Topics:

1. Software architecture (Responsibility-driven design, different control styles)

focuses on the contract by asking:

* What actions is this object responsible for?
* What information does this object share?

Control styles: Centralized (one object is in charge of everything), clustered (a few objects in charge), delegated(passes some of the decision making and much of the action to objects surrounding a control center, preferred!), dispersed (no centers)

1. trust boundaries: A data trust boundary is a point where data comes from an untrusted source.
2. Facade pattern: facade is an object that provides a simplified interface to a larger body of code
3. Command Design Pattern:
   1. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
   2. Promote "invocation of a method on an object" to full object status
   3. An object-oriented callback[s](http://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-its)
4. The **observer pattern** is a [software design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) in which an [object](https://en.wikipedia.org/wiki/Object_(computer_science)#Objects_in_object-oriented_programming), called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their [methods](https://en.wikipedia.org/wiki/Method_(computer_science))
5. MVC, MVP

active/passive mvc

Here is useful link that makes a good distinction: [http://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-it](http://www.codeproject.com/Articles/674959/MVC-Patterns-Active-and-Passive-Model-and-its)

**MVC** : Model = data (not directly related/mapped to what is shown in view, can contain program logic and other stuff);

View = screen (+ how data is shown; );

Controller = program logic (interaction of user with data; updates model, can update view to rep resent the model)

Example - web, click a link, controller redirects you to another page (view)

**MVP** (derivation of MVC)

Model: same as in MVC, but doesn’t usually interact with the view

View: displays data, routs user commands to presenter, gets updated by presenter to display data. May be passive or directly bound to model.

* Separate view for each stakeholder: eg, building → electricians, plumbers, white collar, manager all need different “map” (view)
* 1) Logical/conceptual

Presenter: like a controller, but has more responsibility as middle-man between view and model

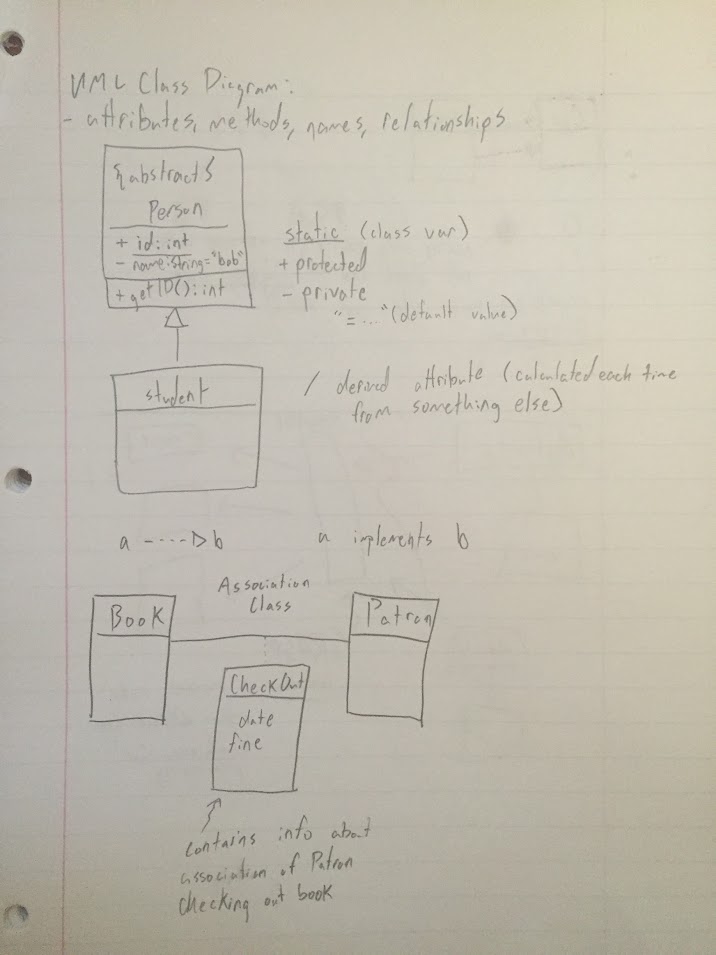
SOLID

* [**Single responsibility principle**](https://en.wikipedia.org/wiki/Single_responsibility_principle) - class/module should be responsible only for 1 part of the functionality! And it should change only if that part of funct. change
* [**Open/closed principle**](https://en.wikipedia.org/wiki/Open/closed_principle) - “software entities … should be open for extension, but closed for modification.” it is good if you inherit a class and add new stuff, but bad if you change its inner data or workings.
* [**Liskov substitution principle**](https://en.wikipedia.org/wiki/Liskov_substitution_principle) - an object of a type that’s a suptype of the expected type should work fine. This is used by polymorphism
* [**Interface segregation principle**](https://en.wikipedia.org/wiki/Interface_segregation_principle) - each interface should have as small a role as possible. Like SRP, but with interfaces. You can implement many of them!
* [**Dependency inversion principle**](https://en.wikipedia.org/wiki/Dependency_inversion_principle) - always better to use interfaces or abstract classes as types for, like, method parameters, so that any implementation/extension can be passed in

GRASP (**General Responsibility Assignment Software Patterns/Principles**):

* Controller: object used for handling events, can be used to deal with all events of 1 or more use-cases.
* Creator: a class B can create instances of class A if 1) B compositionally aggregates A; 2) B stores instances of A; 3) B closely uses instances of A; 4) instances of B have information needed to instantiate A.
* Indirection: intermediate object to manage relations between 2 other objects; like controller in MVC. Supports low coupling.
* Information Expert: place responsibility for an action on the class that has the most info to do the action
* High Cohesion: class is highly focused on what it should be doing, without taking too many different random responsibilities.
* Low Coupling: the less two classes depend on each other, the better.
* Polymorphism: a subclass can redefine behaviour (with virtual methods) of a parent class, or define behaviour of an interface. Instead of switching on a type, use re-defined behaviour.
* Protected Variations: if you have a class that can change (variate), wrap it in an interface and implement it. You get bonus polymorphism!
* Pure Fabrication: a class that does not represent a concept on the domain model. Usually made to satisfy the other patterns.

1. UML class diagram



1. Sequence diagrams
2. Law of Demeter:
   1. Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
   2. Each unit should only talk to its friends; don't talk to strangers.
   3. Only talk to your immediate friends.

1. UI/ different types of evaluation

- Create effective mental models -Make appropriate functionality visible \*

-Natural mappings -Use affordances -Use constraints -Provide feedback \*

-Design with errors in mind \* - Natural mapping, mental models

**lConvey possible / appropriate actions** -physical (floppy disk, keys)

-semantic (menu graying) -cultural (red/green) -logical (spatial)

**lLearnability (novice user support)**

-Predictability (Operation Visibility, Value of previous use of system)

-Synthesizability (Immediate/Eventual Honesty, Making the mental model)

-Familiarity (Affordance/Guessability, Value of use of other systems)

-Generalizability (Apply knowledge of prior use to enable future use)

-Consistency

**lFlexibility (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)

**lRobustness (Achievement of goals)**

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

-Recoverability (Ability to take corrective action) -Responsiveness (communication rate)

-Task Conformance

**Accessibility**

Heuristic evaluation: test if the application complies to a set of predefined rules (can be done on UI prototypes). Assign severity code to the issues.

Cognitive walkthrough: imagine the app is used by a novice user; complete tasks by performing action steps and then construct a believability story.

Think aloud: bring a real user (no a technical person) and collect raw data.

1. FURPS (software quality attributes): functionality, usability, reliability, performance, supportability,
2. JUnits/Testing

Error: what did the person do wrong

Defect/Bug/Fault: what is actually wrong with the program

Failure: expected outcome does not match the actual outcome

Static test: do not run the code (e.g. checkstyle)

Dynamic test: run the code (JUnit)

White box: we can see the code and we need to execute every line of code

**Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.

**Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.

Black Box testing:

•“Plausible” faults (specific values)

•Equivalence partitions (one case per) – group all inputs that the system handles the same way

•Boundary conditions (on boundary, and one off) – look for boundaries between the way that inputs are handled

•Edge Cases – inputs requiring special handling, or at the extremes

1. Exception handling

The program has to be reliable a.k.a:

- Correctness (does what it supposed to do and ONLY what it supposed to do)

- Robust (actually work)

Stages of risk:

I. Loss of comfort

II. Loss of discretionary money

III. Loss of essential money

IV. Loss of life

Stages of handling the error:

1. Detection

2. Containment (holding the error in place)

3. Masking

4. Correction

Different strategies of error handling:

* Do nothing
* Balk (admitting defeat)
* Guarded suspension (waiting until the problem disappears)
* Provisional action (pretend nothing happened)
* Recovery (take an alternative pathway)
* Roll back
* Retry
* Appeal to higher authority (ask user to decide)
* Resign (minimize damage and sagely exit)

Rules for using exceptions:

1. Never swallow an exception

2. Do not catch a general exception

3. Handle as close as possible

4. Only catch what you can handle

5. Don’t use try/catch for a control flow

1. Coding contracts

Operation: enterItem(itemID: ItemID, quantity: integer)

Cross References: Use Cases: Process Sale

Preconditions: There is a sale underway.

Postconditions:

• A SalesLineItem instance sli was created (instance creation).

• sli was associated with the current Sale (association formed).sli.quantity became quantity (attribute modification).

• sli was associated with a ProductDescription, based on itemID match (association

formed).

The categorizations such as “(instance creation)” are a learning aid, not properly part of the contract.