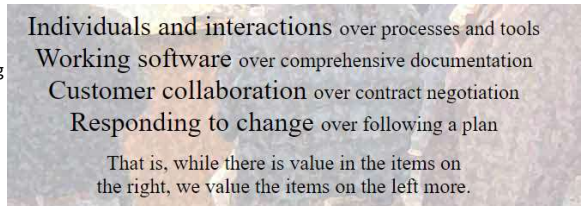
Final Study Guide

Wednesday, December 6, 2017 6:43 PM

This guide will cover all of the quiz answers and look up information from the course slides to try and erudiate the needed points for the final. I would read through the slides just to be safe.

CHAPTER 1 Enterprise And Overview:

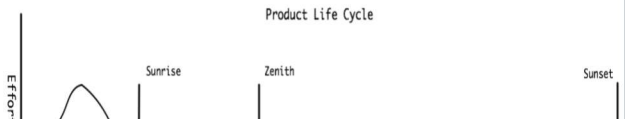
- An **enterprise system** is the software, processes, org structure, and people who interact within a business to provide business value. An alignment between business and IT allows for collaborative problem solving. Many challenges exist in this space that deal with people, processes, tech, and environments. We use **frameworks** to frame these complex problems, as well as evaluate solutions to them.
- Some of these problems include:
  - Organizational challenges, like inflexible policies, information silos, rapidly changing business needs, and a master/slave relationship between business and IT.
  - People challenges, like "Rockstar coders", not my job mentalities, lack of planning and issue exaggeration, and **cognitive illusions,** which are errors disguised emotionally as facts/memories.
  - Political challenges, like the lack of documentation on a project, and applying the right solution to the wrong problem.
- The best possible relationship between business and IT is one where business and IT work together to solve the business's problem.
- **Nonfunctional requirements** are those that are often called quality attributes, or -bilities. Stuff like security and reliability of systems.
- Traditional project management does NOT include sociology, team dynamics, and psychology. These are facets of today's PM processes. This new region is called PM-2, and the old is PM1. We explicitly factor in people and cultural influences, use rigorous software engineering principles, leverage best practices, and follow scenario/user based development. These use **continual feedback** (the use of retrospectives to reflect on how to make an agile team more effective) and **progressive elaboration** (the approach of discovering the big picture before diving down into the details over the course of a project's duration) strategies.
- The **Agile Manifesto** is a statement of four values by which developers should build their product. While managers like good work, they usually do not always support the use of agile practices within development teams. **Agile** uses little up-front requirements and design work, is driven by empirical data, has a low cost of change, has a high level of customer interaction, and uses tech support for unification. It is good for projects where requirements can change or need to be explored. It should be used most of the time.
- Software engineering is described as an immature science due to the fact that it uses mainly practitioner based research. This means that Agile and other techniques in SWE are developed in the workforce, not in a university.
- **Iterative development** is the idea that one should deliver working software frequently, from a couple of weeks to a couple of months at a time. **Incremental development** is the idea that we build features of a product a piece of functionality at a time.
- Using a framework is a good technique to use to scope down enterprise system challenges to make them better. This means you have to follow established frameworks, no ad hoc bullshit you made yourself. You use the techniques, principles, and representations from a solid framework like Agile to frame your own problems and measure your performance and solutions regarding these issues.
- In Agile, you do NOT need a detailed project plan before a sponsor commits funding.



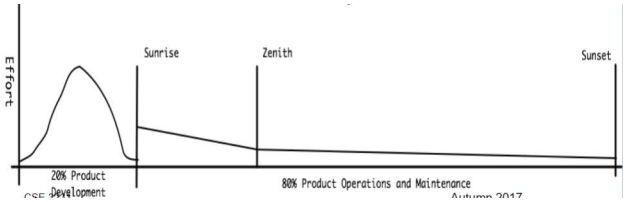
CHAPTER 2 Portfolio Management and Project Charters:

- The **value chain** is a representation of the internal operations of the company. It is represented by Porter's Value Chain Model.
- 

- The different parts of the model are:
  - Inbound Logistics, or how we manage resources coming into the organization.
  - Operations, what products and service we create.
  - Outbound Logistics, how we deliver our products to the customer.
  - Marketing and Sales, how we get the customer to buy our shit.
  - Service, how we make sure our customer doesn't throw our shit at the wall and curse our corporate name for eons to follow.
- The strategies to achieve a good position in the market have to be:
  - Defined based on Porter's Model
  - Translated into operational terms, so we know how to establish process
  - Documented and communicated in an understandable manner throughout the organization
  - Executed in a unified manner throughout the organization
  - Have that execution measured and use these metrics to continuously improve the strategy, design, and other things
- The **balanced scorecard** is a tool that ties business strategy with business processes, customers, finances, and learning and growth.
- The Portfolio Management Office takes care of business value entity management. They start projects. A portfolio is a prioritized collection of projects, both planned and ongoing. There are many types of projects:
  - **Strategic**, which meet long term objectives, at a high risk
  - **Tactica**l, which meet short term objectives at a lower risk
  - **Operational**, which support the business as a whole, used to improve infrastructure
  - **Compliance**, which give no value but are needed due to ethics, laws, and other gotchas
- The most important thing when starting a project is getting someone to sponsor and spend money on it.

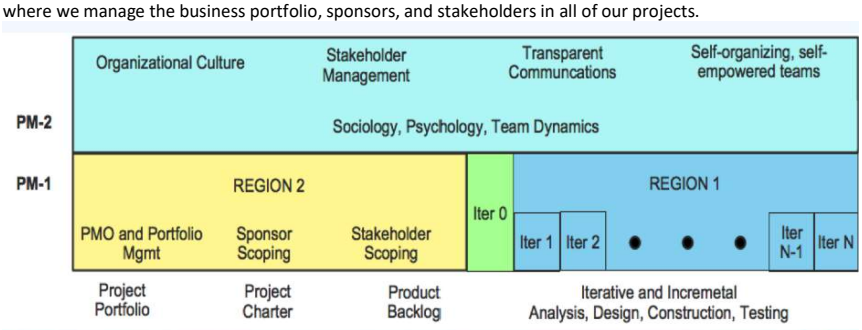


    - **Tactica**l, which meet short term objectives at a lower risk
    - **Operational**, which support the business as a whole, used to improve infrastructure
    - **Compliance**, which give no value but are needed due to ethics, laws, and other gotchas
- The most important thing when starting a project is getting someone to sponsor and spend money on it.
- The **product lifecycle** is usually 20% dev, and 80% maintenance (which costs half the project).
- The **Delphi** technique gives ratings to the business projects in the portfolio so we can easily choose
- The **project charter** is important because it authorizes the project with a sponsor and sets initial scope requirements that everyone agrees on
- We can ask for less funding on a project if partial product releases can pay for future updates.
- **Risk** is an uncertain event or condition that could have a positive or negative effect on a project's objectives. A trigger is something that causes the risk to become an issue, and a risk response plan is a planned action to minimize the impact of the risk either before or when it happens. Those plans are:
    - Avoidance, not taking risky actions
    - Mitigation, taking preventive action to minimize the probability of the risk
    - Contingency, performing an alternate action when the trigger fires off to minimize impact
    - Transfer, moving the risk to another agency
    - Acceptance, taking the consequences when the trigger fires off



CHAPTER 3 The Project Process

- The primary reason we need a software engineering process is to reduce risk. If we do not use a process, we could miss non/functional requirements, and we operate unpredictably.
- The **four benefits of a software engineering process are**:
    - Repeatability
    - Predictability
    - Traceability
    - Continuous Improvement
- A **software process** is a systematically designed method of developing and maintaining a software system through its lifecycle. There are eight phases:
    - Requirements Identification - Identifying problem to be solved, the features of your solution, the application to the business, and how we know the end result is satisfactory
    - Analysis - Understanding what context/domain we are working in, the problem we have, and the solution we propose
    - Architecture and Design - Hopefully following a thin-thread development approach, we determine the overall structure of the system that we propose as our solution
    - Implementation - Building the system
    - Testing - Making sure the system works
    - Deployment - Putting the system to work
    - Maintenance - Keep the system working
    - Project Management (This occurs at every step above, the rest occur in sequential order) - Planning, organizing, leading, controlling, and coordinating all phases of the project
- **Modern Project Management** has two regions, PM-2 which deals with people, and PM-1 which deals with projects. PM-1 is split into two regions, region 1 where typical development of a system takes place, and region 2, where we manage the business portfolio, sponsors, and stakeholders in all of our projects.



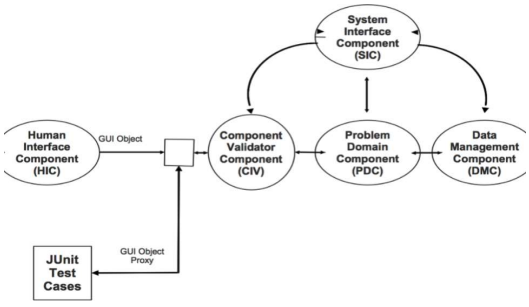
- The process for region 2 goes like so:
    - Finalize your **project charter** with the financial model and the risk plan to define how you fund your project and how to handle issues down the road
    - Hold the **business kickoff meeting** with sponsors and stakeholders, and take minutes to align agreements on project components
    - Collect **business abstracts** (a page from the stakeholder about what they expect the product to do)
    - Build the **project abstract** (a list of objectives for the product, including what it should not do)
    - Build the **project context diagram** which shows what the project is explicitly responsible for, what data flows in and out of the project, from and to whom.
    - Build a **communication plan** to indicate what each stakeholder needs and how they should get it
    - Build a **product backlog** (a list of all the features of the product that need to be made, based off of the abstract) for the high level scope of the product
    - Build a **preliminary release plan**, defining iteration lengths and applying the product backlog to it
- There are many roles involved in region 2. Those are:
    - Sponsors, who give you money and permission for projects, as well as set boundaries on scope and requirements
    - Stakeholders, who can be anyone with a vested interest in the product.
    - Product Owners, who liaison with a collection of stakeholders to represent the requirements and sponsor.
    - Project Manager, or Agile Project Manager, who liaise with business people and maintain scope, cost, quality, and stakeholder expectations, as well as facilitate the technical team during the development iterations
    - Business Analyst, who analyzes business workflows to understand business processes, as well as elicits requirements and performs analysis for correctness and completeness on those
- **Thin Thread Development** involves integration test avoidance, doing this by implementing scenarios called user stories that trace a thread through the system.

CHAPTER 4 Preparing The Project

- Region 1 contains the technical iterations used to develop the system. Special iterations like **hardening iterations** exist, where no development work is done, and code and design is refactored. The last iteration is devoted to release with hardening involved as well. Technical documents and regression test suites are written and packaged here, respectively.
- **Iteration 0** is a transition between region 2 and region 1. It is a transition from business processes to setting up technical construction and support. Development tools, dedicated team members, Trello boards are all acquired here, as well as a definition of a team working agreement (rules to work together by) and the architecture of the system.
- There are a couple of roles in region 1 that should be revealed as well here:
    - The Agile Project Manager, who reminds the team of the working agreement and agile best practices, facilitates the team during iterations, creates iteration reports, and facilitates the tech demo to the users.
    - The developer, who codes. These people are on the **critical path**, which means if they are delayed, the project is most likely delayed.
    - Business Analyst, who elicits detailed requirements, liaisons with customers for requirements changes, stays ahead of the team to figure out future work, and represents the customer/product owner.
    - Tester, who does integration and regression testing, and facilitates the **defect and change** meeting, where defects are brought to the teams attention within stuff that has been completed
- Everyone in the team estimates and commits tasks to the iteration scope, designs and codes, attends **daily standup meetings** (report work yesterday, report work today, report anything blocking you, and nothing else), fixes defects, swaps roles, and approves builds.
- **Agile** is typically incremental, iterative on the scale of 2-4 week iterations, scenario-driven with use cases, thin thread, and utilizes quick feedback, team-testing, self-empowered and self-organized teams. Examples of some Agile frameworks involve XP, SCRUM, and Kanban.
- **Pair Programming** is a technique where two programmers share the same workstation to write the same code in a statistically proven quicker way.
- A **use case** is representative of a product feature. Multiple use cases can also represent one feature. A workflow from the context diagram can define a use case, and this is usually where they are defined from. These use cases follow an **IPO Chain**, where they are input into the system, processed in some problem domain, and output as a result of the system. The best use cases comply with the IEEE 830 standard. We use these because they can be rigorously validated and are not so huge as to create vagueness. These are considered correct when the use case and the requirements model are consistent.
- Each use case can be broken down into multiple paths of execution of the use case, called **user stories**.
- Use cases are put into **Trello boards**, which help a team track and organize their work on use cases during an iteration. Each use case is assigned to a team member, sized by the team to estimate how difficult it will be to complete, and then placed in the iteration backlog, a column of all work to be done. From there, cards move to construction, then to testing, then to done. Something is done only when it passes testing with no defects.

CHAPTER 5 Architecture and Infrastructure

- **Architecture** is a framework into which to build the product. It has a high enough level of design so as not to reveal implementation details. It has a structure that reflects both the functional requirements, and non-functional requirements for the product, and needs a balance between up-front design, and low-level design. It is mainly driven by NFRs.
- The **Kruchten 4+1 Model of Architectural Views** looks at use cases in different ways to drive how the system will be designed. The views are:
    - The logical view, or the functional model of how the system fits together
    - The process view, or the flow and dynamics of a transaction through the system
    - The development view, or a static organization of the software in the system
    - The physical view, or the hardware running the system
    - The requirements view (this is the thing that glues everything together), which is based on user scenarios, and demands that all of the above views explicitly comply with what it represents
- **Architectural validation** identifies the quality attributes that the scenarios will test. These scenarios will test the normal function of your system, as well as the borderline malfunctioning cases, and your ability to grow as well.
- You will have to make many decisions during architectural validation. Some of the key ones are:
    - Your sensitivity points, or where you decide on something that fulfills an architectural requirement
    - Your trade-off points, a property of a system component that touches more than one attribute of your system
    - Your risk points, or a sensitivity point that is close to its functional limits
- Examples of **quality attributes** are:
    - Availability
    - Scalability
    - Performance
    - Security
    - Testability
    - Usability
- The **Model-View-Presenter (MVP)** model kicks MVC's ass. It promotes a strong separation of concerns, and can directly support automated GUI testing. It has a couple of components:
    - The Problem Domain Component, where business logic and data are manipulated
    - The Human Interface Component, where user-specific representation for first input and final output exist
    - The Component Interface Validator, which filters and validates all data to and from the HIC, and translates things into data the PDC can work with. This is used as a proxy hook for testing as well
    - The Data Management Component, where business and product data is stored in a database/data store
    - The System Interface Component, which connects to external entities through networks and other protocols.
- A **Test Proxy** is an object that simulates real data input into the system. It implements a specific interface of the CIV.
- A system has multiple environments. This is where **builds** live, or versions of the system that have been compiled and tested. Types of environments are:
    - The Dev Environment, where currently constructed builds are made and repaired
    - The Test environment, where finished builds are tested before approval to ensure no defects
    - The Staging environment, where the latest approved build lives before it is released



    - The Production environment, where the finished build lives and interacts with customers, totally isolated from anything related to the project environments

CHAPTER 6 Requirements

- **Software Requirements** are the needs or conditions that a new or altered software system must meet, taking into account the possibly conflicting perspectives of the various stakeholders. They tell what needs to be done. Design tells how it needs to be done. There is no one right or best design to meet a single requirement.
- Bad requirements can lead to scope creep, adding things in and ruining your planning for time. They are the largest source of defects in a product. Customers never know what they want, so they change often.
- To do requirements elaboration, you need to take your context diagram, and turn it into a UML and XUML diagram. Use the arrows from your context diagram to figure out the workflow in these new diagrams. From here, you will turn this workflow from the UML stuff into use cases, and those into user stories.
- After this is done, you define the use case catalog, listing all use case descriptions and objectives. You need to prioritize this as well, considering the value to the business, the risk of each use case, and the dependencies they have on each other.
- Every use case in the catalog has a couple of sections you need to complete:
    - Actor, the entity interacting with your system
    - UX, the UX artifact that allows the interaction
    - Pre-conditions, the things needed to be true before your use case can happen
    - Post-conditions, the things needed to be true after your use case finishes
    - Invariants, things that need to be true in the pre and post
    - Detailed Description, where you describe the steps of the use case. In each step, an actor takes an action, and the system response is detailed based on that action.
- These use cases can then be broken down into user stories and test cases using the NEBS method. NEBS stands for a set of types of user stories:
    - Normal, or a normal interaction with the system
    - Error, or an error case within the system that the system needs to respond to
    - Boundary, or a functioning path through the system that almost causes an error
    - Special, anything other than the three above
- After the catalog is done, we need to write acceptance tests, which commit our stakeholders and customers to a determining of product acceptance. All functional requirements have to pass acceptance testing.

CHAPTER 7 Requirements Analysis

- A valid requirements model has a detailed use case, a UML/class diagram, and an XUML/sequence diagram.
- An **object** is an entity with a unique identity, a single responsibility, and a long term state. These can be instantiated when it is created by the constructor of a class. Methods of the object should always be called through the object instance.
- A **class** is a specification for a set of objects of the same type. The class responsibility determines the methods in the class. A class should have low coupling and high cohesion, meaning it should not depend on others to operate, and should represent one logical idea.
- A **function** is a mapping of multiple inputs to one output. A **method** is a callable function or procedure (function with no return). The call request to start a method is called a **message**.
- Classes have relationships:
    - The IS-A relationship represents subclasses. A subclass has a parent class, and can use the methods and data from the parent class.
    - The HAS-A relationship, which represents components of a class. A composite component is a needed object for the owning class to operate. A collection component is a set of the same type of class that the owning class controls. An aggregation component is the same thing as the collection, but without the same type. Drawn as top of owned class to bottom of owning class.
    - All other relationships are associations.
- Abstract classes are parent classes that can have implementations of code in them. Interfaces are the same, but with no implementation. They both support polymorphism, reducing code replication.
- Requirements Analysis is used to understand a system. The domain/requirements model is usually the product of this process. We need to analyze our domain, how our system operates in an enterprise. We also need to analyze our goals and our solutions. This analysis will only take part for the code within the PDC section of our architecture. We do this by object oriented analysis. We use noun and verb lists to ID potential classes and methods from the requirements description, identify collaborations between these classes, and build the domain model to perform the next step, requirement validation.
- **Requirement Validation** ensures that the content is correct through validation, and ensures that the artifact is what the stakeholder wants through verification. We review our use cases to verify requirements and we use user demos to verify results of development. We use the requirements/domain model to validate requirements.
- We use orthogonality to relate functional decomposition (use cases) and scope of control (object model) decomposition.
- UML class diagrams are used to cross check detailed use cases with the object model. After this is done, use case execution is simulated with an XUML sequence diagram for each path through the use case.
- Doing validation verifies existence of the needed data, reveals missing classes and data and methods, reveals missing use case logic, and is the foundation of turning use cases into test cases with NEBS.
- Getters and Setters are evil.
- I'm not going over UML and XUML, it is its own study.