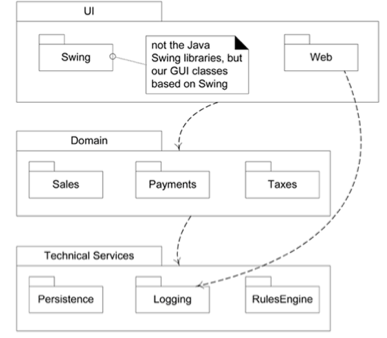


**User Accounts:** Stored in Session Variables, which are validated and set in session.php; Accounts created via register.php, logged in by login.php, and logged out by logout.php**; Planner**:User interacts with page, which call JS Functions that act in 2 parts: First part sends whatever update the user just made to the server's db\_action.php page,with takes the specified info and calls prepared statements in the SQL database to make changes.Now PHP Rebuilds the HTML for the panels that have been updated and passes it back to the JS; Second part receives the contents of the different parts of the page after the changes havebeen applied, and "plops" them in where they belong.

*WBT*: Code not covered by tests is likely to contain bugs; Divide program into elements, define the coverage of suite to be # elements executed by the suite/# elements total; **Test coverage**: goodness is determined by the coverage of the program by the test set so far; Stop testing if 100% have been tested; Bigger % is better; A program passes path-complete test data doesn’t mean its correct; Are all Edges covered/ nodes/ statements?

**JUnit:** Test generator; illegal argument exception**;** To create a test class, then test methods;Use assert methods: parameters (message, expected value, actual value)**;**Annotation: @test: does not affect running prog, just way it is treated by tools and libraries.Ex: Declare: @Test (expected=illegalArumentException.class) public void test(){assertEquals(“24”, Fac.compute(-5));}; to ignore: @ignore; *Test driven development*: Write the tests; do the design implementation; Part of agile approaches like XP; supported by tools like JUnit.



**Project Management: SMART** Tasks

**S**: specific: target a specific area for improvement; **M**: measurable: quantify or at least suggest; **A**: assignable: specify who will do it; **R**: realistic: state what results can realistically be achieved, given available resources; **T**: time-related: specify when the results can be achieved

**Web programming**: Server side programming: Calls to database , Backend, Process user input, Generate response/data, Functionality; PHP, Java, C/C++; Client side programming: UI/design/interaction; Front end; Web page; JavaScript, HTML, CSS, C/C++

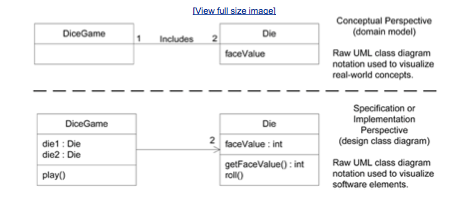
**LAPD**: Layered/Logical architecture and Package Diagram; Logical architecture: organization of software classes into packages, subsystems, layers; Layer: group of package, classes, etc. with cohesive responsibility; *Three layers*: user interface, domain, tech. services

**Operation Contract**: name, use case, precondition, post-condition: OOA artifact; Used to describe detailed changes in the domain model; DM is the most common OOA model, but these contracts are also useful

**Data Model**: Logical: describe in as much detail as possible with no regard to how physically implemented in database;

All entities, relationships, attributes, PK, FK

Physical: how actually built in the database, column names, data types, constraints, keys, etc.



**Agile software development**: usually apply time-boxed iterative and evolutionary development, adaptive planning, rapid/flexible responses;

Iterative, Incremental, Focus on facts/stories/sprints throughout the development process. Includes:

**Extreme Programming (XP):** test driven development, refactoring, continuous integration processes; pair programming;

**Unified Process (UP):** iterative; four phases:

Inception: Use case, Vision/business case, Supplementary specs, glossary

Elaboration:Start coding, major functionality; Design model, Data model, Domain model, SAD

Can still change design

Construction, Just coding, Alpha testing/release; beta testing

Transition: Beta release, Final release, User/development manuals

**Scrum**: war room, can usedaily meetings, team, sprint planning; ask three questions: what did you do, what obstacles did you face, what will you work on for next meeting??

Daily Scrum meeting: focus on sprints; answer three questions; produce a viable, usable product

Waterfall method: sequential life cycle; define all requirements before programming; need for feedback/adaptation

**Design class diagrams**: logical design; software design, software classes, packages; *Domain Model*: analysis artifact; emphasize domain concepts; static; DCD is a design artifact; emphasize the software perspective; **Steps**: Identify software classes, Add properties: include the attributes and associations from the domain model; Add methods: from interaction diagram; model class and interaction diagrams in parallel; Add dependencies

**Sequence Diagrams** (SD not SSD):A SSD visualizes a use case, while a SD visualizes a method of a class

**Software Metrics:** Quantitative ways to measure complexity**;** *Throughout* development process:Before: cost estimation**,** Middle: gauge progress**,** After: maintenance estimation

Error: human activity resulting in software containing a fault; Fault: manifestation of an error; A fault may result in a failure; failure is the runtime, observable manifestation of a fault; *Types*: **Lines of Source** code: Count non blank, non commented lines, Easy to calculate,Caveats: differences in style, language; only valid after development; **Control Flow Graph**: Each and every line gets a node; **Cyclomatic complexity**: Count number of branches (based on the control flow graph), CV=e –n + p +1; e-n+2 (when p = 1);E = edges, n=nodes, p = # connected components; for a single program/procedure; CV = #predicate nodes +1*; To draw CFG*: Split complex conditions into one node per predicate, While/for loops, If statement;*Predicate nodes*: any node that has more than one branch coming out of it; *Black Box Testing*:Generating test cases based on specification alone; Test cases are not biased toward and implementation; *White box testing*: looking into the internals of the program to figure out a set of sufficient test cases;

**What is software engineering?** Application of engineering principles to software design, implementation, maintenance and ecosystem

**Vision**: executive summary to communicate big ideas

**Use cases**: An interaction that an Actor (a user of another system) has with the system that is being designed in order to achieve a goal; Brief; two types: casual: informal, multiple paragraphs, or fully dressed

Describe functional requirements

**Domain rules**: how it operates

**Domain models**: visual representation; ID norms in use case to ID objects in model (Like ER); ID objects or conceptual classes in business domain; Draw them as classes in UML class diagram

Add associations and attributes: \_\_\_\_\_\*B: zero or more B, \_\_\_\_\_1..\*B: one or more B

**SAD**: summarize key architecture issues;

**4 + 1 architectural views**: a view from a certain perspective, focuses on structure, modularity, essential components, and main control flows

**Logical view**: conceptual organization of the software in terms of the most important layers, subsystems, packages, classes, etc.; summary of functionality, structure; show outstanding Use-case realization scenarios, a view of UP **Design Model**; LAPD; DCD

**Component view**: overview of data flows, schema mapping of objects to data; use Data Model; aka data view; DFD

**Process view**: processes and threads; responsibilities, collaborations and allocation of logical elements; SD;

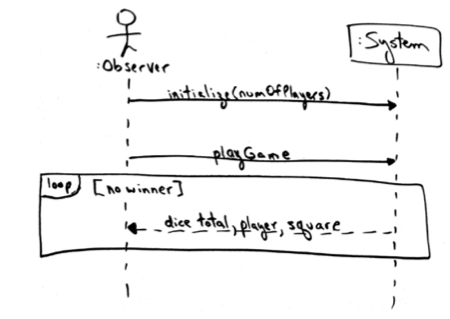
**Deployment view**: physical deployment of processes and components to processing nodes; DVD;

**Use-case view**: summary of the most architecturally significant use cases and their non-functional requirement, UP use case models;

**SSD: System Sequence Diagram**

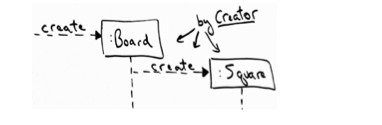
Illustrate input and output events related to the system; Shows for one particular scenario of a use case; Can apply singleton patter: implication of the pattern is that only one instance of the class (mark a 1 in the corner); Messages between objects are represented through a filled arrow and a solid line

Starting message: found message; Return values: solid line with an arrow; Object creation indicated through a dashed line; Frame lines to indicate a conditional or loop



**PathFinder**: Javascript w/ JQuery formats user action to call to db\_action.php; db\_action.php queries the database and can return the updated page elements; SQL prepared statements called via db\_action do the vast majority of the work; IE Javascript and PHP basically just act as intermediaries between user actions and the SQL database, collecting info and passing it to the database's prepared statements; CSS/BootStrap just tie everything on the HTML page together to make it pretty.

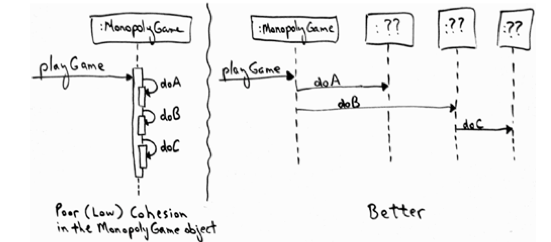
**GRASP** design patterns (General Responsibility Assignment Software Patterns): **Creator**: who creates an instance of A; i.e. class B if…B contains or aggregates A (in a collection), B records A, B closely uses A, or B has the initializing data for A;



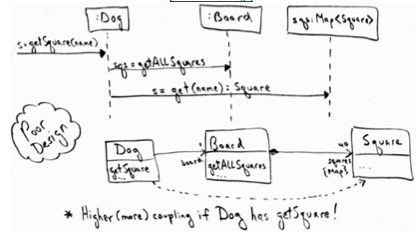
**Gang of Four:** Pure Fabrication: fabricate a new class that is purely responsible for saving objects; creates high cohesion**;** Encourage introducing software classes by this**;** Not domain classes, but introduced for reusability, maintainability, and efficiency**; Modular design**: benefit of tolerating changes (protected variation)**;** ID points of change and create a stable interface; i.e. a stack implemented by an array or a linked list

**Creational** patterns: Simple factory, Abstract Factory, Singleton

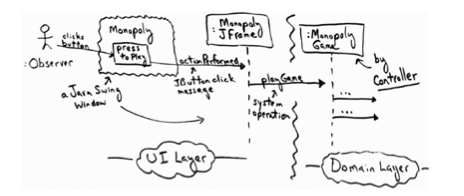
**High cohesion**: how strongly related and focused the responsibilities of an element are; Class should have a relatively low number of methods



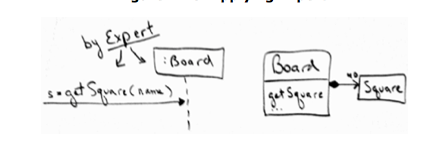
**Low coupling**: assign responsibilities so that dependencies remains low; Expert supports low coupling

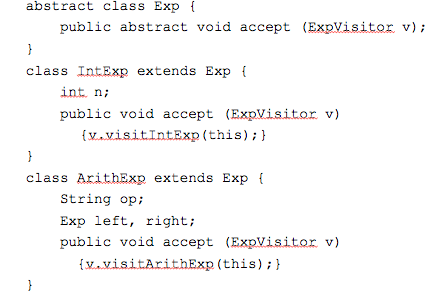


**Controller**: who should handle UI events?? An object representing the entire system (*façade*); An object representing a specific use case (session *controller*)

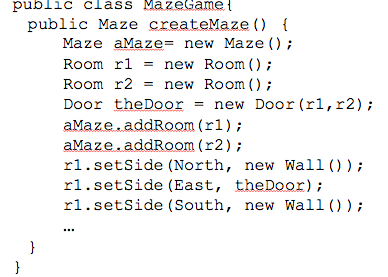


**Information Expert**: Assign responsibility to the class that has all the information to fulfill it

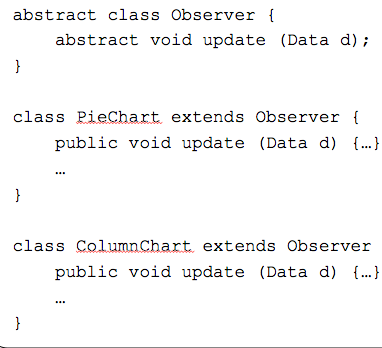




**Visitor**: operations applied to elements of a heterogeneous object structure; Lets you define a new operation without changing the classes of the elements on which it operates; Also, visitors to print in different ways



**Simple factory**: create specialized, complex objects;



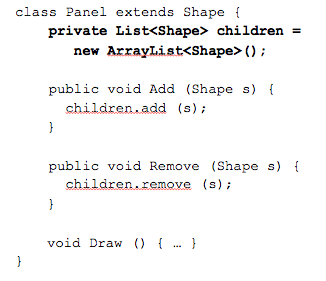
**Behavioral Patterns**: iterator, observer, visitor

**Iterator**: aggregate and access elements sequentially

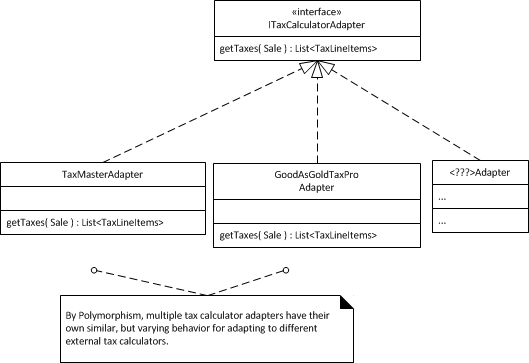
**Observer**: observers update automatically when observed object changes; Motivation in the MVC architecture; use … A change in one object requires changing other objects, and the number of objects to be changed is not known; The objects not to be tightly coupled, i.e. no assumptions should be made about the objects; Encapsulating dependent aspects of an abstraction in separate objects allows to be reused independently

**Façade**: Simplifies the interface of the subsystem; Intent: provide a unified interface to a set of objects in a subsystem; Ex. Compiler

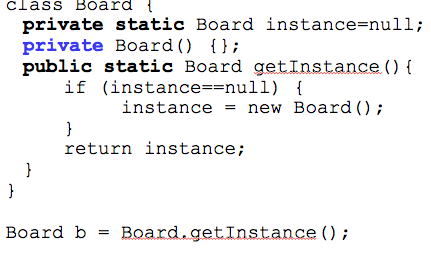
**Composite:** Represents part-whole hierarchies as tree structures**;** Define classes for composite/atomic objects so that they implement the same interface;In shape example, composite shapes hold child shapes 🡪



**Structural patterns**: how classes and objects are composed to form larger structures; adapter, façade, composite; **Adapter**: Converts interface of a class into one that clients expect; AKA wrapper; Benefits: reduces coupling to implementation specific details; Create a common interface between the POS system and any tax calculator; add an adaptor for each to implement the interface; see tax to left;



**Singleton**: guarantee access to a singular instance; one instance at a time; Product Catalog pc = newProductCatalog();Can’t call constructor if its private; static: no objects; Check if instance was already created by checking if null



**Abstract factory**: create a family of specialized factories; Intent: provide an interface for creating families of related or dependent objects without specifying concrete classes

