**Midterm II Study Guide**

**Table of Contents:**

* Lecture 6 is on pages 1 – 4
* Lecture 7 is on pages 4 – 7
* Lecture 8 is on pages 7 – 9
* Lecture 9 is on pages 9 – 10
* Lecture 10 is on pages 11 – 15
* Lecture 11 is on pages 15 – 20

**Lecture 6 – Application Analysis:**

* Steps for constructing an Application Interaction Model: Determine the system boundary, find actors, find use cases, find initial and final events, prepare normal scenarios, add variation and exception scenarios, find external events, & organize actors and use cases
  + Determining the System Boundary:
    - Determine the purpose of the system
    - The boundary of the system must be known to specify functionality
  + Finding Actors: After determining the system boundary, determine external objects that interact directly with the system
    - Actors include humans, external devices, and software systems
  + Finding Use Cases: For each actor, list the fundamentally different ways in which the actor uses the system
  + Find Initial & Final Events: Find initial events (what happens at the start) and final events (what happens at the end) for each use case found in the last step.
  + Prepare Normal Scenarios: Create regular scenarios (a sequential list of actions between actors and the system during a use case) for each use case found in a previous step.
  + Adding Variations & Exception Scenarios: Error handling is most often the difficult part of developing software; these scenarios (for each use case) will highlight all of the different interactions that can be done with the system for each use case.
  + Find External Events: Examine the scenarios to find all external events – include all inputs, decisions, interrupts, & interactions to or from users or external devices.
    - Event: A transmittal of information to an object.
  + Organize Actors & Use Cases: Organize all actors & use cases (possibly using a Use Case Diagram) to demonstrate how each actor can interact with each use case.
* Application Class Model:
  + Defines the application itself, rather than the real-world objects that the application acts on.
  + Steps for Constructing One: Specify interface, define boundary classes, determine controllers, & check against the interaction model.
    - Specify Interface: Treat the interface at a coarse level of detail; do not consider implementation or platform specific details
    - Identifying Boundary Objects: Do not model the visual aspects, use mock-ups; always use end-user terms to describe boundary objects
      * Identify user interface controls that the user needs to initiate a use case (e.g., ReportEmergencyButton).
      * Identify forms the user needs to enter data into the system (e.g., EmergencyReportForm).
      * Identify notices and messages the system uses to respond to the user (e.g.,AcknowledgementNotice).
      * When multiple actors are involved in a use case, identify actor terminals to refer to the user interface under consideration.
    - Identify Controller Objects: Control objects are responsible for coordinating domain (entity) and boundary objects.
      * Often a close relation exists between a use case and a control object.
      * Controller objects are often created at the beginning of a use case and ceases to exist at its end.
      * Heuristics:
        + Identify one control object per use case.
        + Identify one control object per actor in the use case.
        + The life span of a control object should cover the extent of the use case.
    - Checking Against the Interaction Model: Go over the use cases and think about how they would work; when the domain and application class models are in place, you should be able to simulate a use case with the classes.
  + Heuristics for Detailed Sequence Diagrams:
    - The first column should correspond to the actor who initiated the use case.
    - The second column should be a boundary object that the actor used to initiate the use case.
    - The third column should be the control object that manages the rest of the use case.
    - Control objects are created by boundary objects initiating the use case.
    - Boundary objects are created by controller objects.
    - Entity objects are accessed by control and boundary objects.
    - The access from entity objects to boundary or control objects should be minimized.
* Application State Model:
  + Steps for Constructing One: Determine the application classes with state, find events, build state diagrams, check against other state diagrams, check against the class model, & check against the interaction model.
    - Application Classes with State:
      * Boundary objects lack state dependent behavior.
      * Controllers have important states.
      * Choose a class and consider a sequence diagram.
      * Arrange the events involving the class into a path whose arcs are labeled by events.
      * Find loops within the diagram.
      * Merge other sequence diagrams & find in each sequence diagram where it diverges from previous ones.
      * Add variation or exceptional conditions.
    - Finding Events: Study the scenarios from the application interaction model & extract events.
    - Build State Diagrams: Build a state diagram for each application class with temporal behavior.
      * Choose an application class and consider a sequence diagram for it. Arrange the events involving the class into a path whose arcs are labeled by the events.
        + The interval between any two states is a state; give each state a name.
      * Merge other sequence diagrams into the state diagram; every scenario or sequence diagram corresponds to a path through the state diagram.
      * If a sequence of events can be repeated indefinitely, then they form a loop. At least one state in a loop must have multiple transactions leaving it or the loop will never terminate.
      * Partition a state diagram into two concurrent subdiagrams, using one subdiagram for the main line and the other for the distinguishing information.
      * After normal events have been considered, add variation & exception cases.
      * The state diagram for a class is finished when the diagram covers all scenarios & the diagram handles all events that can affect a state.
    - Check Against Other State Diagrams: Check the state diagrams of each class for completeness & consistency; every event should have a sender & a receiver.
    - Check Against the Class Model: Make sure that the state diagrams are consistent with the domain & application class models.
    - Check Against the Interaction Model: When the state model is ready, go back & check it against the scenarios of the interaction model.
      * If an error is discovered, change either the state diagram or the scenarios; don’t assume that the scenarios are always correct.
* Summary:
  + Purpose of Analysis: Understand the problem so that correct design can be constructed.
  + Good Analysis: Captures the essential features of the problem domain without introducing implementation artifacts.
  + Implementation artifacts prematurely limits design decisions.
  + Two Phases of Analysis: Domain & Application.
  + Domain analysis involves class & state models, but it has seldom interaction models.
  + Application analysis focuses on major application artifacts that are important, visible to users, & must be approved by them.
  + The interaction model dominates application analysis.
  + Application Interaction Model shows the interaction between the system and outside world (High-level SSD) & the extension of high-level SSD with interactions among system objects (entity, boundary, controller).
  + For application analysis, determine system boundary, define actors, define use cases, & make up scenarios for normal cases, variations, extreme cases, & exceptions for each use case.
  + Augment the domain classes with application classes.
    - Application classes arise from boundary, interface, & controller classes.
    - Carefully check the use cases and scenarios to find them.
  + Last phase of application analysis is to build an application state model.

**Lecture 7 – Architectural Design:**

* Architectural Design:
  + An early stage of the system design process.
  + Represents the link between specification & design processes.
  + Often carried out in parallel with some specification activities.
  + It involves identifying major system components & their communications.
  + Process:
    - System Structuring: The system is decomposed into several principal sub-systems & communications between these sub-systems are identified.
    - Control Modelling: A model of the control relationships between the different parts of the system is established.
    - Modular Decomposition: The identified sub-systems are decomposed into modules.
  + Subsystems & Modules:
    - A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
    - A module is a system component that provides services to other components but would not normally be considered as a separate system.
  + Architectural Models:
    - Static Structural Model: Shows the major system components.
    - Dynamic Process Model: Shows the process structure of the system.
  + Architectural Styles:
    - The architectural model of a system may conform to a generic architectural model or style.
    - An awareness of these styles can simplify the problem of defining system architectures.
    - Most large systems are heterogeneous and do not follow a single architectural style.
  + System Structuring: Concerned with decomposing the system into interacting sub-systems
    - The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
  + Application Types:
    - Data Processing Applications: Data driven applications that process data in batches without explicit user intervention during the processing.
    - Transaction Processing Applications: Data-centered applications that process user requests and update information in a system database. (E-commerce systems, reservation systems, etc.)
    - Event Processing Systems: Applications where system actions depend on interpreting events from the system’s environment.
    - Language Processing Systems: Applications where the users’ intentions are specified in a formal language that is processed & interpreted by the system. (Compilers, command interpreters, etc.)
    - Two very widely used generic application architectures are Transaction Processing Systems & Language Processing Systems.
  + The Repository Model:
    - Sub-systems must exchange data. This may be done in two ways:
      * Shared data is held in a central database or repository & may be accessed by all sub-systems.
      * Each sub-system maintains its own database & passes data explicitly to other sub-systems.
    - When large amounts of data are to be shared, the repository model of sharing is most commonly used.
    - Characteristics:
      * Advantages:
        + Efficient way to share large amounts of data.
        + Sub-systems need not be concerned with how data is produced; Centralised management (e.g., backup, security, etc.).
        + Sharing model is published as the repository schema.
      * Disadvantages:
        + Sub-systems must agree on a repository data model. Inevitably a compromise.
        + Data evolution is difficult & expensive.
        + No scope for specific management policies.
        + Difficult to distribute efficiently.
  + Client-Server Architecture:
    - Distributed system model which shows how data & processing is distributed across a range of components.
    - Set of stand-alone servers which provide specific services such as printing, data management, etc.
    - Set of clients which call on these services.
    - Network which allows clients to access servers.
    - Characteristics:
      * Advantages:
        + Distribution of data is straightforward.
        + Makes effective use of networked systems. May require cheaper hardware.
        + Easy to add new servers or upgrade existing servers.
      * Disadvantages:
        + No shared data model so sub-systems use different data organization. Data interchange may be inefficient.
        + Redundant management in each server.
        + No central register of names and services – it may be hard to find out what servers & services are available.
  + Abstract Machine Model:
    - Used to model the interfacing of sub-systems.
    - Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
    - Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
    - However, often difficult to structure systems in this way.
* Control Models:
  + Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.
  + Centralized Control: One sub-system has overall responsibility for control & starts & stops other sub-systems.
    - Call-return Model: Top-down subroutine model where control starts at the top of a subroutine hierarchy & moves downwards. Applicable to sequential systems.
    - Manager Model: Applicable to concurrent systems. One system component controls the stopping, starting and co-ordination of other system processes. Can be implemented in sequential systems as a case statement.
  + Event-based Control: Each sub-system can respond to externally generated events from other sub-systems or the system’s environment.
    - Two Principal Event-driven Models:
      * Broadcast Models: An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
        + Effective in integrating sub-systems on different computers in a network.
        + Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event.
        + Control policy is not embedded in the event & message handler. Sub-systems decide on events of interest to them.
        + However, sub-systems don’t know if or when an event will be handled.
      * Interrupt-driven Models: Used in real-time systems where interrupts are detected by an interrupt handler & passed to some other component for processing.
        + Used in real-time systems where fast response to an event is essential.
        + There are known interrupt types with a handler defined for each type.
        + Each type is associated with a memory location & a hardware switch causes transfer to its handler.
        + Allows fast response but complex to program & difficult to validate.
    - Other event driven models include spreadsheets & production systems.
* Domain-specific Architectures:
  + Architectural models which are specific to some application domain.
  + Two Types of Domain-specific Models:
    - Generic Models: Abstractions from a number of real systems & which encapsulate the principal characteristics of these systems.
    - Reference Models: More abstract, idealized models. Provide a means of information about that class of system & of comparing different architectures.
  + Generic models are usually bottom-up models; reference models are top-down models.

**Lecture 8 – Interaction & Class Design:**

* Transition to Design:
  + During requirements & analysis work, we…
    - “Do the right thing”.
    - Understand the domain.
    - Clarify & record the constraints & requirements.
    - Essentially ignore thinking about the design, & focus on understanding the problem.
  + During design, we…
    - “Do the thing right”.
    - Create a software (& hardware) solution that meets the wishes of the stakeholders.
  + A set of requirements-oriented artifacts (& thought) inspire design-oriented artifacts.
* Interaction Diagrams:
  + Collaboration diagrams show object interactions in a graph or network format.
    - Basic Notation: Links, messages, message number sequencing, conditional messages, & iteration or looping.
  + Sequence diagrams illustrate interactions such that object lifetimes are emphasized (in a kind of fence format).
    - Basic Notation: Links, messages (illustrating returns & creation of instances), object lifelines & object destruction, conditional messages, & iteration.
* Designing Objects with Responsibilities:
  + Responsibility-Driven Design (RDD):
    - Detailed object design is usually done from the point of view of the metaphor of:
      * Objects have responsibilities.
      * Objects collaborate.
      * Similar to how we conceive people.
    - In RDD we do object design such that we will ask questions such as:
      * What are the responsibilities of this object?
      * Who does it collaborate with?
    - Responsibilities: An abstraction; they are implemented with methods in objects.
    - The responsibility for persistence (large-grained responsibility) & the responsibility for the sales tax calculation (more fine-grained responsibility).
* GRASP Patterns:
  + GRASP stands for General Responsibility Assignment Software Patterns.
  + Very fundamental, simple, basic principles of object design.
  + The 9 GRASP Patterns:
    - Expert: Assign a responsibility to the object that has the information necessary to fulfill it.
    - Creator: Choose an object C, such that C contains or aggregates X, C closely uses X, & C has the initializing data for X.
    - Controller: Receives requests for work from the UI layer
      * Candidates:
        + An object whose name reflects the use case.
        + An object whose name reflects the overall server, business, or large-scale entity.
    - Low Coupling: Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
      * Assign responsibilities so that coupling remains low.
      * A class with high coupling suffers from the following problems:
        + Changes in related classes force local changes.
        + Harder to understand in isolation.
        + Harder to reuse.
    - High Cohesion: Cohesion is a measure of how strongly related & focused the responsibilities of an element are. An element with highly related responsibilities, & which does not do tremendous amount of work, has high cohesion.
      * Assign responsibilities so that cohesion remains high.
      * A class with low cohesion is…
        + Hard to comprehend, reuse, & maintain.
        + Delicate; constantly effected by change.
    - Polymorphism: Assign a polymorphic operation to the family of classes for which the cases vary; don’t use case logic.
    - Pure Fabrication: Make up an “artificial” class, whose name is not necessarily inspired by the domain vocabulary.
    - Indirection: A common mechanism to reduce coupling. Assign a responsibility to an intermediate object to decouple collaboration from 2 other objects.
    - Don’t Talk to Strangers: Don’t traverse a network of object connections in order to invoke an operation; rather, promote that operation to a “familiar” of the client.
* Class Visibility:
  + Attribute Visibility: Exists from class A to B when B is an attribute of A.
  + Parameter Visibility: Exists from class A to B when B is passed as a parameter to a method of A.
  + Local Visibility: Exists from class A to B when B is declared as a local object within a method of A.
  + Global Visibility: Exists from class A to B when B is global to A.
* Design Class Diagrams (DCDs):
  + When to Create:
    - In our presentation, DCDs follow the creation of interaction diagrams.
    - Yet, in practice they can be created in parallel, as long as they are in synch.
    - It is possible & desirable to do a little interaction diagramming, then create DCDs, then extend the interaction diagrams some more, & so on.
  + A DCD illustrates the specifications for software classes & interfaces in an application.
    - Classes, associations, attributes.
    - Interfaces with their operations & constraints (OCL).
    - Methods.
    - Attribute type information.
    - Navigability.
    - Dependencies.
  + Steps in DCD Generation:
    - Step 1: Identify software classes & illustrate them.
    - Step 2: Add method names.
    - Step 3: Add method & class constraints using OCL.
    - Step 4: Add associations & navigation information.
* Implementation Models:
  + Realizing associations using visibility information constitutes the basis of design class diagram development.
  + Operations of classes are identified using messages represented in the interaction diagram.
  + Class Design: Requires three major steps: interface design, state model design, & method logic design (which is pseudo code or activity diagram).
  + Interface Design:
    - Contracts: Constraints on a class that enable class users, implementers, & extenders to share the same assumptions about the class. Contracts include three types of constraints:
      * Invariant: Predicate that is always true for all instances of a class. They are used to specify constraints among attributes of a class.
      * Precondition: A predicate that must be true before an operation is invoked (useful for class user).
      * Postcondition: A predicate that must be true after an operation is invoked (useful for class developers & extenders).

**Lecture 9 – Creational Design Patterns:**

* Patterns: Present solutions to common software problems arising within a certain context.
  + Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs.
  + Pattern Languages:
    - Define a vocabulary for talking about software development problems.
    - Provide a process for the orderly resolution of these problems.
    - Help to generate & reuse software architectures.
  + Frameworks: An integrated collection of components that collaborate to produce a reusable architecture for a family of related applications.
* Creational Patterns:
  + Abstract Factory:
    - Intent: Provide an interface for creating families of related objects.
    - Applicability: Use the Abstract Factory pattern when…
      * A system should be independent of how its products are created, composed, & represented.
      * A system should be configured with one of multiple families of products.
      * A family of related product objects is designed to be used together, & you need to enforce this constraint.
      * You want to provide a class library of products, & you want to reveal just their interfaces, not their implementations.
  + Builder:
    - Intent: Create composite objects of different representations.
    - Applicability: Use the builder pattern when…
      * The algorithm for creating a complex object should be independent of the parts that makes up the object & how they assembled.
      * The construction process must allow different representations for the object that’s constructed.
  + Factory Method:
    - Intent: Define an interface for creating an object, but defer instantiation to subclasses.
    - Applicability: Use the Factory Method pattern when…
      * A class can’t anticipate the class of objects it must create.
      * A class wants its subclasses to specify the objects it creates.
      * Classes delegate responsibility to one of several helper subclasses, & you want to localize the knowledge of which helper subclass is the delegate.
  + Prototype:
    - Intent: Create new objects by copying a prototype.
    - Applicability: Use the Prototype when…
      * A system should be independent of how its products are created, composed, & represented.
      * The classes to instantiate are specified at run-time, for example, by dynamic loading.
      * To avoid building a class hierarchy of factories that parallels the class hierarchy of products.
      * Instances of a class can have one of only a few different combinations of states.
  + Singleton:
    - Intent: Ensure a class has only one instance.
    - Applicability: Use the Singleton pattern when…
      * There must be exactly one instance of a class, & it must be accessible to client from a well-known access point.
      * When the sole instance should be extensible by subclassing, & clients should be able to use an extended instance without modifying their code.

**Lecture 10 – Structural Design Patterns:**

* Adapter:
  + Intent: Convert the interface of a class into another interface that clients expect.
  + Motivation: Suppose your organization has a useful class Customer with operations getName() & addAccount().
    - The application you are building is filled with references to an abstract interface called Cust, with operations name() & addAcc() etc., & a concrete subclass BankCust.
    - You want to be able to use Customer without changing the application you are building.
  + Applicability:
    - To use an existing class, & its interface does not match the one you need.
    - To create a reusable class the cooperates with classes that don’t have compatible interfaces.
    - (Object adapter only) To use several existing subclasses, but it can be impractical to adapt their interfaces by subclassing every one. An object adapter can adapt the interface of its parent.
  + Issues:
    - Class Adapter Consequences:
      * Lets Adapter override some of the Adaptee’s behavior, since Adapter is a subclass of Adaptee.
      * Introduces only one object, & no additional pointer indirection is needed to get to the Adapter.
      * Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won’t work when we want to adapt a class & all its subclasses.
    - Object Adapter Consequences:
      * Lets a single Adapter work with many Adaptee’s – i.e., the Adaptee itself & all of its subclasses.
      * Makes it harder to override Adaptee’s behavior.
    - Other Issues:
      * How much adapting should an Adapter do? It depends on how similar the Target interface is to Adaptee’s.
      * Using two-way adapters – making two separate interfaces work as both Adapter & Adaptee.
      * Pluggable interface – design an interface for adaptation. This is similar to the idea of Open Implementation. This can be designed using Adapter design pattern.
* Bridge:
  + Intent: Decouple an abstraction from an implementation so that the two can vary independently.
  + Motivation: Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend & reuse abstractions & implementations independently.
    - It is inconvenient to extend the window abstraction to cover different kinds of windows or new platform.
    - It makes the client-code platform dependent.
  + Applicability:
    - To avoid a permanent binding between an abstraction & its implementation. Specific implementation can be selected at run-time.
    - To independently extend abstraction & implementation by subclassing. Different abstractions can be combined with different implementations.
    - Changes in the implementation of an abstraction should have no impact on the clients.
    - To avoid proliferation of classes as shown in the Motivation section.
    - To share an implementation among multiple objects (e.g., Handle/Body from Coplien).
  + Consequences:
    - Decoupling interface & implementation.
    - Improved extensibility of both abstraction & implementation class hierarchies.
    - Hiding implementation detail from client.
  + Related Patterns:
    - Similar structure to Adapter but different intent.
    - Abstract Factory pattern can be used with the Bridge for creating objects.
* Composite:
  + Intent: Compose objects into tree structure to represent part-whole hierarchies. Composite lets clients treat individual objects & compositions of objects uniformly.
  + Motivation: Classes for containers (e.g., Rectangle) & primitives (e.g., Line) are designed separately, but in many cases the client wants to treat the container & the primitive identically. The Composite pattern describes how to use recursive compositions so that clients don’t have to make this distinction.
  + Applicability:
    - To represent part-whole hierarchies of objects.
    - To be able to ignore the differences between containers & primitives. Clients will treat all objects in the Composite pattern uniformly.
  + Consequences:
    - Clients can treat composite structures (containers) & individual objects (primitives) uniformly.
    - Composite makes it easier to add new kinds of components (leaf or composite).
    - Can make the design overly general. Sometimes you want the composite to have certain kinds of components.
* Decorator:
  + Intent: Attach additional responsibility to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  + Motivation: Suppose that bank application must deal with many different varieties of customers (having checking only, checking & money-market fund & safe-deposit box etc.). The combinatorial varieties are too much to handle using one big class.
  + Applicability:
    - Too add responsibility to individual objects dynamically & transparently, that is, without affecting other objects.
    - For responsibility that can be withdrawn.
    - When extension by subclassing is impractical.
  + Consequences:
    - More flexibility than static inheritance – responsibilities can be added & removed at run-time.
    - Avoids feature-laden classes high up in the hierarchy. Responsibility is added on demand.
    - Lots of little objects. Decorator may produce a system with lots of little look-alike objects. The objects only differ in the way they are interconnected. These systems can be hard to learn & debug.
  + Related Patterns:
    - Adapter gives an object a completely new interface, whereas the Decorator changes an object’s responsibility but not its interface.
    - Decorator can be used with a Composite to represent a whole-part relationship.
* Façade:
  + Intent: Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.
  + Motivation: To reduce complexity of large systems, we need to structure it into subsystems. A common goal is to reduce dependencies between subsystems. This can be done using Façade, which provides a single well-defined interface for the more general facilities of the subsystem.
  + Applicability:
    - To provide simple interface to a complex subsystem, which is useful for most clients.
    - To reduce the dependencies between the client & the subsystem, or dependencies between various subsystems.
    - To simplify the dependencies between the layers of a subsystem by making them communicate solely through their facades.
  + Consequences:
    - It shields the clients from subsystem components, thereby making the subsystem easier to use.
    - It promotes weak coupling between subsystem & its clients.
      * Components in a subsystems can change without affecting the clients.
      * Porting of subsystems is easier.
    - Simplified interface of the Façade may not be adequate for all clients.
* Flyweight:
  + Intent: Use sharing to support large number of fine-grained objects efficiently.
  + Motivation: Suppose you want to represent a list of strings from the database. Lots of these strings may be duplicates. So we can store these strings in the flyweight pool, & have the list(s) refer to it.
  + Applicability:
    - Use flyweights when all of the following are true:
      * Application uses a large number of objects.
      * Storage costs are high because of the sheer quantity of objects.
      * Most of the extrinsic state can be computed.
      * Many objects exist with sharable intrinsic state.
      * Application does not depend upon object identity.
  + Consequences:
    - Storage Savings due to:
      * Reduction in the number of instances.
      * The amount of intrinsic state.
      * Computed extrinsic state.
    - Run-time costs associated with transferring, finding, and/or computing extrinsic state.
  + Related Patterns: The leaf nodes of a Composite are good candidates for Flyweight.
* Proxy:
  + Intent: Provide a surrogate or placeholder for another object. This placeholder can serve many purposes including enhanced efficiency, easier access & protection from unauthorized access.
  + Motivation: Suppose multiple clients want to access (read/write) the same object concurrently. This can be achieved by locating the object to the server where multiple clients can connect to it simultaneously. But the client code should be transparent to the location of the object. This can be achieved by providing a proxy object on the client which transparently forwards the client’s request to the remote (real) object.
  + Variants:
    - Remote Proxy:
      * Encapsulates & maintains the physical location of the original.
      * It also implements the IPC routines that perform the actual communication with the original.
    - Protection Proxy: Protects the original component from unauthorized access. The proxy checks the access rights of every client.
    - Cache Proxy: Lets multiple local clients share results from a remote component.
    - Synchronization Proxy: Controls multiple client access to the original to implement thread-safety.
    - Counting Proxy: Implements reference counting.
      * It does not help in finding cycles of otherwise isolated components referring to each other.
    - Virtual Proxy (Lazy Construction): Delays the loading of the state (bulk) of an object on-demand (e.g., for an image object). It initially constructs an incomplete object without the bitmap. Bitmap is loaded only on-demand.
    - Firewall Proxy: Checks the outgoing & incoming requests for compliance with the internal security & access policies.
      * It can perform logging and/or caching.
      * It is a potential bottleneck for internet access.
      * User needs proxied version of all internet software (e.g., browser, ftp, etc.).
  + Consequences:
    - Enhanced efficiency & lower cost for remote, virtual & cache proxy.
    - Separation of housekeeping code from functionality.
    - Less efficiency due to indirection.
    - Misuse – Proxy can be a overkill for certain situations (e.g., remote proxy for a read-only system or a client-cache proxy for heavily-updateable system).

**Lecture 11 – Behavioral Design Patterns:**

* Chain of Responsibility:
  + Intent: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects & pass the request along the chain until an object handles it.
  + Motivation: A button on a dialog box may have some specific help, or can pass it to its immediate successor, & so on, until the request gets handled.
  + Applicability:
    - To let more than one object handle the request, & the handler is not known prior to it.
    - To issue a request to one of the several objects without specifying the receiver explicitly.
    - To allow dynamic selection of the receiver.
  + Consequences:
    - Reduced coupling between the sender & the receiver.
    - Responsibilities can be added dynamically by configuring the chain at runtime.
  + Related Patterns: Composite can be used in defining the successor chain.
* Mediator:
  + Intent: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, & it lets you vary their interaction independently.
  + Motivation: In designing dialog box, there can be dependencies between the controls (widgets) of the dialog box as:
    - Empty text field disables a button.
    - Typing text selects an entry in the list-box.
    - These problems can be avoided by putting the collective behavior in the Mediator object.
  + Applicability:
    - To a set of objects that communicate in a well-defined but complex way.
    - To allow for greater reuse of an object, an object which refers to & communicates with many other objects.
    - To easily customize the behavior which is distributed between several classes, without a lot of subclassing.
  + Consequences:
    - It limits subclassing.
    - It decouples colleagues.
    - It simplifies object protocol – replaces many-to-many interactions with one-to-many.
    - It abstracts the interaction of objects.
  + Related Patterns:
    - Differences with Façade:
      * Façade abstracts a subsystem of objects to provide a more convenient interface.
      * Its protocol is unidirectional.
* Iterator:
  + Intent: Provide a way to access the elements of an aggregate object sequentially without exposing the underlying representation.
  + Motivation:
    - You do not want to know the internals of a list.
    - You want to traverse a list in different ways.
    - You may have more than one traversal pending on the same list.
    - You want to change the aggregate class without changing your client code.
  + Applicability:
    - To access an aggregate’s contents without exposing its internal representation.
    - To support multiple simultaneous traversals.
    - To provide a uniform interface for traversing different aggregate structure.
  + Consequences:
    - It supports variation in traversal algorithms.
    - Iterators simplify the aggregate interface.
    - There can be more than one traversal pending on the aggregate.
  + Implementation Issues:
    - External iterator vs. internal iterator.
    - Who defines the traversal algorithms? Iterator vs. aggregate.
    - Implementing mutable, modifiable & synchronized iterators.
    - NullIterator can be used for a composite.
  + Related Patterns:
    - Iterators can be applied to recursive structure such as a Composite.
    - Factory Method is used to create an iterator object.
* Command:
  + Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, & support undoable operations.
  + Motivation: RDBMS systems provide triggers on the CRUD semantics. These triggers can be instances of Command objects. A typical use-case would be:
    - RDBMS will provide the abstract Command class.
    - DBA’s extend Command class to implement triggers.
    - Users fire the triggers while using CRUD semantics.
  + Applicability:
    - To parameterize objects by an action to perform.
    - To specify, queue & execute requests at different times.
    - To support undo, add unExecute() to Command.
    - To support logging so that they can be reapplied at system crash.
    - To model transactions.
  + Consequences:
    - It decouples the object that invokes the operation from the one which knows how to perform it.
    - Command can be extended & manipulated like any other object.
    - Commands can be assembled to form macro-command.
    - It is easy to add new commands without changing any existing classes.
  + Related Patterns:
    - Composite can be used to implement MacroCommands.
    - Prototype can be used to copy a command that has to be placed on the history list.
    - Momento can be used to store the state of the Command object, which can be later used to undo.
* Momento:
  + Intent: Capture & externalize an object’s internal state, so that the object can be restored to its state later.
  + Motivation: You want to undo an executed command. This happens in a RDBMS system, when undo-redo logs are applied after the system terminates abnormally. So at every checkpoint, you want to store the state of the object as a momento, such that it can be later used for undo-redo logs.
  + Participants:
    - Momento stores the internal state of the Originator object.
    - It provides two interfaces:
      * Wide interface (with accessor & mutator) for the Originator.
      * Narrow interface for the Caretaker.
  + Applicability: To store the snapshot of an object, so that it can be restored to the same state later; & a direct interface would expose implementation details & break object’s encapsulation.
  + Consequences:
    - It preserves encapsulation boundary.
    - It simplifies the Originator.
* Observer:
  + Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified & updated automatically.
  + Motivation: Suppose you have a desktop OLAP (on-line analytical processing) application. A window on the application is showing a graph & a table of sales by product, geography & time. You enter the latest figures for sales of the month into the table. These changes should also be reflected in the graph. Observer pattern describes how to establish these relationships.
  + Applicability:
    - When an abstraction has two aspects, the view (observer) aspect, & the document (subject) aspect.
    - To notify other objects (observers) of the change.
      * Number of observers are not known.
      * Subject & observer are to be loosely coupled.
  + Consequences:
    - Abstract coupling between subject & observer.
    - Support for broadcast communication.
  + Implementation Issues:
    - Observing more than one subject: extend update() method to take the Subject as a parameter.
    - Make sure that the Subject’s state is self-consistent before calling notify().
    - Push vs. the pull model for broadcasting other changes.
    - Who triggers the update()?
      * State-setting on Subject.
      * Client calls notify().
    - Specifying modifications of interest explicitly.
    - Encapsulate complex update semantics: Create a Mediator called ChangeManager.
      * It maps subject to its observers.
      * It defines particular update strategy.
      * It updates all dependent observers at the request of a subject.
  + MVC (Model View Controller):
    - Model maps to Subject; View maps to Observer.
    - Controller is new. It has the following collaborations & responsibilities:
      * Every View has a Controller.
      * The user interacts through the Controller.
      * It accepts user input as events.
      * It translates an event for the Model or a display request for the View.
      * It implements the update(), if necessary.
* State:
  + Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  + Motivation: Consider a voice recorder. It responds differently depending on its current state (recording, rewinding, etc.). In the State pattern, this difference in behavior is captured as separate classes.
  + Applicability:
    - When an object behavior depends on its state, & it must change its behavior at runtime depending on the state.
    - To avoid large switch statements for state-specific behavior.
  + Consequences:
    - It localizes state-specific behavior & partitions behavior for several states.
    - It makes state-transition explicit & more atomic.
    - State objects can be shared as Flyweights.
  + Related Patterns:
    - The Flyweight pattern can be used to share state objects.
    - State objects are often Singletons.
* Strategy:
  + Intent: Define a family of algorithms, encapsulate each one, & make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
  + Applicability:
    - When many related classes differ only in their behavior. Strategies provide a way to configure a class with one of the many behaviors.
    - When you need different variants of an algorithm.
    - To avoid exposing complex, algorithm-specific data structures.
    - To move partitions of a conditional statement into its classes.
  + Consequences:
    - Hierarchies of Strategy classes define a family of algorithms or behaviors for Context to reuse.
    - An alternate to subclassing. Encapsulating the behavior in separate Strategy class lets you vary the algorithm independently of its Context, making it easier to switch, understand & extend.
    - Eliminates conditional statements.
* Template Method:
  + Intent: Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.
  + Applicability:
    - To implement the invariant part of an algorithm.
    - When common behavior among subclasses should be factored & localized in a common class to avoid code duplication.
    - To control subclass extension.
  + Consequences:
    - It is a fundamental technique for code reuse.
    - It lets the subclasses refine the (default) behavior of an operation.
  + Related Patterns:
    - Factory Methods are often called by Template Methods.
    - Strategy uses delegation to vary the entire algorithm, while Template Method uses inheritance to vary part of an algorithm.
    - Builder lets ConcreteBuilders vary the part of a construction, while Template Method lets its subclasses vary part of an operation.
* Visitor:
  + Intent: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
  + Applicability:
    - When an object structure contains many classes of objects with different interfaces, & you want to perform operations on objects that depend on their concrete classes.
    - To avoid cluttering of classes. Visitor lets you keep related operations together.
    - When the classes defining the object structure rarely change.
  + Consequences:
    - It makes it easy to add new operations to an object structure.
    - It gathers related operations & separates unrelated ones.
    - Visitor can visit objects that do not have a common parent class.
    - It can accumulate state?
  + Dependency Cycle Problem:
    - The base class of the Element hierarchy depends on the base class of the corresponding Visitor hierarchy.
    - The Visitor base class has methods for each of the derivatives of the Element base class.
    - Each derivative of Element depends on the base Element.
    - Thus there is a cycle of dependencies that causes elements to transitively depend upon all its derivatives. So, a change in derivative hierarchy asks for massive recompilation.
* Acyclic Visitor:
  + Intent: It is to allow new functions to be added to existing class hierarchies without affecting those hierarchies, & without creating the troublesome dependency cycles inherent in GoF Visitor.
  + Applicability:
    - When the Visited class hierarchy can be frequently extended with new derivatives of the Element class.
    - When the recompilation, relinking, retesting & redistribution of derivatives of Element is very expensive.
  + Consequences:
    - Elimination of dependency cycle.
    - Derived elements of an object structure do not depend on each other.
    - Recompilation is minimized.
    - Partial visitation is natural & does not require additional code overhead.
* Default Visitor:
  + Intent: It is to allow new functions to be added to existing class hierarchies without affecting those hierarchies, & without creating the troublesome dependency cycles inherent in GoF Visitor.
  + Applicability:
    - When the elements to be visited come from a small set of polymorphic class hierarchies.
    - When several ConcreteVisitors can employ default handlers for a small set of abstract elements.
  + Consequences:
    - It makes it easier to add new ConcreteElement classes.
    - It provides a base method.

**Midterm 2 Questions:**

1. You are given two subsections – one is an architectural style and the other is a control style.
   1. Name the control style for each architectural style (Repository, etc.).
   2. Make a collaboration diagram for each subsection.
2. Maybe given a diagram (probably a collaboration diagram).
   1. For message x (assuming a diagram is given), tell me why a certain GRASP guideline can address this. If no diagram is presented, explain the given GRASP guideline and what it can be used for.
   2. How do you understand the SOLID principles? How do they enforce the guideline between superclass and subclass?
3. Given a domain class model with associations and attributes, an SSD, and a UML collaboration graph from a message in the SSD (Question with largest weight on the test).
   1. Create a collaboration diagram for the next message in the SSD.
   2. Justify why you do certain things with GRASP guidelines (and possibly assign GRASP guidelines to the resulting collaboration diagram).
   3. Extend the domain class model into a design class model using the two collaboration diagrams.
   4. What is the visibility of certain attributes within the new design class model?

**Info for These Questions:**

* Open/Closed Principle: This is a principle that says “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” – in other words, we should strive to write code that doesn’t have to be changed every time the requirements change. This is part of the SOLID principles.
  + This is necessary to enforce the guideline between superclass and subclass because we need to write code for the superclass so that subclasses can extend certain behavior without us having to modify our code. This way we can uphold the principle and allow reusability within our code. This would also have the benefit of allowing us to create a new subclass if a new problem arises so that our superclass abstract template does not have to be changed.
* Design Class Diagrams (DCDs):
  + When to Create:
    - In our presentation, DCDs follow the creation of interaction diagrams.
    - Yet, in practice they can be created in parallel, as long as they are in synch.
    - It is possible & desirable to do a little interaction diagramming, then create DCDs, then extend the interaction diagrams some more, & so on.
  + A DCD illustrates the specifications for software classes & interfaces in an application.
    - Classes, associations, attributes.
    - Interfaces with their operations & constraints (OCL).
    - Methods.
    - Attribute type information.
    - Navigability.
    - Dependencies.
  + Steps in DCD Generation:
    - Step 1: Identify software classes & illustrate them.
    - Step 2: Add method names.
    - Step 3: Add method & class constraints using OCL.
    - Step 4: Add associations & navigation information.
* GRASP Patterns:
  + GRASP stands for General Responsibility Assignment Software Patterns.
  + Very fundamental, simple, basic principles of object design.
  + The 9 GRASP Patterns:
    - Expert: Assign a responsibility to the object that has the information necessary to fulfill it.
    - Creator: Choose an object C, such that C contains or aggregates X, C closely uses X, & C has the initializing data for X.
    - Controller: Receives requests for work from the UI layer
      * Candidates:
        + An object whose name reflects the use case.
        + An object whose name reflects the overall server, business, or large-scale entity.
    - Low Coupling: Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
      * Assign responsibilities so that coupling remains low.
      * A class with high coupling suffers from the following problems:
        + Changes in related classes force local changes.
        + Harder to understand in isolation.
        + Harder to reuse.
    - High Cohesion: Cohesion is a measure of how strongly related & focused the responsibilities of an element are. An element with highly related responsibilities, & which does not do tremendous amount of work, has high cohesion.
      * Assign responsibilities so that cohesion remains high.
      * A class with low cohesion is…
        + Hard to comprehend, reuse, & maintain.
        + Delicate; constantly effected by change.
    - Polymorphism: Assign a polymorphic operation to the family of classes for which the cases vary; don’t use case logic.
    - Pure Fabrication: Make up an “artificial” class, whose name is not necessarily inspired by the domain vocabulary.
    - Indirection: A common mechanism to reduce coupling. Assign a responsibility to an intermediate object to decouple collaboration from 2 other objects.
    - Don’t Talk to Strangers: Don’t traverse a network of object connections in order to invoke an operation; rather, promote that operation to a “familiar” of the client.
* Class Visibility:
  + Attribute Visibility: Exists from class A to B when B is an attribute of A.
  + Parameter Visibility: Exists from class A to B when B is passed as a parameter to a method of A.
  + Local Visibility: Exists from class A to B when B is declared as a local object within a method of A.
  + Global Visibility: Exists from class A to B when B is global to A.
* Architectural Design:
  + An early stage of the system design process.
  + Represents the link between specification & design processes.
  + Often carried out in parallel with some specification activities.
  + It involves identifying major system components & their communications.
  + Process:
    - System Structuring: The system is decomposed into several principal sub-systems & communications between these sub-systems are identified.
    - Control Modelling: A model of the control relationships between the different parts of the system is established.
    - Modular Decomposition: The identified sub-systems are decomposed into modules.
  + Subsystems & Modules:
    - A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
    - A module is a system component that provides services to other components but would not normally be considered as a separate system.
  + Architectural Models:
    - Static Structural Model: Shows the major system components.
    - Dynamic Process Model: Shows the process structure of the system.
  + Architectural Styles:
    - The architectural model of a system may conform to a generic architectural model or style.
    - An awareness of these styles can simplify the problem of defining system architectures.
    - Most large systems are heterogeneous and do not follow a single architectural style.
  + System Structuring: Concerned with decomposing the system into interacting sub-systems
    - The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
  + Application Types:
    - Data Processing Applications: Data driven applications that process data in batches without explicit user intervention during the processing.
    - Transaction Processing Applications: Data-centered applications that process user requests and update information in a system database. (E-commerce systems, reservation systems, etc.)
    - Event Processing Systems: Applications where system actions depend on interpreting events from the system’s environment.
    - Language Processing Systems: Applications where the users’ intentions are specified in a formal language that is processed & interpreted by the system. (Compilers, command interpreters, etc.)
    - Two very widely used generic application architectures are Transaction Processing Systems & Language Processing Systems.
  + The Repository Model:
    - Sub-systems must exchange data. This may be done in two ways:
      * Shared data is held in a central database or repository & may be accessed by all sub-systems.
      * Each sub-system maintains its own database & passes data explicitly to other sub-systems.
    - When large amounts of data are to be shared, the repository model of sharing is most commonly used.
    - Characteristics:
      * Advantages:
        + Efficient way to share large amounts of data.
        + Sub-systems need not be concerned with how data is produced; Centralised management (e.g., backup, security, etc.).
        + Sharing model is published as the repository schema.
      * Disadvantages:
        + Sub-systems must agree on a repository data model. Inevitably a compromise.
        + Data evolution is difficult & expensive.
        + No scope for specific management policies.
        + Difficult to distribute efficiently.
  + Client-Server Architecture:
    - Distributed system model which shows how data & processing is distributed across a range of components.
    - Set of stand-alone servers which provide specific services such as printing, data management, etc.
    - Set of clients which call on these services.
    - Network which allows clients to access servers.
    - Characteristics:
      * Advantages:
        + Distribution of data is straightforward.
        + Makes effective use of networked systems. May require cheaper hardware.
        + Easy to add new servers or upgrade existing servers.
      * Disadvantages:
        + No shared data model so sub-systems use different data organization. Data interchange may be inefficient.
        + Redundant management in each server.
        + No central register of names and services – it may be hard to find out what servers & services are available.
  + Abstract Machine Model:
    - Used to model the interfacing of sub-systems.
    - Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
    - Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
    - However, often difficult to structure systems in this way.
* Control Models:
  + Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.
  + Centralized Control: One sub-system has overall responsibility for control & starts & stops other sub-systems.
    - Call-return Model: Top-down subroutine model where control starts at the top of a subroutine hierarchy & moves downwards. Applicable to sequential systems.
    - Manager Model: Applicable to concurrent systems. One system component controls the stopping, starting and co-ordination of other system processes. Can be implemented in sequential systems as a case statement.
  + Event-based Control: Each sub-system can respond to externally generated events from other sub-systems or the system’s environment.
    - Two Principal Event-driven Models:
      * Broadcast Models: An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
        + Effective in integrating sub-systems on different computers in a network.
        + Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event.
        + Control policy is not embedded in the event & message handler. Sub-systems decide on events of interest to them.
        + However, sub-systems don’t know if or when an event will be handled.
      * Interrupt-driven Models: Used in real-time systems where interrupts are detected by an interrupt handler & passed to some other component for processing.
        + Used in real-time systems where fast response to an event is essential.
        + There are known interrupt types with a handler defined for each type.
        + Each type is associated with a memory location & a hardware switch causes transfer to its handler.
        + Allows fast response but complex to program & difficult to validate.
    - Other event driven models include spreadsheets & production systems.

**(4-25pts).** Consider the following Collaboration Design Diagram (message numbering is intentionally

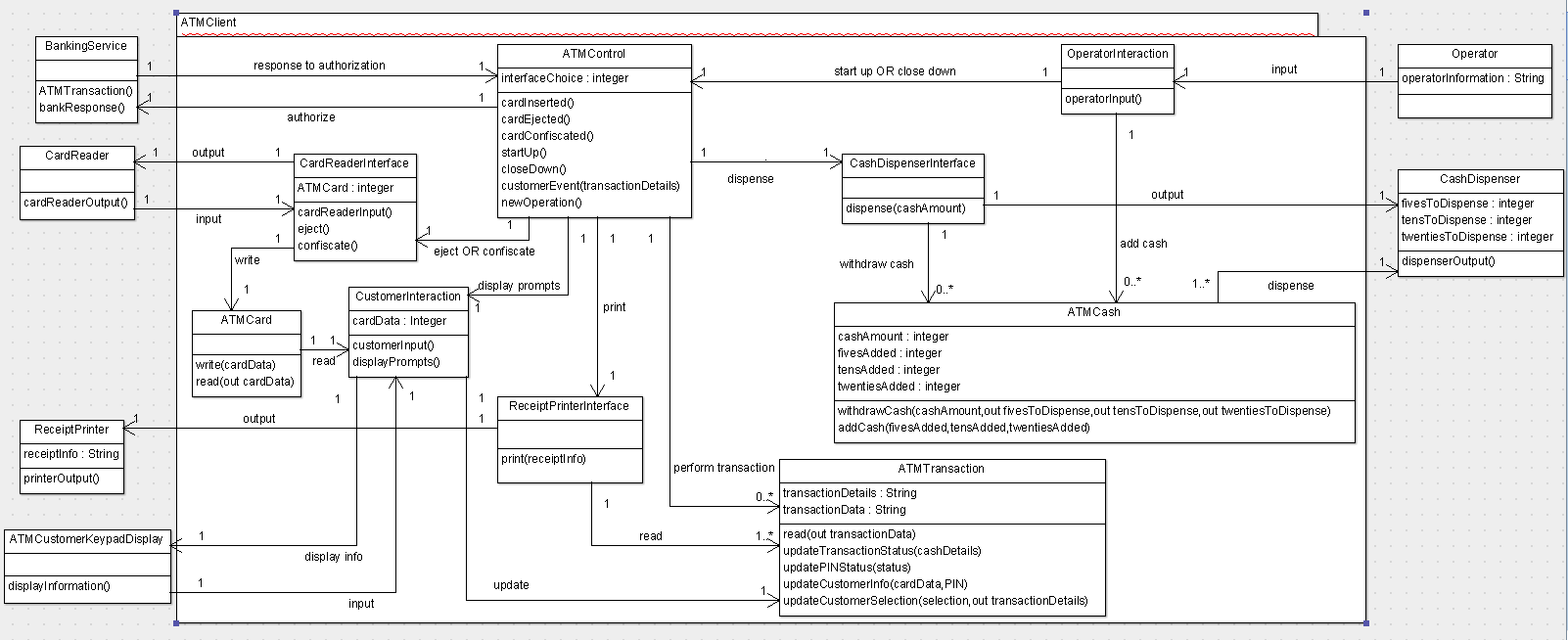
left out)



1. For each message listed in the Collaboration Diagram, suggest one or more GRASP guidelines suitable for justifying the allocation of the responsibility to the recipient object.

* ATM Transaction: Controller, High Cohesion,
* Bank Response: Controller, Indirection, Creator, Pure Fabrication, High Cohesion
* Card Reader Input: Expert, Pure Fabrication, Low Coupling
* Card Reader Output: Expert, Creator
* Customer Input: Expert, Pure Fabrication, Low Coupling
* Display Information: Expert, Creator
* Dispenser Output: Expert, Creator
* Operator Information: Expert, Creator
* Operator Input: Expert, Indirection, Pure Fabrication, Low Coupling
* Printer Output: Expert, Creator
* Dispense (cashAmount): Expert, Indirection, Creator, Pure Fabrication
* withdrawCash (in cashAmount, out fivesToDispense, out tensToDispense, out twentiesToDispense): Expert, Polymorphism, Don’t Talk to Strangers
* startUp, closedown: Don’t Talk to Strangers, Low Coupling
* write (cardData): Expert, Polymorphism, Don’t Talk to Strangers
* cardInserted, cardEjected, cardConfiscated: Don’t Talk to Strangers, High Cohesion
* eject, confiscate: Indirection, Creator, High Cohesion, Low Coupling
* read (out cardData): Expert, Polymorphism, Don’t Talk to Strangers
* customerEvent (transactionDetails): Don’t Talk to Strangers
* addCash (in fivesAdded, in tensAdded, in twentiesAdded): Expert, Polymorphism, Don’t Talk to Strangers
* print (receiptInfo): Expert, Indirection, Pure Fabrication
* displayPrompts: Indirection, Creator, Low Coupling
* updateTransactionStatus (cashDetails), updatePINStatus (status): Expert, Polymorphism, Don’t Talk to Strangers, High Cohesion
* read (out transactionData): Expert, Polymorphism, Don’t Talk to Strangers
* updateCustomerInfo (cardData, PIN), updateCustomerSelection (in selection, out transactionDetails): Expert, Polymorphism, Don’t Talk to Strangers, High Cohesion

1. Draw a UML Design Class Diagram, including classes, methods (based on the messages, which are interpreted as responsibilities of the recipient objects), and direction, as well as type of visibility. Based on your choice of visibility among the associated classes, indicate which dependencies are attribute, local, and parameter visibilities.



**Visibilities:**

* ReceiptPrinterInterface 🡪 ReceiptPrinter: Parameter visibility
* CashDispenserInterface 🡪 ATMCash: Parameter visibility
* ATMCard 🡪 CustomerInteraction: Parameter visibility, Local visibility
* ATMControl 🡪 ATMTransaction: Parameter visibility
* ReceiptPrinterInterface 🡪 ATMTransaction: Local visibility
* ATMCash 🡪 CashDispenser: Local visibility
* CardReaderInterface 🡪 ATMCard: Attribute visibility
* All other dependencies have global visibility.