 🡪 Use methodology to reduce complexity, prevent inconsistencies via Configuration Management and apply verification to prevent algorithmic faults. Use reviews to identify faults already visible in the design.  
     Fault detection: Testing, Debugging, Monitoring  
     Fault tolerance: Exception handling, Modular redundancy*
106. *Test model consolidates all test related decision and components into one package. It consists of a test driver 🡪 program that executes the test case  
     a test case 🡪 Test / Function that is usually derived from a use case  
     input data 🡪 Input Data necessary for test cases  
     oracle 🡪 the expected output  
     test harness 🡪 framework or software components that allows to run tests under varying conditions and monitor the behavior and outputs of the SUT*
107. *Automated generation of test data and test cases. Running the test is also done automatically and comparison with the oracle as well. 🡨🡪 Manual: Developer sets up runs and examines test results.*
108. *System Model is used for generation of the test model. SUT, part of the system model which is being tested.*
109. *MDE: Development method focused on creating and using domain models. Usually used when development consists of hardware and software components.  
     MDD: When googling it you get the same results as MDE. Generating Code from Models I guess.  
     MDA: System functionality defined using a platform-independent model and an appropriate domain-specific language.*
110. *Dummy Object 🡪 does nothing, Fake object 🡪 contains shortcuts, Stub 🡪 gives same answer, Mock Object 🡪 mimic behavior of real object for a specific sequence of classes (real subsystem should be specified with a façade)*
111. *Dynamic Analysis: Black-box testing(In-,output) and White-box testing (statement, loop, path, branch)  
     Static Analysis: Hand execution by reading through code (Debugging, Monitoring?), walk-through by informal presentation, Code inspection by formal presentation, automatic tools*
112. *White-box Testing: When trying to validate the functionality of our system as well as the internal behavior matching the expected internal behavior.*
113. *A Java Framework used to write and execute test cases for programs. Offers a wide range of Assertion Methods. 🡪 Analysis knowledge, design knowledge, implementation knowledge*
114. *@Test, @fail(String s), @BeforeClass, @AfterClass, @Before, @After, @Test(timeout= 100)*
115. *Unit Testing, Integration Testing, System Testing, Acceptance Testing 🡪 Unit tests individual components (class or subsystem), Integration tests groups of subsystems and their interaction, System tests the entire system in the development environment, Acceptance test the entire system in the target environment 🡪 tests should meet the requirements and criteria of the customer*
116. *Big Bang integration 🡪 All subsystems are tested at the same time, requires all subsystems to be ready for testing and needs them to be integrated  
     Bottom-up Testing Strategy* 🡪 *We start implementing, integrating and testing from the bottom layer, afterwards we continue with the layer above - Pros: No doubles, useful for integration testing of Object-oriented systems and Systems with strict performance requirements – Cons: Tests UI last, Test drivers are needed  
     Top Down: Bottom-Up in Reverse – Pros: Test cases can be defined as functional requirements, no test drivers – Cons: Doubles needed 🡪 difficult and large in number, Alternative: Test each layer individually then merge Layers 🡪 Doubles & Test drivers  
     🡪 Risk#1 High complexity 🡪 Difficult integration  
     🡪 Risk#2 Late Integration 🡪 High risk of unexpected failure  
     Vertical Integration: Slices of every Layer / Component, Scenario-driven testing and Scrum 🡪 User story driven testing – Pros: Always executable version of the system, Team members have good overview of the project status*
117. *Functional Testing, Structure Testing, Performance Testing, Acceptance Testing 🡪 more explicit requirements are easier to test, Quality of use-cases determines ease of functional testing, Quality of Subsystem decomposition determines ease of structure testing, Quality of functional requirements determines the ease of performance tests*
118. *Collaboration, Presentation, Management, Technical Writing*
119. *Communication*
120. *Communication Event: Exchange of Information with defined objectives and scope (Scheduled, Unscheduled)  
     Communication Mechanism: Tool or procedure that can be used to transmit information (Synchronous, Asynchronous)*
121. *Problem Definition, Project Review, Client Review, Walkthrough, Inspection*
122. *Request for change, Issue resolution*
123. *Smoke Signals, Hallway Conversation, Meeting, E-Mail, Newsgroup*
124. *Undertaking, limited in time, to achieve a set of goals*
125. *People grouped into departments, each addressing an activity (Function)*
126. *People are assigned a problem to be solved within a limited time and budget (Project)*
127. *People from different departments of a functional organization are assigned to work on one or more projects, members usually not always assigned to one project*
128. *Tester, System architect, Liaison; One-To-One, Many-To-Few, Many-To-“TooMany”*
129. *Authority, Responsibility, Accountability, Delegation*
130. *Work package: Description of work to be done, preconditions for starting, duration, required resources. Work products to be produced, acceptance criteria. Risks.  
     Completion Criteria: Includes acceptance criteria for work products produced by the task*
131. *Conception, Definition, Start, Steady State, Termination*
132. *Defines relationships among resources, particularly participant roles: Decision structure, reporting structure, communication structure*
133. *Stereotypes allow you to extend the vocabulary of UML, e.g. <<boundary>>, <<control>>, <<interface>>, <<extend>>*
134. *People who have a legitimate interest in the project.*
135. *Verification, equivalence check between two models, and Validation, comparison of the model with reality/client*
136. *State represents an abstraction of the attributes of a class*
137. *Greenfield, Reengineering,*
138. *Scenario, Use case, User story*
139. *Identifying the purpose (requirements and constraints), Identifying system boundary (inside outside)*