• identifying the causes of the identified vulnerabilities . **SORUCES TO GET THE INFO:** Conversations with the management, end users, IT/System administrators. Observing people at work. Recording the current task completion. Using forms that provide information on data requirements and workflows. Through flyers/brochures we can learn about our competencies that still needs to strengthen or how a company wants to be seen. **Object -Oriented Analysis. 1.** **Develop a system idea: (Objective)** is to find the fundamental objective and system idea. What should be achieved with the system to be developed. **(Action)** Developing the system idea together with the client, product recipient, user and the developer, actively clarifying conflicts of interest and contradictions. We also formulate the system idea briefly and concisely. Consider most important characteristics, features, framework conditions, prerequisites and explicit exclusions of performance. Make sure the clients, product recipients, users and developers know the system and support it without reservation – **(Artifact)** Vision-Doc. **. 2.** **Identify Stakeholders: (Objective)** Ensure that all relevant stakeholders are considered. Find out which groups of people can provide requirements for the system. **(Action)** Identifying stakeholders. Assess the importance of stakeholders based on relevance and risk by having if they “must - should – could” be considered. Identification of concrete projects contact persons (name, functions, contact data). Classifying the contact persons into expert, or those responsible for the requirements and those affected by the system on a more detailed level **Key stakeholder groups:** End Users, Specialist department, audit department, Client, Financier, Management, Board of Directors, Management. System administrators, service personnel, training personnel, hotline, support System Developer, System Maintenance. Project opponents and supporters. **There are three categories of contact persons as their identity in the project:** Domain experts. Requirements manager. People affected by the system. The quality of this categorizing is naturally decisively influenced by the contact person. Therefore, communication is with them is of curial importance. **3. Identify stakeholder’s interests**: **(Objective):** Ensure that the requirements of all stakeholders are considered. **(Action):**We describe the objectives and interests of each stakeholder. Identifying existing problems and vulnerabilities from the stakeholder perspective. Describing the important required system properties from the stakeholders’ point of view. **(Artifact)** Vision Doc. + Stakeholder request – Detailed description of individual interests. **Typical difficulties** Interest holders know what they want, but they can't express it. Interest holders don't know what they want. Interest holders think they know what they want until you give them what they want. Analysts believe they understand user problems better than the users themselves. Everyone thinks everyone else is politically motivated. **4. Collect and study material Learning from previous practices and accessing additional sources of information. (Objective):** Learning from previous practices and accessing additional sources of information. **(Action)** identification and analysis of objects, examples and patterns from the domain. Evaluation of the material with regard to relevance and usability for the current project. **(Artifacts)** A list of all collected materials **5. Create Glossary (Objective)** Creating a uniform, consistent understanding of terms and minimise misunderstandings between developers and users. **(Action)** Creation of technical glossary and definition of all important technical terms. Defining all classes of the class model as a term in the glossary. Defining all association roles as a term in the glossary. Define all other important technical subjects, concepts and states of these subjects in the glossary. Defining all important general and technical process words in the glossary. **(Artifact)** Glossary. **Structure of how to create a glossary.** Example below

|  |  |
| --- | --- |
| **Term** | **Invoice** |
| **Synonyms** | **-** |
| **Shortcut** | **-** |
| **Definition** | **Each invoice results from a contract. It invoices services rendered or deliveries and is addressed to a customer.** |
| **Delimitation** | **There are individual, monthly, partial and collective invoices. An invoice has an invoice recipient, a date, an invoice number, and invoice items that are used to list the individual services and deliveries to be billed for** |
| **Constraints** |  |
| **Contact Person** |  |
| **Status** | **Final** |
| **Changes** |  |

**Things to consider while creating this**: Use Active instead of passive formulation. Do not use synonyms, homonyms or tautologies. Use verbs instead of nouns that are not technical terms. Use terms only in justified cases in the plural. **Use Case:** A use case is a written story. It describes business processes or procedures when using a planned system at a high level of abstraction. In other words, a use case describes the interaction with a system by means of a coherent workflow. A use case is always initiated by an actor and usually leads to a visible result for the actor. **Scenario** is a specific sequence of actions and interactions between actors and the system. Each concrete path through use case represents scenario. A use case is a collection of related scenarios. **Business use case** describes a process flow on a business level independent of a technical system implementation. Triggered by a business event and typically leads to a result that represents a business value. **System use case** describes the behaviours ofa system (hardware or software) that can be perceived by external actors (users, neighbouring system). **What are useful use cases?** Indication 1: The boss test. Indication 2: Elementary Business Process Test. Indication 3: size test. SHOULD I ADD STUFF REALTED TO HOW TO IDENTIFY SYSTEM,BUSINEESS and USES CASE **User Stories:** are a short description of what your user will do when they use your software. **EXAMPLE:** As an [actor] I want [action] so that [achievement]. **Acceptance criteria** required • Which define the boundaries of a user story, • and are used to confirm when a story is completed and working as intended • written by the Product Owner (in Scrum) **A good user story should be**: • independent • Negotiable • Valuable • Estimable • Small • Testable. The three Cs of the User Story are Card, Conversation and confirmation.  
**User Story and Use Case compared**: User stories are about needs Use cases are about the behavior you’ll build into the software to meet those needs. User stories are easy for users to read Use cases describe a complete interaction between the software and users (and possibly other systems). **Advantages of User Stories** • Simple Concept • Easy to understand • Easy to communicate • Easy to plan • They seem to cause less effort to develop **Disadvantages of using User Stories**: • Lack of context (what's the largest goal) • Sense of completeness that you covered all bases relating to a goal. • No mechanism for looking ahead at upcoming work • What is the basis for regression tests? (The description in user stories is insufficient to derive test cases.)  
A LOT OF SLIDES AFTER ABOVE TOPICS SHOULD BE LOOKED AGAIN TO MAKE SURE YOU HAVE EVERYTHING IMPORTANT  
**5. DESGIN** **Defining a layer model:** The objective is to determine the basic hardware and software layers of the desired architecture. We identify the individual elements for examples, layers components, class types, etc. also we define the relationship. Describing the responsibilities, tasks and specifics are also very important. In the end describing the basic communication mechanism with the help of the interactions diagrams and document the application of design patterns. **Use Case Controller** controls the processes defined in the activity model belonging to the use case. moderates the interactions between the domain-oriented classes / components according to the defined possibilities can also control the flow of several use cases receives events coming from dialog agent and workflow control. **Component:** A component can be instantiated in a similar way to a class and encapsulates behavior. It is used to form units with a high degree of technical coherence. Internally, a component usually consists of a set of classes and defined interfaces. An essential feature of components is the focus on interchangeability. Components communicate via the most primitive protocols possible. **Typical Types of Component Interfaces:** **1.** **Factory Services:** An interface to create or load instances using domain-oriented keys. **2.** **Observer Services:** An interface that defines the monitoring and notification services. **3.** **Objective-Interface:** A domain-oriented interface. (object interface) **Frameworks** ensure a uniform architecture by predefining communication structures and providing numerous abstract classes from which concrete classes are derived. Essentially, a framework acts as an abstract implementation of the application architecture. One key feature is the shifting relationships between layers: basic interfaces are handled by the framework, and communication between layers occurs primarily through the framework’s classes. Consequently, classes based on the framework are designed to communicate mainly within their respective layers. **A** **component diagram** is used to display the system structure in components at runtime. It serves an execution-oriented view and answers the question: "What is the structure of the system and how are these structures created?" **A deployment model** involves determining which software components run on which hardware and how communication occurs. This includes identifying the hardware available at system runtime (nodes), specifying which software components should run on each hardware unit, and determining the technical methods for communication between these units. This configuration is then described using a deployment diagram. **Interface**  
A LOT OF SLIDES AFTER ABOVE TOPICS SHOULD BE LOOKED AGAIN TO MAKE SURE YOU HAVE EVERYTHING IMPORTANT.  
**6. Object Design using GRASP: In Responsibility-Driven Design (RDD)**, we understand software objects as something that has responsibilities, where responsibility is (should be) a generic term for what these objects do. **A design pattern** describes a proven template for a design problem. It is a reusable template for problem solving. **GRASP is an acronym for General Responsibility Assignment Software Patterns**. It is a set of 9 basic OO design principles in pattern form. It was invented or compiled by Craig Larman primarily. It serves as a learning aid for object design. In contrast to typical design patterns, the proposed solutions in GRASP are far less concrete. We therefore refer to the 9 GRASP principles below. The names of the principles are: Low Coupling (First Target), High Cohesion (Second Target), Information Expert (Advice), Creator (Advice), Controller (Advice), Polymorphism (Advice), Pure Fabrication (Advice), Indirection (Advice), Protected Variations (Advice). **Details: Low Coupling** aims to limit the impact of changes and enhance the comprehensibility and reusability of software. The solution involves assigning responsibilities in a way thatminimizes couplings

**Disadvantages**: • Lack of a clean requirements definition • Relatively tight quality management • Lack of alignment of development activities with the overall system • High demands on the qualification, discipline and flexibility of the development team • Requires the development team to work in the same place at the same time due to the high communication ratio • Drafting of contract customer / SW-developer problematic. **SCRUM PROCESS:** Scrum is an agile based software mode. It does not have phases like OpenUP, but its has more like a technical milestone with respect to how a sprint ends with a small release every time a sprint ends. In the agile Scrum world, instead of providing complete, detailed descriptions of how everything is to be done on a project, much of it is left up to the team. As an overview, Scrum is a lightweight process, simple to understand but difficult to master. The procedure is defined with the following few roles in the whole process: Team, Scrum Master, Product Owner. The process is organized in “Sprints”. As a summary the process values on commitment, courage, focus, openness and respect. **Artifacts: Product Backlog**: Consists of User stories which define the requirements, so the product backlog defines all the requirements/features of the project, owned by the PO. **Sprint Backlog**: A small amount of user stories that we want to implement, owned by the team and the Scrum Master. Sprint Planning: Here the product owner with the team talks about the priority of the user stories for the upcoming sprints to be implemented. **Daily Scrum**: The whole team comes up together where each member tells • What i did before • What i am doing • What i will do. **Scrum Review Meeting**: SM hosts. We see the current demos of the project, with the project owner in the team, to make sure that the requirements are met. **Scrum Retrospective Meeting**: SM hosts: Team also attends (Usually the PO is not there, so the team can freely talk about the process and how can they improve themselves, their work/performance). **Rare Case (Sprint Cancellation):** Very rare but in a case if Sprint Goal cannot be met by Team, for example, unsolvable impediments, drastic environmental changes and severe misestimation. **MORE-ARTIFACTS**: **Product Backlog Sprint Goal. Sprint Backlog**: • List of the technical tasks per product backlog item • Owned by the team, status and estimates updated daily • Only Team modifies it (PO must not change the scope) **Blocks List** • Impediments, blocks, pending decisions • Owned by SM, updated daily **Product Burndown** Chart • Visualizes overall progress (estimated remaining efforts) and team velocity • Updated after each sprint. **Sprint Burndown Chart** •Visualizes sprint progress (estimated remaining time) • Helps detect problems in sprint • Updated daily **Definition of Done** (DoD) • List of quality criteria applicable to all requirements • Mutually accepted by Team and Product Owner **Product Increment:** • Potentially shippable product version • DOD-complaint (tested, documented etc.) • Delivered once per sprint **Information Radiators — Additional Issues are defined here — More Details:** • Scrum and other artifacts (e.g. architecture diagrams) • Easily accessible for Stakeholders and other interested parties • Big posters/monitors in public areas or project wiki • Purpose: provide max transparency about project **Advantages** • Lightweights - like xp • Copes well with rapidly changing requirements - like xp • Scrum, being agile, adopts feedback from customers and stakeholders – Like by having a executable version sot that the customer can give good feedbacks • Short sprints enables changes based on feedback a lot more easily • The individual effort of each member is visible during scrum meetings **Disadvantages** • Scrum often leads to scope creep (Means taking the tasks from a sprint to another sprint because of lack of resources – Happens in student projects), due to the lack of a definite end-date • The chances of project failure are high if individuals aren’t very committed or cooperatives • Adopting the scrum framework in large teams is challenging —- That's why you can solve this problem by creating sub teams to make the process more effective. • The framework can be successful only with experienced team members • If any team member leaves in the middle of a project, it can have a huge negative impact on the project • Quality is hard to implement, until the team goes through aggressive testing process. **3. Requirements**: A requirement is a statement about a property to be fulfilled or the performance of a product, a process or the persons involved in the process.  
**Requirements Management**  
Comprehensive, systematic approach to identifying, documenting, organizing and tracking requirements  
**Functional requirements** Functional Requirements describe the domain-oriented functionalities or services that the system is supposed to provide from the user’s point of view. These requirements are specific to the type of system being developed and its intended users. It focuses on what the system basically should be able to do. It captures the specific behaviours or functions of the system. It directly impacts the operations performed by the system. Examples might include user authentication, data processing, reporting, and other specific actions that the system performs. **Non-Functional Requirement:** They describe all other required characteristics of the system that do not directly affect the specific function but are crucial for the system’s overall performance and usability. The key points are that it focuses on how the system performs its functions. Often refers to important system attributes that affect the system. Do not define specific behaviours but rather the qualities or constraints of the system. Examples: User-friendliness: How easy the system is to use. Trustworthiness**:** The system’s reliability and security. Efficiency: Performance metrics such as response time and resource utilization. Maintainability; How easy it is to update and fix the system. **Types of Non-Functional-Requirements: Operational Requirements** (Describe interfaces, data, functionalities and reactions of the system to events) also extraordinary events -> exceptions. Operational requirements are also referred to as a business concept. **Quality requirements:** concern software quality criteria such as reliability, maintainability, efficiency and usability. Should be specified quantitatively if possible (-> Verifiability). **Technical Requirements:** include constraints such as devices to be connected, interfaces to external systems and the use of development tools.  
**Validity and maintenance requirements:** here we prepare tests (e.g.by specifying test cases). Defining the acceptance test. Describing the scope of warranty conditions, maintenance conditions, training,.. **Implementation requirements** concern the process model and the documentation. The resources available (personnel, date, costs). Additional conditions such as legal regulations, (company-internal) guidelines and standards. **Requirements Analysis according to the Unified Process**: A systematic approach to finding, documenting, organizing and controlling changes in the requirements of a system. • So basically, the requirements are recorded iteratively and further developed. • Its NOT considered as A PHASE but a discipline that lasts the whole project (with emphasis on inception and elaboration). • Requirements are developed in workshops in cooperations with customers, application experts, businesspeople and technicians. • A list of documents is proposed. Functional requirements are specified with the help of use cases • The challenge is the requirements are neither obvious nor unchangeable over time • An incorrect level of detail in the formulation of requirements eliminates the flexibility for design decisions •   
**Artifacts of the UP requirements analysis** • **Stakeholder requests** (Describes all stakeholders, identifies their wishes and assesses relevance/risk for the project) • **Vision Doc**. (Describes in general terms the objectives and conditions, the business plan; executive summary) • **Business Case** (Describes the objectives, conditions, and business plan of the project. It includes an executive summary and outlines the rationale, benefits, and overall strategic importance of the project.) • **Glossary** (The main terms of the domain and Data Dictionary) • **Risk List** (Describes risks (business, technical, resource, and schedule related) and ideas for management / mitigation.) • **Supplementary specifications** (Describes non-functional requirements) • **Use-Case Model** (Describes functional requirements) • **Analysis Model** (represents the relevant business issues (use cases, domain model, system process model). **Requirements Analysis Methods: Actual State Analysis: ACTIVITES** • Capturing of the user/user environments (structural and process organization, service regulations, decrees, laws, guidelines and similar more). • Capturing existing data programs • Capturing existing IT equipment • Capturing of time and quantity structure (current processing times, turn around times , waiting times, data quantities) • Capturing of business and technical factors that cannot be influenced • Presenting the threat and the equipment gap • Identification of weak points

Words, it can also be known as the phase that defines feasibility. Moreover, we are also roughly editable the time and the cost this project would require. So, the target is to decide whether to execute or cancel the project. **Elaboration: (CHECK PAGE 47- MAYBE SOEMTHINGS NEEDS TO BE ADDED FROM THERE)**This phase defines the core architecture of the project, most of the requirements are attempted to be identified here. Sometimes we also might have to refine the vision that was created before, due to some limitations or change in the requirements. We can also have multiple iterations in this phase to finally have our final version of the core architecture. This phase has more realistic estimations, because the requirements are getting defined. We also analyse through simulation, benchmarking the possible projects risks that could occur. Furthermore, we need to have a deep understanding of the system that must be developed, so we need to know the scope, its functionality, non-functional requirements (quality, reliability, etc) and performance requirements. **Constructions**. In this phase, we develop the core architecture from the elaboration phase. This phase also has a lot of iterations. Once the code is implemented and tested according to the requirements, we integrate it to the main system. During the implementation, parallel work on the work on the components is possible to shorten the project durations. The target of this is to achieve a beta version of the system, which means that the software is mature enough to deliver to the customer. **Transition:** In this phase we put the system into the operation at the end user site, which means the target is to release the product. This phase also allows as experience the following things: • User documentation which includes the user manual • Training of User can also be expected • Support of user (e.g. through a support centre) • Quick reaction to user problems. **The whole Procedure: Procedure:** The slide presents the Rational Unified Process (RUP), a developmental approach structured into four main phases: Inception, Elaboration, Construction, and Transition. The process is iterative, with each phase consisting of multiple iterations that allow for incremental development and refinement. In the Inception phase, the project's basic viability and scope are established. The Elaboration phase focuses on defining the project's architecture and detailed planning. During Construction, the actual product is developed through repeated iterations, each ending with a minor release of a stable executable part of the final product. The Transition phase is the final stage, where the product is fine-tuned through further iterations until the final production release is ready for deployment to the end-users. Throughout this process, milestones serve as checkpoints for significant decision-making and evaluation, ensuring the project aligns with its objectives at every stage. **Coreflows/Workflows/ Discipline (9 core)** **Workflows:** **Business Modelling** • Use cases of business processes (Business Cases) are defined • Target: Ensure a common understanding between system developers and users of the project **Requirements** • What the system must do • Document Functional or Non-Functional requirements + general conditions **Analysis & Design** • How should the system be implemented? • Design model (System architecture) and optionally create an analysis model, to achieve a better understanding for the tasks that the developers must do, which makes the software development process more efficient and productive. • The design should reflect with the Functional or Non-Functional requirements + general conditions **Implementation** • Develop and integrate individual modules (Classes) • Components can also be developed in parallel, in order to shorten the time of the implementation process, those components can also be further used **Test** Interaction of components and check compliance with requirements defined for the system **Deployment** All activities that here are dealt with training and user support, the compilation and installation of the software, if required, we also address beta tests and integration/migration problems when replacing legacy system. **Supporting Workflows to help the Software Development Process (RUP)** - **Project management** The main task of this workflow is to help and assist the team to complete goals, managing risk and overcoming framework conditions in order to develop a product that meets the needs of the client (those who give the money) and the user **Configuration & Change Management**: • Management of artifacts created in the project (documents, source code,...) and their assignment to certain releases. • Usually produced jointly by several persons **Target**: Ensuring consistency, Possibility of recourse to older versions, Notification of affected persons in case of changes in documents, further: Management of change requests **Environment** • Provision (Installing) of the software development environment. e.g. IDE. • Provision of hardware and software • Appropriate support of the project staff. **Artifacts** Stakeholder Request, Vision Doc. , Business Case, Risk List, Glossary, Software Development Plan, Deployment Plan, Software Architecture Document, Implementation Model, Design Model, Analysis Model, Test Plan, Use-Case Model, Supplementary Specification. **Advantages**: • Process adaptable to individual needs • Has a focus on key best practices • THE UML process • 2-dimensional process that can depict overlapping activities and phases. • Covers the entire life cycle • Supports object-oriented development and reuse. **Disadvantages**: •complex • Less suitable for small projects • Commercial process • Requires computer support. **Agile Process Model eXtreme Programming**: There are no phase.in this model but there are iterations that are similar to the concepts of sprints, which are designed to be at the same length. The major idea of extreme programming is to concentrate on implementation activities, it is important to know that in this process change costs grow logarithmically at most over time but still this process can tolerate the changing with the requirements over time. **Procedure**: • The development and the delivery of the very small subsystems/releases (1-2 months). • Planning of the individual releases • Iterative programming • Acceptance tests for the individual releases • Strong involvement of the customer in the process • Continuous improvement of the code **YOU SKIPPED PAGE 64**   
**Practices: Small Releases**: • Release cycle of one to three months • Release consists of several iterations of one to three weeks • Iterations break down into work packages of one to three days **Planning Game**: • Requirements are created by the customer in the form of so-called user stories. • Customer priorities stories • Customer determines which stories should be implemented in the upcoming iteration. • Developers specify the effort they expect for the realization of a particular features **Automatic Testing** • Besides the user stories, test cases represent the only definition of the desired functionality • A program feature for which stories should be implemented in the upcoming iteration, which means that for most of the functionalities/code we have test cases. • Developers specify the effort they expect for the realization. • Every time a programmer makes a change to the system, he is obliged to run all existing tests again. **System Metaphor** Instead of design activities XP provides the use of so-called metaphors construct replaces a software architecture System metaphor should be understandable for both developers and customers. **Simple Design**: • A change in the internal structure of software to make it easier to understand and change without changing its observable behaviour. • Simplest the design of an existing system while retaining the semantics. • Refactorization aims to improve the comprehensibility and modifiability of the code **Documentation through self-explanatory program code**: • Corresponding program structure. • Meaningful naming. • If a program needs an explanatory comment at one point, it should be refactored. • Reaction to the problem that additional documentation becomes obsolete very quickly due to lack of ongoing adaptation and is therefore unusable of misleading **Programming in pairs:** • The first partner takes care of the current coding. • The second partner checks the code for typos and logical errors and develops strategies for further implementations. • Paring is dynamic: New partners are sought for each work packages **Common code property:** Consequence: Each developer pair may make changes anywhere at any time **Continuous Integration:** Newly developed or modified program parts are to be reintegrated into the current code base after only a few hours. • For this purpose. An integration and test system are used, on which all test cases must be checked again after the changes have been imported. **40-hour week:** Because development in Paris places high demands on the concentration and performance of employees. **On-site Customer:** Should represent the view of the customer, especially in connection with the business games. **Coding Standard** Ensure uniform code that can be understood and modified by all the developers. **Advantages**: Lightweight, opposite of bureaucratic. Copes well with rapidly changing requirements

system functions. **Evolutionary prototyping** is a development approach where an initial version of the system, which is functional but not complete, is built and continuously refined through successive iterations based on user feedback and requirement changes until it evolves into the fully functional final product. Unlike **incremental software development**, where each increment is a fully functional and potentially shippable version of the product that adds specific functionalities in a step-by-step manner, evolutionary prototyping may involve building a base version of the system that has limited functionality and is then expanded and refined over time to meet the complete set of user needs and features. **Iterative Software Development: Idea** is that it tries to combine the strength of other models: Phase Model: Easy to manage, more flexible than other models. • Guiding principle: Risk mitigation Start on a small scale Keep the spiral as tight as possible Achieve the development goals with minimum costs. **Procedure**: • Dismantling of system development into intermediate products/partial products. • Development of each intermediate / sub-product in 4 recurring steps: **->** Determination of objectives, alternatives and framework conditions **->** Evaluation of alternatives and reduction of risks **->** Realization of the intermediate product/subproduct and inspection **->** Project continuation planning • Cyclical through these steps for all intermediate products/partial products. **4 steps in each cycle:** **1. Planning 2. Risk analysis 3. Engineering 4. Evaluation** In the **3rd step** often coupling with other process models: Waterfall model Evolutionary prototyping. **Advantages**: • Periodic review of the process depending on the risks • Process model is not defined for the entire development • Integration of other process models as special cases • Errors and unsuitable alternatives are eliminated at an early stage • Flexible model • Development can be redirected much more easily if insights requires it • Supports the reuse of software by considering alternatives. **Disadvantages**: • High management costs • Less suitable for small and medium-sized projects • Knowledge of risk identification and management is not yet sufficiently widespread. **Rational Unified Process: Idea:** Its a process that can be adapted and extended. The main causes of software development problems are addressed by 6 best practices. The RUP focuses on these 5 best practices which are, Iterative Software Development, requirements management, Use of component-based software architectures, Visual Software Modelling, Software quality testing, Controlled change management. **NOW EXPLAINATION:**  
**Iterative (evolutionary) Software Development.** As large projects cannot be developed in one step. Increasing understanding of the problem area during the project. The focus of consideration: Reduction of the project risk -> Division into manageable sections, each of which leads to an executable version. **Requirements Management (with Use Cases)** Here the development, organization and documentation of the required functionality and the general conditions are to be observed. The approach is by creating use cases from UML, in order to process the modelling of the business transactions by the system. The use cases serve as a reference point for quality assurance, this practice also gives an check whether all the requirements are covered in the system design. **Use of component-based software architectures:** Objective is to support the use of prefabricated components, that is why it is advised to have an early development of a system architecture. The advantages of this practice are isolated development and testing components is quite possible. Reusability in later projects. Semi-finished products (CORBA/EJB) can be used. **Visual Software Modelling:** The goal is to promote readability and clarity of documents. For this it is essential to create graphical notations for the documents. UML diagrams can be used as they are generally known in the world of software development. **Software quality testing**: Integration of QA activities take place here in all sub activities. Important thing to know is, quality assurance is not a subordinate matter, but an integral part of the process. **Controlled change management:** This practice makes changes to requirements, documents, programs, etc. Traceable and enable access to current and past documents. **Process Structure**: 2-dimensional: 1st dimension: dynamic aspects, cycle, phase, iteration, milestone 2nd dimension: static aspects, components, activities, artefacts **Why** **the Rational Unified Process (RUP) is two-dimensional** because it incorporates time-phased project milestones (Inception, Elaboration, Construction, Transition) and concurrent development activities (such as requirements, design, implementation, testing). This allows for activities to overlap and be revisited as needed throughout the development lifecycle. E.g. • During the analysis implementation activity during the realization of a prototype. • Analysis activity during the implementation phase to clarify open analysis questions that have only now become clear. **Concepts of the cycles in RUP**: In RUP, the development process is iterative, meaning it cycles through four distinct phases—Inception, Elaboration, Construction, and Transition—multiple times, with each cycle yielding a more advanced version of the software. Each complete pass through these phases results in a 'generation' of the software. After the first generation is complete, the process begins a new with Inception for the second generation, cycling through each phase again. These cycles allow the software to evolve with improved features and refinements from one generation to the next. **Milestones in RUP**: Once a phase gets fully completed then we have a milestone as an output but inside of the phase we have iterations, where at every iteration we don’t get milestones as an intermediate result, but the intermediate steps are just creating a prove that communication exists within the team. The goal of each of the iteration is to develop a stable executable subset of the final product. **Management Milestones** : Inception: LCO = Lifecycle Objectives Elaboration: LCA = Life Cycle Architecture Construction: IOC = Initial Operational Capabilities Transition: PR = Product Release.  
**The effort and time distribution across different phases of the Rational Unified Process in a medium-sized project**. Inception involves about 5% effort and 10% time, while elaboration takes 20% effort and 30% time. Construction requires the most resources with 65% effort and 50% time, emphasizing the need for clear requirements. Transition has 10% each for effort and time, possibly for final adjustments. Notes on the graph indicate the importance of precise planning and adjustments for frameworks during transition. **PHASES EXPLINATION: Inception:** This phase allows us to prepare an overview of the project by developing the vision, the business cases, we check if this project is even feasible to develop, in other

**Implementation:** • **Input of phase** is the output of the last phase, so once when we have enough Details from the previous phase, we can finally start the implementation process, which are the software specification • **Activities:** • Conversion of specifications into programs **->** Implementation of components according to specification, Systematic testing of components/modules **Result(s) of the Phase:** Software components, documented , tested. The result of this phase is the different components of the Software, documentation and tests. **Test** : **Input:** Tested and documented components / modules**. Activities:** • Integration tests, System Test and acceptance test Integration tests are used to test the interaction points of the Components System test (tests the system against the specification -> Verification) Acceptance test (tests the system in the production Environment -> Validation. **Results of the phase**: System running in the production environment. **What is verification?** Verifications means if the software is correctly developed, in order words it means the conformity between a software product and its specification, so one can question, is the product being developed correctly. **What is validation?** Validation means if the product is rightly developed which simply means that if the software meets the customers requirements, in order words, it means suitability of a product in relation to its intended use, so one can question, is the right product being developed? **Operation and Maintenance: Input** to this phase is the accepted system which is already on the production environment. **Activities** of this phase are administration, bug fixing, optimizing, adaptation to the other system environments and the changes and enhancements to functionality. **Results** of this the system on the production environment **Advantages:** • Clearly defined tasks per phase. • Easy to understand. • Many developers are familiar with this procedure. • Management seems to be comparatively simple, low overhead. **Disadvantages:** • Taking this into account, the idea that we cannot go back to the previous phases once it's fully completed, which in fact is one of the biggest disadvantages of this model, as realistically it's not possible to know all the requirements at the start of the project. • Another disadvantage is the due to lack of communication with the customer, as in this model we will typically meet the customer at the analysis and definition stage and then later on once we are at the Operation/Maintenance we will have to meet him again naturally to show him the final product but the issue due to a big communication gap can lead to these issues that the customer could not even remember what he really wanted or his demands entirely got changed. • Coding starts very late which leads to system integration related issues. Although this model can still be used, but with different variants of it. **Waterfall Model Variants** Another variant of waterfall is possibly combined with the creation of a throwaway prototype. In this prototype we have a validation check by the end of every phase to make sure that we fulfill the customers requirements. Also, we can go back to each phase depending upon our work and need. At last, this time we will be able to meet the customer not just in the system requirements phase but also in the software requirements and operations and maintenance phase. V-Model is a process of software development methodology that represents an extension or variation of the waterfall model, emphasizing the validation and verification phases. The V-Model is sometimes referred to as the "Validation and Verification Model.". It is subgrouped: • System creation (SE), • Quality Assurance (QA), • Configuration Management (KM), • Project Management (PM), The whole idea is that while the development activity is proceeding, at the same time the Quality assurance activities can work in parallel. **The "Acceptance Test" in the context of the V-Model means the following**: The purpose of this testing phase is to evaluate the system against the original requirements and needs of the users to ensure that it fulfills them appropriately which simply means that it meets the business requirements. **Advantages**: • Very well worked out and comprehensive • Generic process model with defined tailoring options • Enables standardized processing • Integration representation of 1. System Creation 2. Quality Assurance 3. Configuration Management 4. Project Management. **Vulnerabilities**: • Intended for large embedded systems. • For small projects, in some cases disproportionate bureaucratic effort. • Maintenance phase not considered. • Heavyweight process. **Evolutionary Prototyping:** A throwaway prototype, also known as a rapid prototype, is a type of prototyping that involves creating a model of a system or application with the intention of discarding it after it has served its purpose of collecting and analysing user feedback. So the main **Idea** is that, instead of a throwaway prototype, a series of prototypes is developed that converges directly into the production version of the software. Continuous improvement of prototypes. **Procedure**: • Develop a first, functionally very limited implementation • Involve customer to comment prototypes • Developnext prototypes based on customer feedback, etc. **Advantages**: • In case of unclear / unknown requirements or missing data the customer can develop his ideas of the target system step by step based on the prototypes. • The customer has a prototype available very early → important for acceptance by end users • The development risk will be significantly reduced. • Very well suited as a supplement to other process models. **Disadvantages and problem areas**: • Documentation: Prototype is considered as a replacement for missing documentation • Maintenance problems: worsening of the system structure due to constant changes. • Management: Covering up a code and fix procedure? Contract: On what basis is the service provided? • Performance: The limitations and limits of prototypes are often unknown to the customer. **Incremental Software Development.** • Stepwise extension of the functionality of the system by increments. • Development in stages of expansion. • The process is very similar to evolutionary prototyping, but the generated software artifacts have no prototypical character but can be used productively. **Procedure**: • Starting point is a small, manageable core • Further functions will be developed gradually (successively or in parallel) • When another increment is finished, it is added to the core. **Advantages**: • Usable product in short cycles → early use! • Experiences from the usage can be brought into the further development at an early stage. •Incremental development leads to manageable sub-projects. **Disadvantages**: • The originally selected software architecture may not be sufficient and may have to be completely revised at a relatively late point in time. • Only useful for relatively independent

or send events (as regular state diagrams can) • Transitions can have preconditions and postconditions shown in square brackets []. You draw your protocol state machine as a group of substates within one large frame. You must name the protocol state machine as such; place the keyword protocol in curly brackets {} next to the name. **Sequence diagram**: Purpose: Scenario oriented Shows objects involved in the scenario from left to right Shows sequence of messages in a sequence from top to bottom Allows specification of runtime scenarios in a graphical manner  
Object Constraint Language (OCL)  
**2. Systems development life cycle (SDLC) Life-Cycle Phases Initiation:**  The customer/sponsor with the software company identifies the opportunity concepts that the software can bring along. They talk about the possibilities of the successfulness of the creation of this software and also what could be the possible future of it. Typically, stakeholders create a vision document for defining the further processes. **System Concept Development:** Here we cover the scope of the system boundaries where we also document that. Moreover, we analyse the cost benefits and plan the Risk Management Plan which means the risk list which covers the risks that the software development life cycle face along the development process. **Planning:** Develops a project management plan and other planning documents which provides the basis for gaining the resources that are needed to achieve a solution. **Requirements Analysis:** The analyst with the team develops the user requirements to have an attempt to fix the system. In this phase they gather as many user requirements as possible for the development team to understand the tasks. Additionally, this phase achieves functional and nonfunctional requirements related documents. **Design**: In this phase, the architect translates the user requirements into a system design document, that document focuses on the details that the for example the skilled programmer needs to develop the software with sufficient additional input design, like an architecture notebook that can include the class domain diagrams, architectural layouts, database schemas, etc. **Development:** Converts a design into a complete information system, which means installing environments or libraries, database prepare statements, coding, test cases, refining programs, review of the tests and procurements activities. **Integration and Test:** This phase makes sure that the developed system follows the functional requirements that were set up previously. Typically, this phase is conducted by a quality assurance officer. The outcome of this phase creates test analysis reports. **Implementation (Deployment- more accurate to say):** We include the implementation that was successfully tested before to be pushed on the production environment, which means we primarily already have to know which piece of software should be now part of the existing software on the production environment. Furthermore, it's a good practice to document what we implemented, so if new members in the team join, it would be easier for them to understand what the project is about. **Operations & Maintenance**: This phase generally lasts longer than the pre-development phase, because there could be further changes and enhancements that have to be included in the production environment, also this includes post-implementation and in-process reviews. **Disposition**: Describes end-of-system activities, emphasis is given proper preparation of data. So, if an existing system on the Production environment has to be shut down. This phase then is taken into consideration**. Purpose of the Software Development Process:** • Structuring the process of software development in software projects • "Engineering" approach to software development Division into well-defined activities, which are to be processed with predefined methods. Providing a structured set of activities required to develop a software system. **Central goals of a process model:** • Supporting the team members to perform the key activities • Specification of the artifacts that will be developed through out the system development • Unify the tasks of individual developers with those of the entire team, which would make the software development process smoother. As a results it would eliminate any dependencies related issues that are experienced by the developers (For example: Experience using Scrum Daily) • Project managers are able to monitor and evaluate products and the activities that's going on the software development process; **Common points that all of the software processes have in them** • Software Specification: This means that the functions of the software and the restrictions must be defined, in order words its called requirements. • Software design and implementation: This concept brings more details to the software process, as before things were considered at a high level. As a result, this helps the project stay accordingly towards the requirements of the software. • Software validation: Most of the software development process ensures that the software does what the customer requires. • Evolution: The software must evolve in order to keep pace with changing boundary conditions: **MODELS: CODE AND FIX MODEL:**Unsystematic approach This approach is very unsystematic, and it cannot be even considered as a process model, because of lack of design, documentation and plan, which leads to unexpected results. This approach makes poor task performance due to insufficient requirements analysis. **CLASSIC WATERFALL MODEL.** This process consists of different isolated phases. There are typically 5 phases in this early model; Analysis and Definition, Design, Implementation, Integration and Test, Operation/Maintenance  
**Phases of waterfall model: Analysis and Definition:** In this phase we define and analyse the goals of the project. Moving on we also define the product requirements. We identify the weak points**.  
Result of this phase:** We get the specific requirements of the project. This phase can bring the following artifacts as an outcome: • Functional Scope • Specification of user interface • Interfaces to the system environment • Required performance of the system Intended hardware and, if applicable, system software • Documentation guidelines • Effort and schedule planning. **Design**: From the previous phase we take requirements specifications as an input quantity for this phase the activities involved in this phase are the following: • Develop and internal structure of the software system. **a)** Functional design —> Application model. **b)** Technical design: **i)** Development of the software architecture - System Design. **ii)** Dividing into components/modules (interface specifications). **iii)** Specification of components/modules and interaction. • **Result of this phase**: brings a clear picture of how the software must be developed, so we are now able to get detailed specifications of the software, this is also an artifact

**1. Software Engineering** is the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing and documentation of software. **Objective of software engineering**: ensuring the required product can be put into operation: • on time, • cost-efficient and • (as far as possible) error free and to ensure that it fulfills its purpose. The product must be developed in such a way that • operation, • maintenance and • further development of the product   
**Software crisis** meansthe technical artistic character of software development, the unacceptably high maintenance effort for software and the associated high maintenance costs, it can also mean the overall quality of the software product does not meet the expectations of the users, so that many projects fail. The term "software crisis" is coined in the 2nd half/end of the 1960s, when for the first time the cost of software exceeds the cost of hardware. It can occur in various forms, an appearance that accompanies us all the time, expressing that the expenditure to be made for software production and operation exceeds or will soon exceed the available forces for this. The main causes of the so-called software crisis are: • Software is becoming increasingly complex. • Software is prone to errors. • Software is increasingly needed (independently or as part of a more comprehensive product).   
**FURPS+** **Functionality**: Veracity: The software must produce accurate and correct results. Appropriateness: The software should meet the specific needs of its users. Interoperability: The ability of the software to work with other systems or products without special effort from the user. • **Usability** - Comprehensibility: Users should easily understand how to use the software. Learnability: The software should be easy to learn for new users. Usability: General user-friendliness, ensuring the software is easy and pleasant to use. Documentary: Availability of helpful documentation for users. • **Reliability** Reliability / Trustworthiness: The software should perform consistently and predictably. Access Control: Proper mechanisms to restrict access to authorized users. Operational Safety /Robustness: The software should operate safely under predefined conditions. Fault Tolerance: The ability of the software to continue operating properly in the event of a failure. Recoverability: The software should be able to recover quickly from failures.• **Performance**: Responsiveness: The speed at which the software responds to user inputs. Processing Time: The time the software takes to process inputs and produce outputs. Memory Utilisation: Efficient use of memory resources. • **Supportability**: Further Development and Adaptation: Ease of making future enhancements and adaptations to the software. Portability and Compatibility: The ability of the software to run on various platforms and systems.   
**The “+” in FURPS+** **•** Design constraints - Do things like I/O devices or DBMS constrain how the software must be built? • Implementation requirements: Do the programmers need to adhere to standards? Is the use of TDD required? Is statistically sound testing in the context of Cleanroom required? • Interface requirements - What downstream feeds must be created? What other systems must this one interface with? How frequent are feeds produced? • Physical requirements - What hardware must the system be deployable on? Must we be able to deploy to a machine no larger than 12" square, to be stationed in the Iraqi desert?   
**The Unified Modelling Language (UML)** 4+1 view model divides the architecture of a system into five interrelated views to address different stakeholder concerns and provide a comprehensive understanding of the system: **Logical View:** Focuses on the static structure of the software using UML diagrams like class, object, package, composite structure, and state machine diagrams. It addresses how the software will be developed. **Implementation View:** Concerned with the organization of the software components. It includes component diagrams and helps developers understand the code structure. **Process View:** Deals with the dynamic aspects of the system, showing how processes interact. It includes sequence, communication, activity, timing, and interaction overview diagrams. This view shows how components interact within the logical view. **Deployment View:** Focuses on the physical deployment of the system, detailing how the software is distributed across hardware resources. It uses deployment diagrams to map software onto hardware. **Use Case View**: Central to the model, it describes the functionality of the system from the user's perspective. It includes use case and activity diagrams. Customers can only see this view, which illustrates the use cases and user activities. **HOWEVER,** the UML does not guarantee in any way higher quality or shorter development times, its not a replacement for any programming languages  
**Use Case Diagram**: Purpose: It is used an early stage of a project. It is used to show user interactions with a system. It only summarizes some of the relationships between use cases, actors, and systems. It does not show the order in which steps are preformed to achieve the goals of each use case. It is an effective technique for communicating system behaviour in the user's terms by specifying all externally visible system behaviour. **Extend** use case defines optional behaviour, whereas **include** use case is integral part of the including use case. purpose of these two things are to simplify large use case by splitting it into several use cases, also to extract common parts of the behaviours of two or more use cases. **Activity Diagram:** Purpose: Activity diagrams may be used for various purposes: Analysing and depicting processes, documenting workflows, Showing the algorithms in a graphical way, Modelling use case steps. Modelling behaviour aspects of software – methods, services. **Class diagram**: Purpose: Class diagrams may be used in various scenarios: Static structure design and analysis System responsibilities modelling Software reverse engineering Source code generation and scaffolding. **State Machine diagram**: Purpose: There are two types of state machine diagrams: **Behavioural state** machine: specifies the behaviour of a model element. In this type of State Machine diagram: • it is an event driven • transitions originating from a state are triggered by relevant events specified by transition • different transitions from the same state should not be able to be triggered by the same event (otherwise they are not unique) • given sequence of events implies sequence of states, where "on the way" arbitrary behaviour can be executed • on execution the system is either in a state or in a transition, alternately. **Protocol State Machine Diagram** is a specialization of behavioural state machine It specifies the allowed usage of the behavioural features of a classifier • A few special rules apply for protocol state machines : States can have names but can’t show entry actions, exit actions, internal actions, or do activities • Transitions show operations but not actions

-0,2 cm

**public class C {**

**public static void doIt(A obj) {**

obj.m1();

obj.m2();

**}  
public static void main(String[] args) {**

B objB = new B();

C.doIt(objB); **}  
}**

**The Interface Segregation Principle (ISP)** emphasizes that having multiple, client-specific interfaces is preferable to a single, general-purpose interface. Clients should not be forced to depend on interfaces they do not use. Fat interfaces should be avoided as they can be broken down into distinct method groups serving different clients, and changes to one method can affect other clients. The solution is to place methods needed by each client in specific interfaces, which are then inherited by the service class. This way, if an interface for one client changes, other clients remain unaffected, ensuring cohesive and maintainable design without unintended side effects.  
**public interface Athlete** {

void compete();

} **public interface SwimmingAthlete extends Athlete** {

void swim();

} **public interface JumpingAthlete extends Athlete** {

void highJump();

void longJump();

} **public class JohnDoe implements SwimmingAthlete** {

**public void compete()** {

System.out.println("John Doe started competing");

}

**public void swim**() {

System.out.println("John Doe started swimming");

}

}  
**When do we say a software is of bad design?**  
Characteristics of software with ” Bad Design“ **rigid:** It is hard to change because every change affects too many other parts of the system. **Fragile:** When you make a change,unexpected parts of the system breaks. **Immobile:** It is hard to reuse in another application because it cannot be disentangled from the current application.  
**What is it that makes a design rigid, fragile and immobile?** Interdependence of the modules within that design, so we should achieve low coupling and high cohesion  
**The Dependency Inversion Principle (DIP)** emphasizes that high-level modules should not depend on low-level modules; instead, both should depend on abstractions. Additionally, abstractions should not depend on details; rather, details should depend on abstractions. This principle highlights that abstractions contain minimal code or logic and remain relatively stable. In contrast, the non-abstract modules of the software are more susceptible to changes. Consequently, changes in the system are safe, as nothing else in the entire system relies on these abstract modules.  
// Abstract classes

**class ButtonClient {**

public:

virtual void turnOn() = 0;

virtual void turnOff() = 0;

};**class Button {**

protected:

ButtonClient\* itsClient;

public:

virtual void detect() = 0;

virtual bool getState() = 0; };

**class BtnImpl : public Button {**

public:

BtnImpl(ButtonClient\* bc) {

this->itsClient = bc; }

void detect() {

bool buttonOn = getState();

if (buttonOn)

itsClient->turnOn();

else

itsClient->turnOff();}  
}; **class Lamp : public ButtonClient {**

public:

**void turnOn() {**

cout << "On" << endl;}

**void turnOff() {**

cout << "Off" << endl;}};  
**The concept of Design by Contract (DbC)** helps ensure that classes conform to the Liskov Substitution Principle (LSP) by defining clear contracts between classes. In DbC, methods of classes declare preconditions and postconditions. Preconditions must be true for a method to execute, and upon completion, the method guarantees that the postconditions will be true. When used with inheritance, preconditions and postconditions may be replaced in derived methods, but only in specific ways: preconditions can only be replaced with weaker preconditions, and postconditions can only be replaced with stronger postconditions.  
**class LinkedList {**

/\*\*

\* Adds an element to the end of the list

\* PRE: element != null

\* POST: this->getLength() == old->getLength() + 1

\* && this->contains(element) == true

\*/

public void addElement(Object element) {

// Implementation

}

}   
**class Set extends LinkedList {**

/\*\*

\* Adds an element to this set, provided

\* element is not already in the set

\* PRE: element != null

\* && this->contains(element) == false

\* POST: this->getLength() == old->getLength() + 1

\* && this->contains(element) == true

\*/

@Override

**public void addElement(Object element) {**

// Implementation

}

} **7. Design Pattern: Pattern Questions during design** - Object-oriented design is much more complex than the "conventional" procedural design - Developers must make many decisions that are not directly derived from the task at hand: - **\*\*Finding suitable objects:\*\*** - Which classes should a system have? - Which attributes and methods are combined in a class? - **\*\*Inheritance hierarchy:\*\*** - Which classes inherit meaningfully from each other? - Where are abstract classes used? - **\*\*Interface design:\*\*** - Which data does an object make available to which other objects using which methods? - **\*\*Relationships between classes:\*\*** - Which object creates or owns another object or manages a reference to another object? - **"Additional effort"** to obtain a more flexible design that can be adapted more easily to changes in requirements. **Each pattern** describes a recurring problem in our environment and explains the core of the solution to this problem, so that you can use the solution as often as you like, without ever taking it out a second time. • Patterns enable the transfer and reuse of proven knowledge • Patterns define a general procedure • Patterns are not language-specific • Patterns serve the understanding • Patterns offer hints on possible design problems Systems documented by patterns are easier to understand • Patterns solve recurring problems during the software development process • Patterns promote the reuse of already proven solutions. **Implementation Inheritance**: Is about where child object receives • all attributes and • all methods including implementation of the methods • Enables you to quickly define new types of objects based on existing objects **Interface Inheritance**: • is about where a child object receives only the signatures of the methods • no implementation • is also called sub-typing, since a class that implements the derived interface is of the inherited interface type, that is, it fulfils all its requests (and typically more). • allows to define family of objects with identical interface • Basis for polymorphism! **The first-order principle of reusable, object-oriented design is to "program to an interface, not an implementation."** This principle emphasizes the importance of manipulating objects exclusively via the interface defined by abstract classes. **Two significant advantages arise from this approach:** clients know nothing about the specific classes of the objects they use as long as they satisfy the interface, and clients remain unaware of the classes that implement these objects, knowing only the abstract class(es) that implement their interfaces. This leads to a reduction of implementation dependencies between subsystems, thereby achieving low coupling and enhancing system flexibility and maintainability. **Reusability through implementation inheritance**: • Subclass inherits implementation and type of superclass • Can be expanded • White box reuse, since the structure of the superclass is included and is therefore known. **Advantages of implementation inheritance**: •Easy to implement, as part of the (OO) programming language. • Inherited methods can be retained or overwritten. • Variables (attributes) can be added, used or ignored. **Disadvantages**: • Class inheritance can lead to an unclean programming style. • Static dependencies are created, i.e. any change in the implementation or structure of an ancestor class affects the inheriting class (which is often explicitly desired, so this can be an advantage or a disadvantage). **Reusability through Object composition**: • Objects in existing classes are included in a structure (for example, by aggregation). • Only the interface of the embedded objects is known!!! • Black box reuse. **Advantages of object composition**: • Increased flexibility: Relationships between objects can be created and discarded dynamically at runtime (in contrast to those between classes). • Involved objects are only known through their interface i.e. they can be exchanged for objects with the same interface without having to take different implementations into account! **Disadvantage**: More complicated to develop

Past Exam **– WS07 – Theory**  
**How is the Unified Process structured? Describe the process structure at the top level (possibly with sketch)**  
The RUP focuses on these 5 best practices which are, Iterative Software Development, requirements management, Use of component-based software architectures, Visual Software Modelling, Software quality testing, Controlled change management. The process structure is 2-dimenstional, because it incorporates time-phased project milestones (Inception, Elaboration, Construction, Transition) and concurrent development activities (such as requirements, design, implementation, testing). This allows for activities to overlap and be revisited as needed throughout the development lifecycle. E.g. during the analysis implementation activity during the realization of a prototype. Analysis activity during the implementation phase to clarify open analysis questions that have only now become clear, so the Process Structure is **1st dimensional**: This dimension represents the time-based aspects of the process. It includes Cycle, Phase, Iteration, Milestone. **2nd dimensional:** Components, activities, Artifacts, Disciplines, static aspects  
A screenshot of a computer

Description automatically generated  
**Which eXtreme Programming practices can be meaningfully connected to the UP. Choose five and justify your answer?  
Small Releases**: This means that the release cycle of one to three months, the release consists of several iterations of one to three weeks . Iterations can be broken down into work packages of one to three days **Planning Game**: Here the requirements are created by the customer in the form of so-called user stories. Its the customer who priorities the stories. Also, the Customer determines which stories should be implemented in the upcoming iteration. Although the Developers specify the effort they expect for the realization of a particular features. **Programming in pairs:** The first partner takes care of the current coding, whereas the second partner checks the code for typos and logical errors and develops strategies for further implementations. Paring is dynamic: New partners are sought for each work packages. **Continuous Integration:** Newly developed or modified program parts are to be reintegrated into the current code base after only a few hours. For this purpose. An integration and test system are used, on which all test cases must be checked again after the changes have been imported, this practice makes sure the overall production environment is running no problems by handling the exceptions.  **Coding Standard** Ensuring uniform code that can be understood and modified by all the developers, will help the overall development process to be smoother and stable. **List any important steps to adapt the UP to small projects**? In order for us to have a successful and effective development process using an extensive software development process on a smaller project, we should only produce artifacts that bring concrete added value to the projects, which means we should avoid the creations of unnecessary documentation or models that do not directly contribute to the success of the project. Even for smaller projects, planning should be done iteratively. This involves breaking down the projects into smaller, manageable iterations, introducing multiple releases despite a small project helps in getting early feedback, improving the product incrementally, and ensuring that the project stays on track. **A software system is to be developed to support the circulation process of a library. A first workshop with the customer resulted in the use case diagram attached on the last page**. **a) Explain the meaning of "extend"** Extending use case is an optional addition. "label" denotes so-called "extension point", i.e. an identifying name for the case in which the supplementary use case is to be executed. "condition" describes the condition that must be met for the supplementary use case to be executed. **b) Explain the meaning of "include"** Included use case is a necessary component. Goal: Avoid redundancies. **What are the main difficulties and hazards in developing the use case model? What techniques do you know to mitigate these difficulties? Describe them very briefly**. We can say that the choice of granularity and the right level of abstraction is one of the difficulties developing the use case model. On the one hand, functional requirements should be presented in context of related requirements; on the other hand, there should be as little content overlap as possible in the presentation. Customer-experience functionality cannot always be broken down into non overlapping segments. This can result in very heterogeneous collections of use cases that are difficult to keep consistent. Another danger is that requirements and design aspects become mixed, which means it should a conclusive justification at the end providing clear and concise concepts. To improve our analysis, the advice is to check granularity with tests, by writing in an essential style. **Boss Test.** What did you do all day? Test from “Elementary Business Process”: activity performed by 1 person at 1 place at 1 time that leads to measurable result. **Size Test**: Elaborated use case approx.. 6-10 text pages. **A customer's bank details, telephone number and address are recorded, and the system allows one customer to have several bank details, telephone numbers or addresses. The type of address can be street address, P.O. box address or foreign address. The foreign address has no uniform structure, while the street address and P.O. box address comply with the postal code and city and differ only by street and P.O. box, respectively. The system should be tolerant both to the expansion of address types and to the change of address type at runtime. a) Develop a class diagram for the Customer Management component of the library application from Task 3. b) Demonstrate how you use GRASP principles in your design**. **Low Coupling**: 3 interfaces to the different address types become one. High Cohesion: Component address contains only classes with high internal cohesion. Protected Variations: changes within the Addresses are decoupled by stable interface**. c) If you use a design pattern, name it and explain its purpose and goals/motivation**. **Facade Pattern** Purpose: Provide a uniform interface to a set of interfaces of a subsystem. The facade class defines an abstract interface that simplifies the use of the subsystem. **Motivation**: Minimization of communication Minimization of dependencies between subsystems. **a)** **What is the purpose of the Observer pattern?** The Observer behavior pattern is used to be able to observe the status of an object in the program and to react accordingly in case of changes. Maintaining consistency of objects that are related to each other without coupling them too tightly. **b) What is the structure of the pattern**A diagram of a server

Description automatically generated **c) Give a concrete example of the use of the Observer pattern. (Can also be within another pattern)** Example: Model-View-Controller (MVC). The view observes the model, because as soon as something changes in the model, the view must notice this to display the part of the model data for which it is responsible in an updated form. The view depends on the model and must be kept consistent. This means that the view logs on to the model as an observer, provides an update functionality that is called by the model as soon as it has changed and then retrieves the required data from the model.

**Past Exam – SS17** **– Theory What project phases does the Unified Process distinguish? Name each phase and briefly characterise each phase in terms of the objectives to be achieved and the activities taking place**

**ASK THE PROF. AS ON THE INTERNET YOU SHOW THERE ARE SOME MINOR DIFFERNECES THE WAY IN TEH OPENUP, UP and RUP, their phases and disciplines are called, so they are different right? And i cannot use the names from the RUP for all the other ones?  
What so-called "disciplines" does the OpenUP distinguish? Can they be assigned to phases?**  
OpenUP is a lean Unified Process that applies iterative and incremental approaches within a structured lifecycle. These disciplines can indeed be assigned to the various phases of a project. The main disciplines in OpenUP include **Architecture**: This discipline explains how to create an architecture from architecturally significant requirements. The purpose of this discipline is to evolve a robust architecture for the system. Then we have a discipline known as **configuration and change management**: This discipline explains how to control changes to artifacts, ensuring synchronized evolution of the set of Work Products composing a software system. The purpose of this discipline is to: Maintain a consistent set of work products as they evolve. Maintain consistent builds of the software. Provide an efficient means to adapt to changes and issues, and re-plan work accordingly. Another Discipline that we have in this process is called **Development:** This one explains how to design and implement a technical solution that conforms to the architecture and supports the requirements. The purpose of this discipline is to: Transform the requirements into a design of the system-to-be. To also adapt the design to match the implementation environment. To build the system incrementally. To verify the technical units used to build the system work as specified. Next one is **Project Management:** This one explains how to coach, facilitate and support the team, helping it to deal with risk and obstacles found when building software. The purpose of this discipline is to: Encourage stakeholder consensus and prioritizing the sequence of work. Stimulate team collaboration on creating long term and short-term plans for the project. Focuses the team on continually delivering tested software for stakeholder evaluation. Helps to create an effective working environment to maximize team productivity. Moreover, keep stakeholders and the team informed on project progress. **Requirements**: This discipline explains how to elicit, analyse, specify, validate, and manage the requirements for the system to be developed. The purpose of this discipline is to: Understand the problem to be solved Understand stakeholder needs (what users want) Define the requirements for the solution (what the system must do), the boundaries, external interfaces, technical constraints. Provide the initial basis for estimating cost and schedule. **Test:** This discipline explains how to provide feedback about the maturing system by designing, implementing, running, and evaluating tests. The purpose of this discipline is to: Provide early and frequent feedback that the system satisfies the requirements Objectively measure progress in small increments Identify issues with the solution. Provide assurance that changes to the system do not introduce new defects. OpenUP has the following phases, Inception, Elaboration Construction and Transition. Each discipline is involved in multiple phases, but their primary focus can differ in different phases **Which eXtreme Programming practices can be usefully combined with the UP? Choose five and give reasons for your answer.**  
Several eXtreme Programming (XP) practices can be usefully combined with the Unified Process (UP) to enhance its effectiveness. Firstly, **Small Releases** can align well with UP's iterative approach, allowing for frequent, incremental releases that ensure continuous customer feedback and validation. **Planning Game** can be integrated into UP's requirements discipline, where user stories created and prioritized by the customer help in defining the scope and focus of each iteration, ensuring that the most valuable features are delivered first. **Continuous Integration** fits seamlessly into UP's development and testing phases, promoting regular integration and testing of code to detect and fix issues early. **Test-Driven Development (TDD)**, an aspect of XP's automatic testing practice, ensures that tests are written before code, leading to better-designed, more reliable software and aligning with UP's emphasis on quality. Lastly, **Pair Programming** can enhance UP's development discipline by improving code quality through continuous code review and fostering knowledge sharing among team members. These practices collectively enhance UP's structured framework with agile flexibility, continuous feedback, and improved collaboration.  
**What does the open/closed principle mean? Explain!**  
As a concept, the Software entities (classes, modules, functions) should be open for extension but closed for modification. Open For Extension“ This means that the behaviour of the module can be extended. we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. ” Closed for Modification“ The source code of such a module is inviolate. No one is allowed to make source code changes to it. This can be achieved by using abstraction, which means, any module that manipulates solely abstractions will never need to change since the abstractions are fixed → closed for modification and deriving a new class from the abstractions can change the behaviour of the module → open for extension  
**Given a first implementation of a class "AreaCalculator" according to the implementation below. The requirement is to calculate the total area of a list of rectangles. The implementation shown does this**A white background with black text

Description automatically generated **As is so often the case, additional requirements arise during the course of the project: "AreaCalculator" should now no longer only be able to calculate the total area from rectangles, but also from other shapes (circles, triangles, ...), which are transferred to calculateArea in a mixed field. The complete set of possible shapes is not yet known at the current stage of the project.   
1. Specify an implementation that is tolerant of newly added forms. Follow the open/closed principle.   
On the 2nd page end   
2. Use this example to explain the open/closed principle.** The Open/Closed Principle is demonstrated in the provided `AreaCalculator` example by ensuring the class is open for extension but closed for modification. The `AreaCalculator` calculates the total area of various shapes by operating on a collection of `Shape` objects, an abstract class with a `getArea()` method. To add new shapes like `Triangle` or `Polygon`, new classes can be created that implement the `Shape` interface without modifying the `AreaCalculator` code. This approach allows extending the system's functionality with new shapes while maintaining the existing functionality, ensuring stability and reducing the risk of bugs.

**Past Exam – SS18** **– Theory** **The waterfall model is one of the best-known process models. However, it has serious weaknesses. Explain them!** Central assumption of the waterfall model: All requirements for the system to be developed are known and stable, i.e. they remain valid throughout the duration of the project. Very academic! Rarely true! The coding starts very late: Many problems only occur during implementation or even during system integration → delayed recognition of problems! Feedback with the end user only shortly before commissioning of the system. Consequence: It is necessary to jump back to supposedly completed phases and to revise parts. Project progress control misleading: Formally simple control, e.g. based on phase results Together with delayed problem identification and delayed feedback from the end user, real and formal project progress do not match. **In newer process models such as the Unified Process, an attempt has been made to avoid these weaknesses through appropriate measures. What are significant improvements?** A process framework that can be adapted and extended. The main causes of software development problems are addressed by 6 best practices. The RUP focuses on these 6 best practices: Iterative Software Development requirements management Use of component-based software architectures Visual software modelling checking software quality Controlled change management. Process structure → 2-dimensional, Workflows via phases arranged in time in all phases almost all kinds of activities can occur, because of explained points before, we can say that this process is adaptable to individual needs . Has a focus on key best practices. THE UML process can be fairly used. 2-dimensional process that can depict overlapping activities and phases. Covers the entire life cycle and supports object-oriented development and reuse. c**) What are the essential practices of agile procedural models? Do they conflict with more formal procedural models such as the Unified Process**. The 10 XP-practices are not contradictory to the formal models. Agile procedural models, like Extreme Programming (XP), emphasize flexibility, collaboration, and rapid iteration. Planning involves writing user stories, creating a release schedule, making frequent small releases, measuring project velocity, and holding daily stand-up meetings. Design principles include simplicity, using system metaphors, CRC cards for design, spike solutions to reduce risk, and continuous refactoring. Coding practices ensure the customer is always available, code follows agreed standards, unit tests are written first, pair programming is standard, integration is frequent, collective code ownership is maintained, and optimization is delayed. Testing mandates that all code has unit tests, passes them before release, and bugs lead to new tests, with acceptance tests run regularly and results published. These practices are not contradictory to formal process models and can enhance them by adding agility and responsiveness**. Name the 5 most important quality features that one is aiming for when developing software. Characterize each quality characteristic in brief and thus indicate what it means.** LOOK AT THE FIRST BOX OF THE FIRST PAGE. Explain the Abstract Factory Pattern in terms of purpose, applicability and its structure. Give a small code example. **Explain the Abstract Factory Pattern in terms of purpose, applicability and its structure. Give a small code example**. **The purpose** of the abstract factory pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. **The applicability** of this pattern can be understood accordingly. Use the Abstract Factory Pattern if: A system should be independent of how its products are created, assembled, and represented. A system is to be configured with one of several product families. A family of related product families has been designed to be used together, and you need to ensure this consistency. You want to provide a class library of products for which you only want to disclose the interfaces, not the implementations.  
**The Structure:**  
A screenshot of a computer program

Description automatically generated

Past Exam **–** **WS22 – Theory**  
**1. a) Popular processes “Waterfall model” and “Unified Process” are ……… Describe their phases.**  
YOU CAN FIND THE ANSWER ON THE FRIST PAGE-2nd BOX – AT THE END (Waterfall) AND FIRST PAGE-4th BOX – AT THE END. **b) How is SCRUM structured? Why can SCRUM dispense their phases?**  
Scrum is structured as a framework for agile project management, focusing on iterative progress through time-boxed iterations called "sprints." Within this framework, there are three key roles: the Scrum Team, the Scrum Master, and the Product Owner. The Scrum Team is cross-functional and self-organizing, responsible for delivering product increments. The Scrum Master ensures the team adheres to Scrum practices, facilitates meetings, and helps remove impediments. The Product Owner manages the product backlog, prioritizes user stories, and represents stakeholder interests. Scrum organizes its workflow around several key events. Each sprint is a time-boxed period, typically lasting between 2-4 weeks, during which a potentially shippable product increment is created. At the beginning of each sprint, the Sprint Planning event is held to decide which product backlog items to work on. Daily Scrums are 15-minute meetings where the team synchronizes activities and plans for the next 24 hours. At the end of each sprint, a Sprint Review is conducted to inspect the increment and adapt the product backlog if needed. The Sprint Retrospective provides an opportunity for the team to reflect on their process and make improvements. The framework includes essential artifacts that guide the development process. The Product Backlog is an ordered list of everything that might be needed in the product, managed by the Product Owner. The Sprint Backlog consists of the product backlog items selected for the sprint, along with a plan for delivering the product increment. The Increment is the sum of all completed product backlog items during a sprint. The Definition of Done (DoD) is a shared understanding of what it means for work to be complete, ensuring that increments are releasable. **Scrum can dispense with traditional project** phases because it is designed to be iterative and incremental. Instead of distinct phases like planning, design, implementation, and testing, Scrum incorporates all these activities into each sprint. This iterative development allows for frequent reassessment and adaptation. The flexibility of Scrum enables teams to adapt to changes quickly and efficiently, accommodating new requirements within each sprint. Traditional phases often require detailed upfront planning, which can be problematic in dynamic environments where requirements change. Scrum’s flexibility allows teams to adapt to changes quickly and efficiently, accommodating new requirements within each sprint. Also, why Scrum can dispense their phases because, each sprint aims to deliver a potentially shippable product increment. This focus on delivering value in short cycles ensures that the product evolves based on real feedback and emerging needs, rather than following a predefined, linear path. **2. a) see CODE IN below and describe which design principles it violates.**A screenshot of a computer program

Description automatically generated

The code has violated the SOLD principles. Firstly it breaches the **Single Responsibility Principle** (SRP) as the “Database” class handles multiple responsibilities (get, remove, update, insert operations) that should be ideally segregated. Secondly, it violates the **Dependency Inversion** Principle (DIP) snice “BudgetReport” class is tightly coupled to a specific “Database” implementation rather than an abstraction, which restricts flexibility and reuse. Additionally, the code fails the Open/Closed Principle (OCP) because it cannot be extended to support new types of databases without modifying existing code. Lastly, it does not also follow the **Liskov Substitution** Principle (LSP) because it does not provide a mechanism to easily substitute different database types, thus limiting polymorphism and interchangeability.  
**b) Rewrite the code, so that it does not violate any design principle and that any Database can be used.**A screenshot of a computer code

Description automatically generatedA screenshot of a computer program

Description automatically generated

between elements. Coupling measures the interdependence of elements, which are connected when they know about each other or rely on each other's services. High coupling, where classes depend on many other classes, is undesirable. However, dependencies on stable elements, like standard libraries, are generally not critical. **High Cohesion** aims to assign responsibilities in such a way that cohesion remains high, use this criterion to evaluate alternatives. In the sense of object design, functional cohesion is a measure of the relationship and focus of an element’s responsibilities. An element with closely related responsibilities that perform only a few tasks has a high degree of cohesion, so in other words, classes with many thousands of lines of code are a sure indication of low cohesion. Low cohesion often goes hand in hand with high coupling. The approach should be that the division of the elements should be according to the functional aspects or delegation of partial responsibilities   
  
A diagram of a sales flow

Description automatically generated  
According to Grady Booch: High functional cohesion exists when the elements of a component all work together to create a clearly defined behaviour. **Information Expert:** This refers to assigning responsibilities to the information expert. This is the class that has the necessary information to fulfill the responsibility. Information Expert describes a fundamental principle in object design. This pattern recommends to delegate responsibility to domain specialists and thus sharing responsibility. Decisive point: Who has enough knowledge to solve the problem? information Expert is no silver bullet The use of Information Expert is far less obvious when classes are considered as candidates that do not yet have a fixed place in the domain or design model. In this case, the new classes must first be turned into Information Experts by adding associations. **Controller:** This aims to assign responsibility to a class that represents the overall system or a root object, a device on which the software is running, or a large subsystem, it also represents a use case scenario in which the system operation takes place (a use case or a session controller), to use the same controller class for all system events in the same use case scenario. The controller principle teaches how the principle of strict separation of user interface and application logic can be implemented. By limiting cross-layer communication to a few classes, dependencies between the layers are minimized. Controllers of the first category (class that represents the overall system, so-called facade controllers) can lead to low cohesion, so replace controllers with multiple session or use case controllers and share responsibilities accordingly. **Polymorphism**: The question is, who is responsible if the behavior varies according to type? Polymorphism aims to address this by assigning responsibility for behavior with polymorphic operations to the types for which the behavior varies. This approach essentially removes the if-else cascade, providing the advantage of extending the code. New variants can be added without changing the existing code. **Pure Fabrication** in GRASP addresses the problem of high coupling or low cohesion that arises when software objects are based strictly on real-world objects. The solution is to create a highly cohesive utility class that is not a part of the domain model but serves a specific purpose, thereby reducing dependencies between software layers and encapsulating technical details. For example, a PersistentStorage class manages data persistence for sale objects, making the functionality broadly available and maintaining separation of concerns within the system. **Indirection** addresses the challenge of delegating responsibility without a direct link between the service provider and recipient. The solution is to assign this responsibility to an intermediate object that mediates between components or services, preventing direct contact. For instance, Tax Master adapters facilitate the interaction between sales and tax calculations, while PersistentStorage handles mediation between sales and databases. This principle is echoed by David Wheeler's saying, "Most problems in computer science can be solved by another indirection layer," though it's also noted that "most performance problems can be solved by removing another indirection layer." **The Protected Variations** addresses the design of objects, subsystems, and systems to prevent undesirable effects from changes or instabilities in these elements. The solution involves identifying potential points of change or instability and encapsulating these points with stable interfaces to minimize their impact. This principle is central to object design and underpins various other principles and patterns, such as information hiding, polymorphism, and indirection. It can be challenging due to the difficulty in predicting where instabilities will occur. An example application is tax calculation using the TaxMasterAdapter interface. **Example for fragile Code**   
**public void fragileMethod**() {   
AccountHolder holder = sale.getPayment().getAccount().getAccountHolder(); … }   
**Fundamental OO-Design Principles:** Law of Constantine A modular structure is stable if cohesion within each component is high coupling between components is low. A first-order principle of software architecture is to increase cohesion and reduce coupling. **SOLID Design Principles**: **Single Responsibility Principle (SRP)** Every object should have a single responsibility. **Open Closed Principle (OCP)** A software module should be closed for modification but open for extension. **Liskov’s Substitution Principle (LSP)** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. **Interface Segregation Principle (ISP)** Many specific interfaces are better than one combined general interface. **Dependency Inversion Principle (DIP)** High-level modules should not depend upon low-level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. **Law of Demeter (LoD)** An object should avoid invoking methods of a member object returned by another method (”Use only one dot!”). ”Don’t talk to strangers”. **Information hiding** Hide the information about internal structures and allow access only via defined interfaces. **Single Choice Principle** The exhaustive list of alternatives should live in exactly one place. **Design by Contract (DbC)** Software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. **IN DETAILS NOW**  
**Single Responsibility Principle (SRP)** Every object should have a single responsibility. That responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility. ”There should never be more than one reason for a class to change.” What is a Responsibility? One responsibility of a class = one reason for the class to change. **Open Closed Principle (OCP)** Software entities (classes, modules, functions) should be open for extension but closed for modification. ” **Open For Extension“** This means that the behavior of the module can be extended. we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. **Closed for Modification.** The source code of such a module is inviolate. No is allowed to make source changes to it. **How to Achieve the Open/Closed Principle?** A contradiction can be solved by using abstraction: any module that manipulates solely abstractions will never need to change since the abstractions are fixed, which refers to closed for modification. On the other hand, deriving a new class from the abstractions can change the behavior of the module, refers to open for extension. (Try to make the Member Variables Private and No Global Variables )  
**// Open/Closed Principle**  
**public abstract class Shape**{   
public abstract double Area();  
}  
**public class Rectangle : Shape**{  
 public double Width { get; set; }   
 public double Height { get; set; }  
 public override double Area(){  
return Width \* Height;}  
}  
**public class Circle : Shape**{  
 public double Radius { get; set; }  
 public override double Area(){  
 return Math.PI \* Radius \* Radius;}  
}  
**public double Area(Shape[] shapes)**{  
 double area = 0;  
 foreach (var shape in shapes){area += shape.Area();}  
 return area;  
}  
**Structural Subtyping**  
Language rules of object-oriented programming languages provide structural replaceability: Objects of a subclass can be used as substitutes for objects of (direct or indirect) superclasses. subclass has all attributes and methods of the superclass. static type checking guarantees that no run-time errors due to missing or inappropriate attributes/operations can occur. **Liskov Substitution Principle (LSP)** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. Modules that use references to base types must be able to use references to derived types without knowing the difference. Not only structural, but also behavioural subtyping required!  
**public class A {**

public void m1() {}  
public void m2() {}

**}public class B extends A {**

public void m2() {}  
 public void m3() {**}**

**}**

**public class C {**

**public static void doIt(A obj) {**

obj.m1();

obj.m2();

**}  
public static void main(String[] args) {**

B objB = new B();

C.doIt(objB); **}  
}**

**The Interface Segregation Principle (ISP)** emphasizes that having multiple, client-specific interfaces is preferable to a single, general-purpose interface. Clients should not be forced to depend on interfaces they do not use. Fat interfaces should be avoided as they can be broken down into distinct method groups serving different clients, and changes to one method can affect other clients. The solution is to place methods needed by each client in specific interfaces, which are then inherited by the service class. This way, if an interface for one client changes, other clients remain unaffected, ensuring cohesive and maintainable design without unintended side effects.  
**public interface Athlete** {

void compete();

} **public interface SwimmingAthlete extends Athlete** {

void swim();

} **public interface JumpingAthlete extends Athlete** {

void highJump();

void longJump();

} **public class JohnDoe implements SwimmingAthlete** {

**public void compete()** {

System.out.println("John Doe started competing");

}

**public void swim**() {

System.out.println("John Doe started swimming");

}

}  
**When do we say a software is of bad design?**  
Characteristics of software with ” Bad Design“ **rigid:** It is hard to change because every change affects too many other parts of the system. **Fragile:** When you make a change,unexpected parts of the system breaks. **Immobile:** It is hard to reuse in another application because it cannot be disentangled from the current application.  
**What is it that makes a design rigid, fragile and immobile?** Interdependence of the modules within that design, so we should achieve low coupling and high cohesion  
**The Dependency Inversion Principle (DIP)** emphasizes that high-level modules should not depend on low-level modules; instead, both should depend on abstractions. Additionally, abstractions should not depend on details; rather, details should depend on abstractions. This principle highlights that abstractions contain minimal code or logic and remain relatively stable. In contrast, the non-abstract modules of the software are more susceptible to changes. Consequently, changes in the system are safe, as nothing else in the entire system relies on these abstract modules.  
// Abstract classes

**class ButtonClient {**

public:

virtual void turnOn() = 0;

virtual void turnOff() = 0;

};**class Button {**

protected:

ButtonClient\* itsClient;

public:

virtual void detect() = 0;

virtual bool getState() = 0; };

**class BtnImpl : public Button {**

public:

BtnImpl(ButtonClient\* bc) {

this->itsClient = bc; }

void detect() {

bool buttonOn = getState();

if (buttonOn)

itsClient->turnOn();

else

itsClient->turnOff();}  
}; **class Lamp : public ButtonClient {**

public:

**void turnOn() {**

cout << "On" << endl;}

**void turnOff() {**

cout << "Off" << endl;}};  
**The concept of Design by Contract (DbC)** helps ensure that classes conform to the Liskov Substitution Principle (LSP) by defining clear contracts between classes. In DbC, methods of classes declare preconditions and postconditions. Preconditions must be true for a method to execute, and upon completion, the method guarantees that the postconditions will be true. When used with inheritance, preconditions and postconditions may be replaced in derived methods, but only in specific ways: preconditions can only be replaced with weaker preconditions, and postconditions can only be replaced with stronger postconditions.  
**class LinkedList {**

/\*\*

\* Adds an element to the end of the list

\* PRE: element != null

\* POST: this->getLength() == old->getLength() + 1

\* && this->contains(element) == true

\*/

public void addElement(Object element) {

// Implementation

}

}   
**class Set extends LinkedList {**

/\*\*

\* Adds an element to this set, provided

\* element is not already in the set

\* PRE: element != null

\* && this->contains(element) == false

\* POST: this->getLength() == old->getLength() + 1

\* && this->contains(element) == true

\*/

@Override

**public void addElement(Object element) {**

// Implementation

}

} **7. Design Pattern: Pattern Questions during design** - Object-oriented design is much more complex than the "conventional" procedural design - Developers must make many decisions that are not directly derived from the task at hand: - **\*\*Finding suitable objects:\*\*** - Which classes should a system have? - Which attributes and methods are combined in a class? - **\*\*Inheritance hierarchy:\*\*** - Which classes inherit meaningfully from each other? - Where are abstract classes used? - **\*\*Interface design:\*\*** - Which data does an object make available to which other objects using which methods? - **\*\*Relationships between classes:\*\*** - Which object creates or owns another object or manages a reference to another object? - **"Additional effort"** to obtain a more flexible design that can be adapted more easily to changes in requirements. **Each pattern** describes a recurring problem in our environment and explains the core of the solution to this problem, so that you can use the solution as often as you like, without ever taking it out a second time. • Patterns enable the transfer and reuse of proven knowledge • Patterns define a general procedure • Patterns are not language-specific • Patterns serve the understanding • Patterns offer hints on possible design problems Systems documented by patterns are easier to understand • Patterns solve recurring problems during the software development process • Patterns promote the reuse of already proven solutions. **Implementation Inheritance**: Is about where child object receives • all attributes and • all methods including implementation of the methods • Enables you to quickly define new types of objects based on existing objects **Interface Inheritance**: • is about where a child object receives only the signatures of the methods • no implementation • is also called sub-typing, since a class that implements the derived interface is of the inherited interface type, that is, it fulfils all its requests (and typically more). • allows to define family of objects with identical interface • Basis for polymorphism! **The first-order principle of reusable, object-oriented design is to "program to an interface, not an implementation."** This principle emphasizes the importance of manipulating objects exclusively via the interface defined by abstract classes. **Two significant advantages arise from this approach:** clients know nothing about the specific classes of the objects they use as long as they satisfy the interface, and clients remain unaware of the classes that implement these objects, knowing only the abstract class(es) that implement their interfaces. This leads to a reduction of implementation dependencies between subsystems, thereby achieving low coupling and enhancing system flexibility and maintainability. **Reusability through implementation inheritance**: • Subclass inherits implementation and type of superclass • Can be expanded • White box reuse, since the structure of the superclass is included and is therefore known. **Advantages of implementation inheritance**: •Easy to implement, as part of the (OO) programming language. • Inherited methods can be retained or overwritten. • Variables (attributes) can be added, used or ignored. **Disadvantages**: • Class inheritance can lead to an unclean programming style. • Static dependencies are created, i.e. any change in the implementation or structure of an ancestor class affects the inheriting class (which is often explicitly desired, so this can be an advantage or a disadvantage). **Reusability through Object composition**: • Objects in existing classes are included in a structure (for example, by aggregation). • Only the interface of the embedded objects is known!!! • Black box reuse. **Advantages of object composition**: • Increased flexibility: Relationships between objects can be created and discarded dynamically at runtime (in contrast to those between classes). • Involved objects are only known through their interface i.e. they can be exchanged for objects with the same interface without having to take different implementations into account! **Disadvantage**: More complicated to develop

OPEN AND CLOSED ANSWER FROM PAST-EXAM-SS17

A screenshot of a computer code

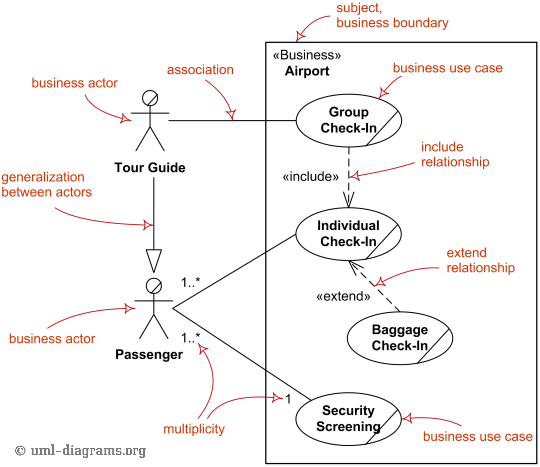
Description automatically generated

A computer code with text

Description automatically generated

A diagram of a process

Description automatically generatedA diagram of sales

Description automatically generated

**ACTIVITY DIAGRAMS**

**STATE MACHINE DIAGRAMS**

**USE CASE**

A diagram of a machine

Description automatically generatedA diagram of a customer payment

Description automatically generated

A diagram of a diagram

Description automatically generated

|  |  |
| --- | --- |
| **Term** | **Invoice** |
| **Synonyms** |  |
| **Shortcut** | **-** |
| **Definition** | **Each invoice results from a contract. It invoices services rendered or deliveries and is addressed to a customer.** |
| **Delimitation** | **There are individual, monthly, partial and collective invoices. An invoice has an invoice recipient, a date, an invoice number, and invoice items that are used to list the individual services and deliveries to be billed for. Each item contains a description, a number, an individual amount and a total amount (item total). The invoice contains a final total (sum of all items). The sales tax is displayed separately for each item and all totals.** |
|  |  |
| **Constraints** | **-** |
| **Contact person** | **-** |
| **Status** | **Final** |
| **Changes** | **...** |

A diagram of a tree

Description automatically generatedA diagram of a business process

Description automatically generated

Assosication Aggeration and composition between classes and relationship also multiplicity (Class Diagram ofc)

Component diagram check that too

What you need to do here is to make sure that you have one example from each type of the diagram from the course. Where you need to mark every symbol meanings for example, also for the SOLIDs chapter add one example each   
  
Maybe you need to apply patterns on a question of a class domain  
Generally in the exam, the class domain in the exam just expects you to create the classes with their relations to each other, multiplicity, add defined attributes and if you know any functions like to calculate control