Software Design Exam Study Guide

This study guide synthesizes information from the provided lecture materials to help you prepare for an exam covering key concepts in Software Design.

1. Introduction to Software Engineering and Process

- Software is ubiquitous, encompassing executable programs, associated data, and documentation. Unlike hardware which wears out, software deteriorates, is custom-built rather than manufactured using components, and is complex.
- The **software crisis** refers to past problems like projects being late/over-budget, inefficient/low-quality software, unmet requirements, unmanageable projects, and outright failure of delivery.
- **Software Engineering (SE)** is a discipline using **process**, **methods**, and **tools** to manage the complexities and problems of software development.
- Frederick Brooks' "No Silver Bullet" argues there is no single development in technology or management offering an order-of-magnitude improvement in productivity, reliability, or simplicity. The **essential difficulties** of building software include **complexity**, **conformity** (to existing interfaces), **changeability**, and **invisibility** (no ready geometric representation).
- Software development often faces **myths**, such as adding people to a late project making it faster (Fact: it makes it later), or vague objectives being sufficient to start coding (Fact: leads to failure).

2. Software Process & Agile Development

- The software process provides a framework for tasks needed to build high-quality software, offering stability, control, and organization.
- Typical phases include the **Definition Phase** (problem definition, requirements, planning), the **Development Phase** (problem solution, design, coding, testing), and the **Support (Maintenance) Phase** (software evolution, adaptation, correction, enhancements, refactoring).
- The **Agile Manifesto** values **individuals and interactions** over processes and tools, **working software** over comprehensive documentation, **customer collaboration** over contract negotiation, and **responding to change** over following a plan.
- **Agile Modeling** emphasizes that the purpose of modeling is primarily to **understand**, not to document. It focuses on difficult parts, suggests modeling in pairs/triads, having developers model for themselves, creating models in parallel, treating diagrams as throw-away, and using simple tools/notation. Tested code is the true design.
- **Scrum** is an **iterative, incremental framework** founded on empirical process control theory. It's most beneficial for complicated or complex problems.
- The Scrum framework includes **Roles** (Product Owner, Development Team, ScrumMaster), **Artifacts** (Product Backlog, Sprint Backlog, Product Increment), and **Ceremonies** (Backlog Refinement, Sprint Planning, Daily Scrum, Sprint Review, Retrospective).
- Development in Scrum occurs in 2-4 week **"sprints"**.

3. Project Management and SCM

- Effective project management focuses on the **4 Ps**: **People**, **Product**, **Process**, and **Project** (progress control).
- The **"First Law"** states that no matter where you are in the system life cycle, the system will change, and the desire to change it will persist. These changes can affect data, code, documents, requirements, and software models.
- **Software Configuration Management (SCM)** is the task of tracking and controlling changes in software.
- An SCM repository typically includes a **Version Control System (VCS)** and an **Issue Tracking System (ITS)**.
- A **VCS** (or Revision Control System) manages changes to collections of information. It tracks what, why, and who made changes, allows undoing/redoing changes, and supports branching.
- Basic VCS features include keeping history of changes, adding/recovering versions, access control, and logging.
- **Centralized VCS** (like Subversion - SVN) uses a single server. Projects have a main line (**trunk**), markers for releases (**tags**), and side lines (**branches**).
- **Distributed VCS** (like Git) gives everyone their own local repository. Remote updates/commits are like branch merges. Key features include **Diff** (showing differences), **Branch** (creating separate lines of development), and **Merge** (combining changes).
- An **ITS** (Issue Tracking System) manages lists of issues, such as bugs or feature requests. Basic features include structurally describing issues (status, severity), tracking status, and assigning a unique ID. A typical workflow involves issues being opened by a requester, assessed/assigned by a triager, investigated/resolved by a developer, and tested by QA. Issue resolutions can be Fixed, Invalid, Duplicate, or Won't fix.
- Many project hosting websites integrate ITS and VCS. Tools like GitLab map Agile artifacts (User Story, Task, Epic, Sprint) to features like Issues, Task Lists, Epics, and Milestones.

4. Requirements Analysis

- A **requirement** is a capability or condition the system (and project) must conform to. Requirements analysis focuses on the **WHAT** the system should do, not the HOW.
- Requirements analysis is hard, and major causes of project failures include **incomplete requirements**, **changing requirements**, and **poor user input**. Essential solutions involve classifying requirements, using iterative/evolutionary analysis, and employing Use Cases.
- Requirements can be classified as **Functional** (features, capabilities, security) or **Non-functional**. Non-functional requirements include **Usability**, **Reliability**, and **Performance**.
- **Iterative and Evolutionary Requirements Analysis** is motivated by the fact that requirements often change. Strategies involve specifying the most architecturally significant, risky, and high-business-value requirements early, and adapting quickly in short iterations.
- Requirements Elicitation methods include Brainstorming, Interviewing, Ethnography, and Strawman/Prototype.
- **Use Cases** are the most widely used approach for capturing requirements and serve as input to many subsequent activities. A use case is a **story of using the system to fulfill stakeholder goals**. It is a **text document**, not a diagram. The **Use-Case Model** is the set of all written use cases; **Use-Case Modeling** is primarily writing text.
- **Stakeholders** are people affected by the project (managers, users, customers, etc.).
- An **Actor** is something with behavior (person, system, organization). A **Scenario** (use case instance) is a specific sequence of actions/interactions. A use case is a collection of related success and failure scenarios describing an actor achieving a goal.
- **Primary actors** use the system to fulfill goals and drive use cases. **Supporting actors** provide services to the system and clarify external interfaces.
- Use Cases are **"Black-Box"**; they describe only **system responsibilities** and focus on **what** the system should do, **not** its internal workings or implementation details.
- Use Cases can have different **Levels of Formality**: **Brief** (one-paragraph main success scenario), **Casual** (multiple paragraphs covering scenarios), or **Fully dressed** (all steps/variations, developed iteratively).
- A **Fully Dressed Use Case Outline** includes sections like Primary Actor, Stakeholders, **Preconditions** (what must be true before), **Success Guarantee** (what's true on successful completion), **Main Success Scenario** (basic flow, deferring conditionals), **Extensions** (alternative flows, success/failure branches), **Special Requirements** (non-functional reqs specific to the UC), and **Technology and Data Variations List** (technical variations in *how*, data schemes).
- A **Use Case Diagram** is a representation of interactions between actors and the system, showing the scope, external actors, and how actors use the system. It is secondary to text documentation. Key elements are Actor symbols, Use Case symbols, Associations, **Includes** dependency (base UC includes sub UC), and **Extends** dependency (UC extends base UC).
- **User Stories** take the form: "As a [role/who], I want [feature/what], so that [business value/why]". They are simple, clear, short descriptions of a customer-valued function.
- A user story has three parts: a written description, a conversation to flesh out details, and tests to determine completeness (Acceptance

Criteria). **Acceptance Criteria** are measurable, define "Done", manage expectations, and can lead to new requirements.
- User stories are written from the user's perspective, focusing on their goal and value.
- User stories relate to use cases: Actor -> Role, Summary of Scenario -> Feature, Reason for Use Case -> Business Value for User Story. The conversation around a user story can lead to more detailed documentation like design documents or use cases.

5. Domain Modeling (Conceptual Design)

- A **Domain Model** is a **visual representation of conceptual classes** or real-situation objects, showing the domain objects, relationships, and attributes. It is illustrated with a set of UML Class diagrams.
- The Domain Model is built upon use cases, serves as a **basis for design and implementation**, and is considered the most important model in OO analysis.
- A **UML Class Diagram** is a visual representation of the main objects and their relations for a system. Elements include **Classes** (with Attributes and Operations) and various **Relationships** (Association, Aggregation, Composition, Generalization).
- Key points about UML Class Diagrams: UML is notation, and a diagram's meaning depends on the **perspective** (Conceptual, Specification, or Implementation). The Domain Model uses a small set of elements focusing on conceptual classes and key relationships.
- **Generalization** is an "is-a" relationship where a subclass inherits from a superclass.
- **Multiplicity** describes the number of instances that can be associated (e.g., *, 1*.., 0..1, 1).
- How to **Build the Domain Model**: Step 1: Identify conceptual classes, Step 2: Decide attributes, Step 3: Identify associations. Steps 1 and 2 often occur together.
- To Identify Conceptual Classes: Reuse/modify existing models, consider common categories (physical objects, places, transactions, roles, etc.), and identify nouns/noun phrases from use cases. There is often no single "correct solution".
- To Decide Attributes: Identify properties of conceptual classes relevant to the problem domain, often nouns/phrases in requirements that need to be remembered. A rule of thumb for distinguishing a Class from an Attribute: If you don't think of something as just a number or text in the real world, it's likely a conceptual class. A **Description Class** contains information describing something.
- To Identify Associations: Determine relationships between instances of conceptual classes. Focus on associations relevant to the use cases and avoid creating too many. Typical associations include part-of, contained-in, recorded-in, description-of, uses/manages, related-to-transaction, and owned-by.

6. Software Design Concepts & Principles

- **Design Engineering** is the process of making decisions about **HOW** to implement software solutions to meet requirements.
- Key concepts in software design include **Modularity**, **Cohesion & Coupling**, **Information Hiding**, **Abstraction & Refinement**, and **Refactoring**.
- **Modularity** means dividing software into separately named and addressable components.
- **Cohesion** is the degree to which elements of a module belong together (perform a single task). **Coupling** is the degree of interdependence between modules. The goal is **high cohesion** and **low coupling**.
- **Information Hiding** means not exposing a module's internal information unless necessary (e.g., using private fields and getter/setter methods).
- **Abstraction** helps manage complexity and anticipate changes by focusing on essential properties and suppressing details. It allows postponement of design decisions. Two types are **Procedural abstraction** (specifies input/output, hides algorithm) and **Data abstraction** (specifies attributes/values, hides representation/manipulation). Abstraction can be applied at different levels, from high-level concepts to detailed implementations. Defining interfaces is a way to use abstraction to anticipate changes.
- **Refinement** is a top-down strategy to reveal lower-level details from high-level abstractions as design progresses. Design proceeds in stages: **High-level (Architecture design)** defining major components and their relationships, and **Low-level (Detailed design)** deciding classes, interfaces, and algorithms.
- **Refactoring** is changing a system's internal structure without altering its external behavior, aiming to make it easier to integrate, test, and maintain.
- Design principles, such as the **SOLID principles**, guide design decisions. The **Single-responsibility principle (SRP)** states a class should have only one job, relating to modularity, high cohesion, and low coupling. The **Open-closed principle (OCP)** states objects should be open for extension but closed for modification. The **Dependency Inversion principle (DIP)** states entities should depend on abstractions, not concretions.

7. High-level Design (Architecture)

- **Software architecture** is a comprehensive framework describing a system's form and structure – its components and how they fit together.
- **Architectural Design** involves designing the overall shape and structure of the system, its components, their properties, and their relationships. The goal is to choose an architecture that reduces risks and meets requirements.
- Software **Architectural Styles** are composed of components, connectors between them (communication, coordination), constraints on integration, and semantic models.
- Common Architecture Patterns include **Pipe & Filter**, **Event-based**, **Layered**, and **Service-oriented Architecture (SOA)**.
- **Pipe & Filter** architecture chains data processing elements, where the output of one is the input to the next.
- **Event-based** architecture involves components reacting to events.
- **Layered/Tiered** architecture defines multiple layers for responsibilities with hierarchical communication.
- **Variant architectures** include **2-layer** (Client-Server, Data-centric) and **3-layer** (Model-View-Controller - MVC).
- In **Client-Server**, tasks are partitioned between providers (servers) and consumers (clients).
- In **Data-centric**, a data store (like a blackboard) is central, accessed by agents, and can notify subscribers of changes.
- A **3-layer** architecture separates **Presentation** (UI), **Logic** (application coordination/calculations), and **Data** (storage/retrieval).
- In **MVC**, the View layer should not handle system events, the Controller handles application logic and events, and the Model layer responds to data operations.
- **SOA** involves principles like **Abstraction** (service is a blackbox), **Autonomous** services, **Stateless** operation, and **Discoverable** services. Patterns include Web Services, RESTful, WCF, and Microservices.

8. Detailed Design

- **Detailed design** involves decomposing subsystems into modules.
- Two approaches to decomposition are **Procedural** (system into functional modules, achieving procedural abstractions) and **Object-oriented** (system into communicating objects, achieving procedural and data abstractions).
- **OO Design** involves identifying responsibilities and assigning them to classes and objects.
- **Responsibilities** are obligations or behaviors based on an object's role. Types are **Doing responsibilities** (doing something itself, initiating/controlling actions in others) and **Knowing responsibilities** (knowing private data, related objects, derivable info).
- Developers design objects through coding (design-while-coding), drawing then coding (UML), or only drawing (tool generates code). Time spent drawing UML should be focused on hard/creative parts early in an iteration, serving as inspiration.
- Design results include **Dynamic models** (logic/behaviors, UML interaction diagrams like sequence and communication diagrams) and **Static models** (packages, classes, attributes, methods, UML class diagrams). Guidelines suggest spending significant time on interaction diagrams before static models, applying RDD and GRASP.
- **UML Interaction Diagrams** illustrate how objects interact via messages. **Sequence diagrams** show interactions in a fence format, good for seeing call flow, with better tool support. **Communication diagrams** show interactions in a graph format, more space-efficient, easier for wall sketches.

- **Responsibility-Driven Design (RDD)** focuses on assigning responsibilities to collaborating objects. Key questions are: What are an object's responsibilities? Who does it collaborate with? What design patterns apply?.
- **GRASP** (General Responsibility Assignment Software Principles) provides principles (patterns) to guide responsibility assignment choices.
    - **Creator**: Assign class B responsibility to create A if B contains/aggregates/records/closely uses/has initializing data for A. Containers/recorders are usually creators, but complex creation may use Factory patterns.
    - **Information Expert**: Assign responsibility to the class with the information needed to fulfill it. A 'Sale' object is responsible for knowing its total because it has the information.
    - **Low Coupling**: Assign responsibilities to minimize dependencies between classes.
    - **Controller**: Assign responsibility for system events (e.g., from GUI) to a class representing the overall system (Facade Controller) or a handler for a use case (Use-case Controller). Avoid "bloated" controllers with too many responsibilities.
    - **High Cohesion**: Assign responsibilities to keep objects focused and manageable, avoiding too much data and operations in one class. A class with high cohesion has a small number of methods with highly related functionality and doesn't do too much work. Benefits include clear separation of concerns and easier maintenance.

9. Design Class Diagrams (DCDs)

- **Design Class Diagrams (DCDs)** describe the type information, accessibility, visibility, attributes, and methods of classes, providing a map to code.
- DCDs differ from conceptual class diagrams (domain model) by including **types**, **directed associations** with multiplicities, numbered actions (often implicit from interaction diagrams), and providing **visibility** between objects.
- DCD elements include **Type Information** for attributes and method parameters/returns.
- **Accessibility** is shown with symbols: + for public, - for private, # for protected. Fields are usually private with getters/setters.
- **Visibility** means if object A sends a message to object B, A must have access to a reference to B.
- While DCDs describe classes, interactions described in interaction diagrams map to the **method bodies**.
- **Mapping design to code**: DCDs map to classes in code (names, methods, attributes, superclasses, associations). Tools can automate this. Associations like one-to-many or many-to-many are typically mapped using collections (e.g., Lists or ArrayLists) in code.

10. Database Design

- A **database** stores data and allows creating, reading, updating, and deleting it. They are critical for non-trivial applications, and poor design can lead to corrupted data or poor performance.
- A **Relational Database** is a collection of tables. Each table has rows (records) and columns (fields), with a unique key for each row. Each entity type typically has its own table.
- **Entity-Relationship (ER) Models** or diagrams are similar to semantic object models (class diagrams) but use different notations and focus more on relations.
- In ER models, an **Entity** is similar to a semantic object and includes attributes. A **Relationship** between entities is indicated with a diamond. **Cardinality** numbers describe the quantitative nature of a relationship (e.g., 1..N). **Inheritance** ("IsA") is represented by a triangle.
- **Mapping Class Diagrams to Tables**: Classes generally map to tables. Attributes map to columns. A **primary key** is often explicitly added to distinguish data in tables. Associations map to foreign keys (for one-to-one or one-to-many) or join tables (for many-to-many).

11. UI Design

- **User Interface (UI)** is the communication medium between a human and a computer. **UI Design** identifies interface objects/actions and creates a screen layout based on design principles. The goal is for the interface to be easy to understand, learn, and use.
- Primary UI styles include **Direct manipulation**, **Menu selection**, **Form fill-in**, **Command Prompt**, and **Natural Language**.
- Typical UI design errors include lack of consistency, too much memorization, no guidance/help, no context sensitivity, poor response, and being arcane/unfriendly.
- **Mandel's Three Golden Rules**: **Place the user in control**, **Reduce the user's memory load**, and **Make the interface consistent**.
- **Place the User in Control**: Avoid forcing unnecessary actions, provide flexible interaction, allow interruptible/undoable actions, allow customization for skill levels, hide technical details.
- **Reduce the User's Memory Load**: Reduce demand on short-term memory (autofill, SSO), establish meaningful defaults, define intuitive shortcuts, use real-world metaphors, disclose information in stages.
- **Make Interface Consistent**: Allow understanding tasks in context (titles, icons, color), maintain consistency across products, avoid changing user expectations from past models.
- **Schneiderman's Eight Golden Rules of Dialog Design** include striving for consistency, enabling shortcuts for frequent users, offering informative feedback, designing for closure, offering simple error handling (making serious errors impossible), permitting easy reversal, supporting internal locus of control, and reducing short-term memory load.
- Different **Kinds of Users** (Novice, Knowledgeable intermittent, Frequent) have different needs that design should address.
- Methods for **Getting User Attention** should be used sparingly, including intensity (bold), marking (underline), size, font choice, blinking, inverse video, color, and audio.
- **Error Handling** should describe the problem clearly/non-judgmentally, provide constructive advice, indicate consequences, and use visual/audio cues.
- A **Help Facility** should be easy to request, presented effectively (window, suggestion), allow easy return to normal mode, and be organized (flat or structured).
- Effective UI design requires understanding user needs (types, tasks/use-cases) and offering interactions that fit those requirements.
- **Accessibility** is a critical requirement (legal in EU/US) that benefits a wider range of users than typically assumed and influences design and testing.

12. Design Patterns

- A **Design Pattern** is a **named general reusable solution** to common design problems. Major sources include the GoF book.
- Patterns can be classified by purpose: **Creational** (object creation), **Structural** (composition of classes/objects), and **Behavioral** (interaction).
- The **Adapter Pattern** (Structural) solves the problem of needing to use an existing class with an incompatible interface by creating an adapter that wraps the existing class and provides the desired interface. **Pluggable Adapters** involve defining a narrow interface in anticipation of future adaptation.
- The **Factory Pattern** (Creational) solves the problem of having many ways to create certain objects by creating a framework responsible for creation, hiding details from the client. A **Factory Method** is an alternative where the creation logic is abstracted into a method implemented by subclasses.
- The **Iterator Pattern** provides a way to access elements of an aggregate object sequentially without exposing its underlying structure.
- The **Composite Design Pattern** allows treating individual objects and compositions of objects uniformly.

13. Software Testing

- **Software Testing** is exercising a program to find errors before delivery. It shows errors, requirements conformance, performance, and indicates quality.
- **Verification** ("building the product right?") ensures software correctly implements a function. **Validation** ("building the right product?") ensures built software traces to customer requirements.

- Software can be tested by the developer (understands system, tests gently, delivery-driven) or an independent tester (learns system, tries to break it, quality-driven).
- General testing criteria include **Interface integrity**, **Functional validity**, **Information content**, and **Performance**.
- Testing strategies progress from **testing-in-the-small** (module/class) to **testing-in-the-large** (integration). For OO software, the class is the initial focus, followed by integration of classes via communication/collaboration.
- **Unit Testing** is verification on the smallest unit (module or class). It tests interfaces, local data structures, boundary conditions, and independent paths (aiming for statement coverage).
- **Integration Testing** uncovers errors associated with interfacing between units. Incremental methods include **Top-down** (test top module with stubs, replace stubs) and **Bottom-up** (group worker modules, use drivers, integrate builds).
- **Regression Testing** reruns executed tests after changes to ensure no unintended side effects.
- **Smoke Testing** is a daily integration test for time-critical projects that exercises the entire system end-to-end to expose major ("show stopper") errors.
- **OO Testing Strategies**: Unit testing maps to **class testing**. Integration testing maps to **Thread-based** (classes for one input/event), **Use-based** (classes for one use case), or **Cluster testing** (classes for one collaboration).
- **Validation Testing** checks if software functions as expected by customers, ensuring requirements are met and documentation is correct.
- **Acceptance tests** include **Alpha testing** (by users at the developer site) and **Beta testing** (by users at their own sites without developers).
- **System Testing** is a series of tests exercising the integrated system. Types include **Recovery testing** (verify recovery from failure), **Security testing** (check protection mechanisms), **Stress testing** (demand abnormal resources), **Performance testing** (test run-time performance), and **Deployment (Configuration) testing** (ensure software works in target environments).
- Stopping testing is often dictated by running out of time, though metrics like error discovery rate can inform the decision.
- **Debugging** involves organizing error data to isolate causes and testing hypotheses. The "Brute Force" method (recording extensive information) can work but is time-consuming and yields too much data.

14. Testing Approaches (Methods)

- A **"Good" test** has a high probability of finding an error, is not redundant, is "best of breed" (high likelihood of uncovering errors), and is neither too simple nor too complex.
- Software can be tested from an **Internal view** (knowing workings, White-box) or an **External view** (knowing specification, Black-box).
- **White-box testing** uses the internal view to ensure all statements and conditions are executed. Understanding execution paths is key because logic errors are inversely proportional to a path's execution probability.
- A **Control Flow Graph (CFG)** represents all possible paths through a program. A **Data Flow Graph** represents the flow of data.
- **Exhaustive testing** (enumerating all paths) is impractical due to the sheer number of possibilities. **Selective testing** is used instead.
- **Control flow-based testing** includes **Basis path testing**, **Condition testing**, and **Loop testing**.
- **Basis Path Testing** aims to execute all independent paths through a program to guarantee **statement coverage** (every statement executed at least once). **Cyclomatic complexity V(G)** measures the structural complexity of a program and indicates the number of independent paths. Higher V(G) correlates with higher error probability. Test cases are designed to cover these paths.
- **Condition Testing** aims to guarantee every branch of predicate nodes is covered (**branch coverage**). Branches include true/false conditions of IFs, loop conditions, and SWITCH alternatives. Branch coverage implies statement coverage, but not vice versa.
- **Loop Testing** involves strategies for simple loops (zero pass, one pass, min/max/typical iterations, max/min +/- 1), nested loops (fixing outer, testing inner), and concatenated loops. Unstructured loops should ideally be redesigned.
- **Data flow-based testing** tests connections between variable definitions (D) and uses (U). Key terms are **DU pair** (definition and use of a variable) and **DU path** (definition-clear path from D to U). Test cases can aim to cover all DU pairs (All-uses), all DU paths, or all definitions (All-def).
- **Black-box testing** focuses on the software's functional requirements using the external view. Testers devise input conditions to exercise all functional requirements. It complements white-box testing ("doing the right thing" vs. "doing things rightly") and is typically applied in later stages.
- Black-box methods include **Equivalence partition** and **Boundary value analysis**.
- **Equivalence partition** divides the input domain into classes where the software should behave equivalently, testing with values from different classes. Classes are defined based on input conditions like ranges, specific values, or sets.
- **Boundary value analysis** complements equivalence partition by focusing tests on the boundary values of equivalence classes and deriving tests from the output domain. For a range [a,b], test cases use values a, b, and just above/below a/b. This also applies to internal data structures at their boundaries.

Good luck with your exam!