* **Basic Principles (SOLID)**
* **S = Single Responsibility** - Each class should have a single overriding responsibility (High Cohesion). Each class has one reason why it should change
* **O = Open/Closed** - Objects are open for extension but closed for modification. Extension via inheritance, polymorphism.
* **L = Liskov Substitution** - Subclasses should be substitutable for their base classes. Consider behavior, not just “is a”. Square/rectangle problem. Preconditions cannot be strengthened. Postconditions cannot be weakened.
* **I = Interface Segregation** - Don’t make large multipurpose interfaces – instead use several small focused ones. Don’t make clients depend on interfaces they don’t use. Class should depend on each other through the smallest possible interface.

**D = Dependency Inversion** - Depend on abstractions, not concretions. Program to interfaces not implementations. Program to most abstract class possible. Hollywood Principle

* **OOA vs. OOD**
* In analysis, we are mostly concerned with the DOMAIN MODEL. What are the objects in the domain and how do they collaborate. We do not think about how things are done, only what has to be done.
* In design, we need to integrate an APPLICATION MODEL. What objects do I need to add to get this thing to run on a computer and to be realized in some programming language

**GRASP Design Principles**

**Controller:** the first object beyond the UI layer that is responsible for receiving or handling a system operation message. Assign the responsibility to a class representing one of the following choices: represents the overall “system,” a “root object,” a device that the software is running within, or a major subsystem - these are all variations of a facade controller. Or represents a use case scenario within which the system event occurs, often names <UseCaseName>Handler/Coordinator/Session.

**Creator:** the one responsible for creating a new instance of some class. Assign class B the responsibility to create an instance of class A if one of these is true: B “contains” or compositely aggregates A, B records A, B closely uses A, B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A. B is a creator of A objects.

**Information Expert:** general principle of assigning responsibilities to objects. Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.

**High Cohesion:** how to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling. Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

**Indirection:** where to assign a responsibility, to avoid direct coupling between two (or more) things. Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an indirection between the other components.

**Low Coupling:** how to support low dependency, low change impact, and increased reuse. Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

**Polymorphism:** how to handle alternatives based on type. When related alternatives or behaviors vary by type (class), assign responsibility for the behavior - using polymorphic operations - to the types for which the behavior varies.

**Protected Variations:** how to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements. Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

**Pure Fabrication:** what object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other gorals, but solutions offered by Expert (for example) are not appropriate). Assign a high cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept - something made up, to support high cohesion, low coupling, and reuse.

**Other Design Principles**

**Tell Don’t Ask:** Rather than asking an object for data and acting on that data, we should instead tell an object what to do.

**Law of Demeter:** Each unit should have only limited knowledge of other units: only units “closely” related to the current unit. Each unit should only talk to it’s friend, don’t talk to strangers. Only talk to your immediate friends. A given object should assume as little as possible about the structure or properties of anything else (including it’s subcomponents).

**Open Closed Principle:** States that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. Design should be done in a way to allow the adding of new functionality as new classes. keeping as much as possible existing code unchanged. Software entities like classes, modules, and functions should be open for extension but closed for modifications.

**Design Patterns**

**~Information Expert** - Assign a responsibility to the class that has the information necessary to fulfill the responsibility.

**~Singleton** - want to limit the application to only one instance of a particular class, but need global access to that class. Used to control access to key resources. ToDo: override new, make static accessor method. private constructor. Caveats: anyone has access reduces reuse

**~Factory** - Define an interface for creating an object, but let subclasses decide which class to instantiate. Allows a class to defer instantiation to subclasses.

Think of a multi-document application framework. An application object may know when an object needs to be created, but not which object. How do we create the correct object when needed?

Can also be used when a complex initialization of objects is necessary, for instance when aggregation is heavily used.

Can also be used to take advantage of memory-optimization like object pools, cached objects, etc.

**~Command** - Encapsulate commands in objects, so we can queue them, undo them or make macros.

**~Prototype** - I need to create customized objects without knowing the exact class. “I want something just like this, except with new instance values”...Solution – clone/copy the object and initialize the data.

**~Observer** - Allows objects to dynamically register their interest in being notified of any changes that occur in the state of the observed object.

**~Null Event** - An alternative to using null to indicate the absence of an object to delegate an operation to. Can eliminate a test for null by using an object that doesn’t do anything. This allows objects to always behave the same way, i.e. use an object without any additional tests required.

**~Delegation** - A way to extend and use the functionality of a class by writing an additional class with added functionality that uses instances of the original class to provide original functionality. An alternative to inheritance. Used to handle multiple roles. In Java “event sources” delegate responsibility for processing an event to an “event listener."

**~Strategy** - Allow selection of algorithm to vary by object and time.

**~State Pattern** - Like Strategy, but what if data had to vary

instead of the behavior? Or what if needed to simulate state machine?...Solution: Encapsulate changing data in a class or each state of state machine in a class

**~Template Method** - I have a method, where some steps are common to all subclasses, but

others are specialized....Solution. Create a method with concrete steps, but that calls abstract methods for the specialized parts.

**~Flyweight** - I have a bunch of classes, but I need to minimize the number of objects I am

using. Class has only those attributes shared amongst all instances.

**~Visitor** - I need to apply different operations to a collection of objects. I want to centralize these operations. I want to reduce coupling. For example in a word processor, grammar check, spell check, table of contents builder, outliner all need to traverse the document. Solution: Implement a visitor object that knows how to traverse the object in question (a document in our example). Subclass the visitor for each specialized operation.

**UI Evaluations**

**Cognitive Walkthrough** - Checking how easy your system is to use by a novice user. **Heuristic Evaluation** - Checking your user interface against a set of rules that are generally accepted as good design **Think Aloud** - Having an actual user operate your system and provide you with feedback **Natural Mapping** – (Predictable link between action and consequences) Designing a rotating control so that turning it to the right increases the value **False Affordance** – (affordance - perceived properties, relationship between object- person-interaction, combination of good visibility, natural mapping, constraints, feedback) Giving users a label that looks like a button. **Mental Model** – (Designers and Users can have different models) An idea formed by the user of how your system works; **Recognition** - Putting a button bar on the screen so that the user can pick an action based on what seems needed **Recall** - Having a command prompt where the user enters memorized commands **Constraints** – convey possible / appropriate actions, e.g. red associated with bad, green with good so can’t make a green stop sign **UI Principles**: Predictability (value of previous use of system, operation visibility), Synthesizability (making the mental model), Generalizability (apply knowledge of prior use to future use), Familiarity (Affordance/Guessabiliity), Consistency, Flexibility (multiple ways of interaction), Robustness (achievement of goals e.g. recoverability, responsiveness), Learnability (novice user support), Accessibility (e.g. how accommodating is your UI for disabled people), more rules: recognition rather than recall -> give users choices of actions they can make rather than memorizing, Must assist user, not become a task, must not make user/system appear stupid, user control and freedom, error prevention, visibility of system status

**Model-View-Presenter:** The model is an interface defining the data to be displayed or otherwise acted upon in the user interface. The presenter acts upon the model and the view - it retrieves data from the repositories (the model), and formats it for display in the view. The view is a passive interface that displays data (the model) and rates user commands (events) to the presenter to act upon that data.

**Model:** Information for the application.

**View:** Visual Representation of the Information.

**Controller:** Mechanism to allow user to change information or view.

**Passive MVC –** model is unaware of theview/controller (ie, simple text editor)

**Active MVC** – model actively notifies view that it has changed (changed: and update: methods

**Model -> View** loose coupling through dependents collection (class Model)

**View <-> Controller** Tight coupling through direct reference

**View -> Model** Tight coupling through direct reference

**Controller -> Model** Tight coupling through direct reference

**Model -> Controller** No reference at all

*MVC is the original -- developed by Xerox Parc with Smalltalk.    For a particular UI screen  the triad consists of   Model (data and business logic), View (presentation of information to user) and Controller (handles obtaining and processing user inputs)*

*MVC had some weaknesses like introducing a lot of coupling between the view and the model*

*MVP -- Model is still data/core business logic,  View is still presentation to user.    The difference is in the presenter.   The Presenter talks to the Model and View only through generic interfaces.   The Model and View elements are completely oblivious to the presence of any other components  (as opposed to MVC where the view had to query the model for data, and the model had to notify the view that things were changing.)    This means the presenter handles all user inputs, but also any required coordination between the model and view.*

**STATEMENT VS BRANCH COVERAGE:**

statement coverage - the test cases together ensure that every line of code is executed at least once

branch coverage - statement coverage plus every decision is taken in both the true and false direction.

if (x > 6) do();

statement coverage  x = 89  will execute all statements

branch coverage,   need x = 89 for true direction, but would also need x = 4  for false direction.

**TESTING**

Early in the project – Acceptance tests

During Design – Usability tests

During Implementation – Unit and Component tests

During Integration – Integration tests

During Testing – System and Alpha/Beta tests

Failure: Incorrect behavior of a component – A symptom

Fault (Bug, Defect): Incorrect program or data object – Result of error

Error: Mistake made by human – omission or commission

Whitebox: Use the code to develop tests. Creates tests to execute all lines of code. Creates tests to take each branch direction. Creates tests to find plausible “faults”

Blackbox: Don’t look at code, review the specification of the class

What responsibilities does it have? Contracts? Does it do what it claims?

Plausible faults, equivalence partitions, boundary conditions, edge cases

Behavior Driven Testing: Supported by frameworks such s Cucumber, JBehave, and easy-b. Specify desired behaviors of the software given certain conditions

Test Driven Deployment: Write the Unit tests BEFORE writing the code. Write tests to discover any failures that crop up in operation. Base tests on what the system has to do. Then write code to make the tests pass

**Misc. Questions:**

which design principles are supported by the use of interfaces in MVP:

*Dependency Injection, Pure Fabrication, Indirection, Low Coupling, Open-Closed*

Switch statements in OO code are generally considered to be bad. Which of the following principles tell us to limit their use:

*Polymorphism, Indirection, Open-Closed*

Advantage and disadvantage of singleton:

*Adv: can access throughout the application, DA: create global piece of data, could be problematic for security*

You need the ability to undo player moves back any number of moves. Design pattern most useful is:

*Command*

You want to preserve low coupling when notifying objects of user arriving on planet, best design pattern is:

*Observer*

The most useable systems are those where the system model, the developer’s mental model and the user’s mental model are the same *True*

The best technique to use if you only have random pen and paper drawings of an interface is: *Heuristic*

GRASP Principles supported by classic MVP Paradigm:

*Indirection, protected variation, polymorphism*

**Design Reviews and Validation**

Design Criteria – FURPS: Functionality, usability, reliability, performance, security/safety/sustainability

Traceability – the system shall maintain security log files

**User Interface Design Principles**

* + Create effective mental models
  + Make appropriate functionality visible
  + Natural mappings
  + Use affordances
  + Use constraints
  + Provide feedback
  + Design with errors in mind

UI Principles

Flexibility (multiple ways of interaction)

*Dialogue Initiative (System/User Pre-emption, modal dialogs, User freedom)*

*Multi-threading (Ability to support more than one task)*

*Task migratability (Pass control between user and system)*

*Substitutivity (representation flexibility)*

*Customizability (modifiability of the UI)*

Robustness (Achievement of goals)

*Observability (Browsability, Defaults, Viewing internal state of system)*

*Recoverability (Ability to take corrective action)*

*Responsiveness (communication rate)*

*Task Conformance*

**Exceptions and Contracts (Interface or API)**

Reliability - Probablility that something satisfies its behavior requirements over time and under given conditions. Correctness (does what it’s supposed to do), Robustness (acts acceptably when it can’t do what it supposed to do) Safety - an absence of accidents (an unplanned and unacceptable loss) Reliable but unsafe → Mars Polar lander (misinterpretation of noise on landing leg deployment)

Safe but unreliable → Human operators (procedure deviation) Contract - Defining a precondition and a postcondition for a routine is a way to define a contract that binds the routine and its callers (Example) Operation: + push(x : object) : bool where push is a public instance method of Stack class and x is any instance or subclass of Object. method returns true if successful, false if x not added to stack Preconditions: true (a condition that must be true of parameters before running the method) Postconditions: size’ = size + 1. Original contents unchanged, x is new top of stack. (condition that is true after running the code) Class invariant – what is always true about a class. TimeOfDay class (hours is always between 0 and 24) Exceptions are deviations from the norm that we expect. We try to get the application in a predictable state and continue on. (Incorrect Login) Errors are when things go unexpectedly wrong. Little can be done to recover. (Hard disk failure, Memory Error, Loss of Network Connection) Strategies for errors - Inaction – Do nothing, | Balk – Admit defeat, | Guarded Suspension – Suspend execution until conditions are OK to continue, | Provisional Action – Pretend to perform the action, but don’t commit until conditions are OK. | Recovery – Perform an acceptable alternative. | Rollback – Try to perform, but on failure undo everything. | Retry – Repeatedly attempt, after recovering from failed attempts. | Appeal to higher authority – Allow user to steer application | Resign – Minimize damage, write to log, signal definite and safe failure. R – Runtime : (do nothing/let it die) | E – Exception (catch and handle an exception at throw point) | P – Propogation (allow exceptions to move up system to point where you will handle them) | A – Atomic (transactions – ensure something either happens or it doesn’t) | I – Isolation (keep other processes from seeing changes. i.e. until transaction commits, nothing has happened) | R – Resumption (retry, code tries to “heal” itself)



