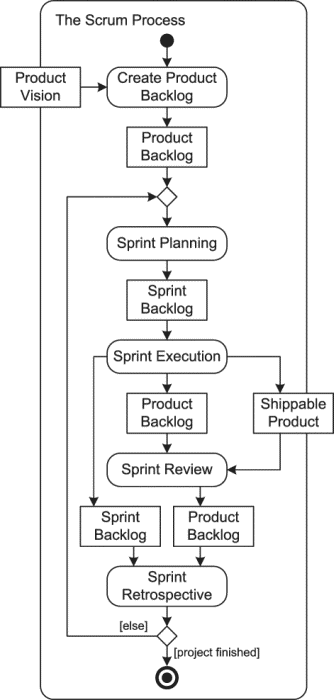
**1. Scrum Process**

**-Scrum is a process framework r**ather than a detailed process specification, which makes it more adaptable

- **The Scrum a**pproach to agile software development marks a dramatic departure from waterfall management **• Scrum** is part of the Agile movement, also an incremental process.

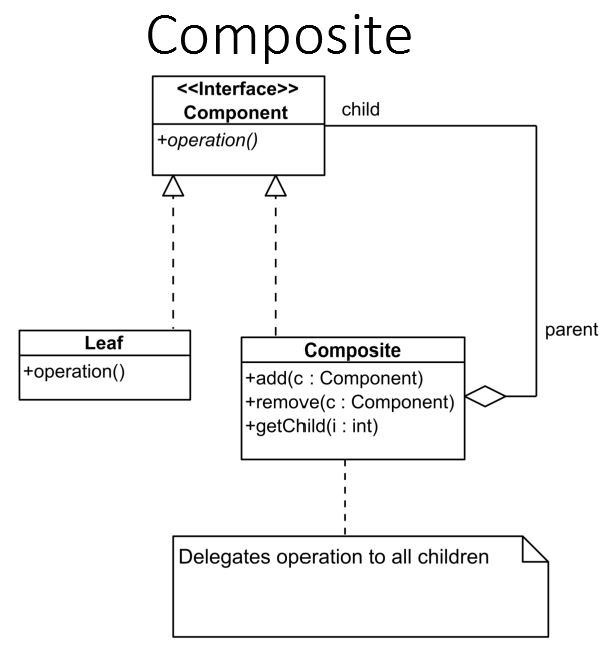
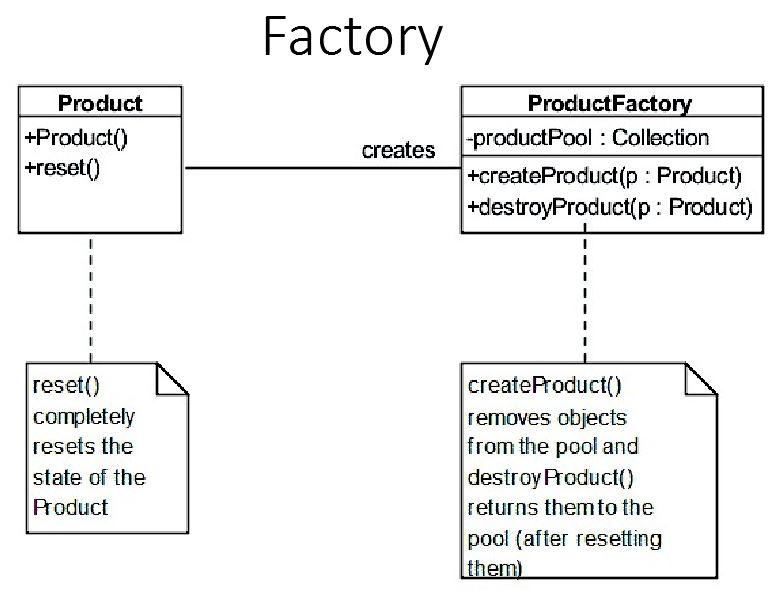


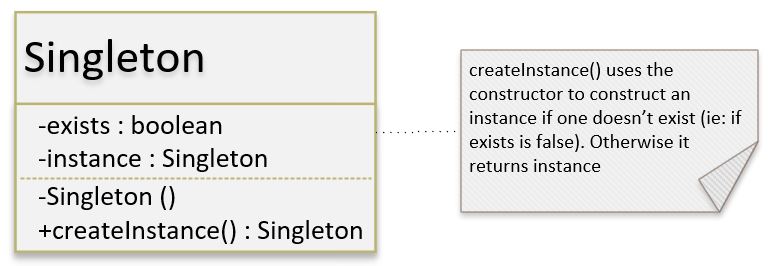
• **Scrum** and other agile methods were inspired by its shortcomings. **The most popular form of specifying product features in Scrum is the user story.** A user story is a description of a product feature or characteristic in the form As a I want to so that . \*The goal is what the user wants to accomplish and the benefit explains why. For example, the following are user stories. ***Example:*** As a book critic I want to be able to search for words and phases in a book so that I can analyze the book. User stories smaller than epics. • **Scrum** emphasizes collaboration, functioning software, team self-management, and the flexibility to adapt to emerging business realities. • **The Scrum** is an empirical Framework for learning not a methodology **• Scrum is a simple set of** roles, responsibilities, and meetings that never change. By removing unnecessary unpredictability, we’re better able to cope with the necessary unpredictability of continuous discovery and learning. **• Scrum has three roles**: Product Owner, Scrum Master, and Team. **-Scrum has three main artifacts:** product backlog,sprint backlog and shippable product. T**he product backlog**: is a prioritized list of not-yet implemented product features or characteristics. Its elements are product backlog items .The PO (product owner) is responsible for the product backlog -**A sprint backlo**g is a collection of PBIs, the tasks needed to complete them, and estimates of how much effort each task will require. The PO and the rest of the team choose items from the product backlog to be implemented in a sprint during sprint planning. The sprint backlog guides the team in its work during sprint execution. **USER STORIES ARE USUALLY EXPRESSED THE “C”’S USER STORIEs - CARD , CONVERSATION AND CONFIRMATION** -The overarching goal of every sprint is to produce a potentially **shippable produc**t, that is, a product increment that could actually be shipped to customers. **A sprint** is a product development iteration whose goal is to add something to the product of value to a customer or user. \***Sprint Planning—**Sprints begin with sprint planning, during which the PO, SM, and other team members select PBIs from the product backlog to accomplish in a sprint. \***Sprint Execution**—After sprint planning, sprint execution occurs; in other words, the tasks are performed and the PBIs implemented. \***Sprint Review**—At the end of a sprint, a sprint review occurs. All stakeholders are invited to the sprint review to see a product demonstration and to discuss the features or functions added, changes made. \***Sprint Retrospective -** After the sprint review, the team conducts a sprint retrospective during which they discuss what went well, what did not, and how the next sprint can be done better. The sprint retrospective provides a means to improve the development process. One of the main tools for **tracking progress in Scrum projects is a burn chart:**  burn chart shows how much work remains against time, then it is a burndown Scrum 9 chart; if it shows how much work has been accomplished, then it is a burn-up chart. Both kinds are used in Scrum projects, but burn-down charts are usually. **Finishing a Sprint** The two activities at the conclusion of a sprint are both aimed at reflection and improvement: the sprint review reflects on the product to improve it, and the sprint retrospective reflects on the process to improve it. At a sprint review the team presents what it accomplished in the sprint. Only completed aspects of the product should be presented. All stakeholders should be invited

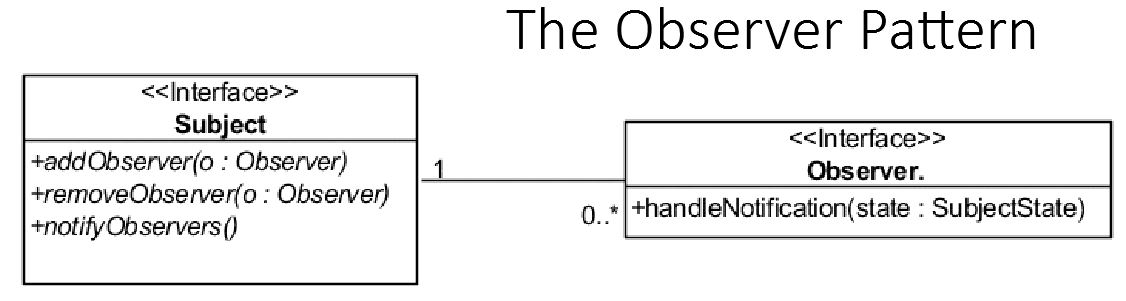
**2. Design Patterns**

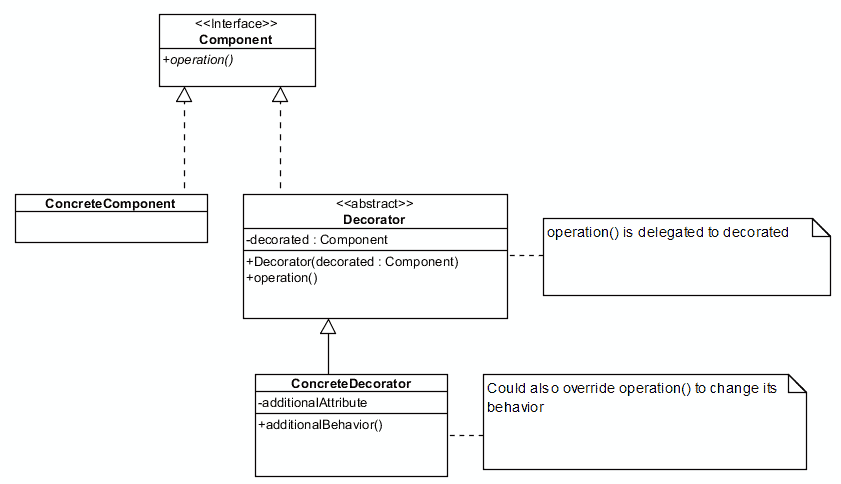
**Creational:**  ***Factory***  (Creational Design Pattern) - Creates an instance of several derived classes. ***Singleton*** (Creational Design Pattern) - Only a single instance of a class exists by hiding the constructor of the class. **Problem:** How to keep it a single instance? **Solution:** Hide constructor and define a public static operation (getinstance()). **Behavioral: *Iterator*** (Behavioral) - enables us to access the elements of an aggregate object while hiding its internal structure. Ex: loops or array lists. **Problem**: shouldn’t expose data structures while accessing/traversing. **Sol:** Clients use iterator. **Problem:** New traversal operations should be defined w/o changing its interface.  **Sol:** Define separator object to encapsulate aggregate object. ***Observer*** (Behavioral) - one- to- many dependency (if one object in a class changes its state, all of its dependents in other classes are notified. Ex: a Text Processor calls other objects like word count which is constantly updated. ***Strategy*  (**Behavioral) **-** Encapsulates algorithms inside a class to accomplish the same objective. Ex: Paragraph formatting in Word Processor. **Structural:**

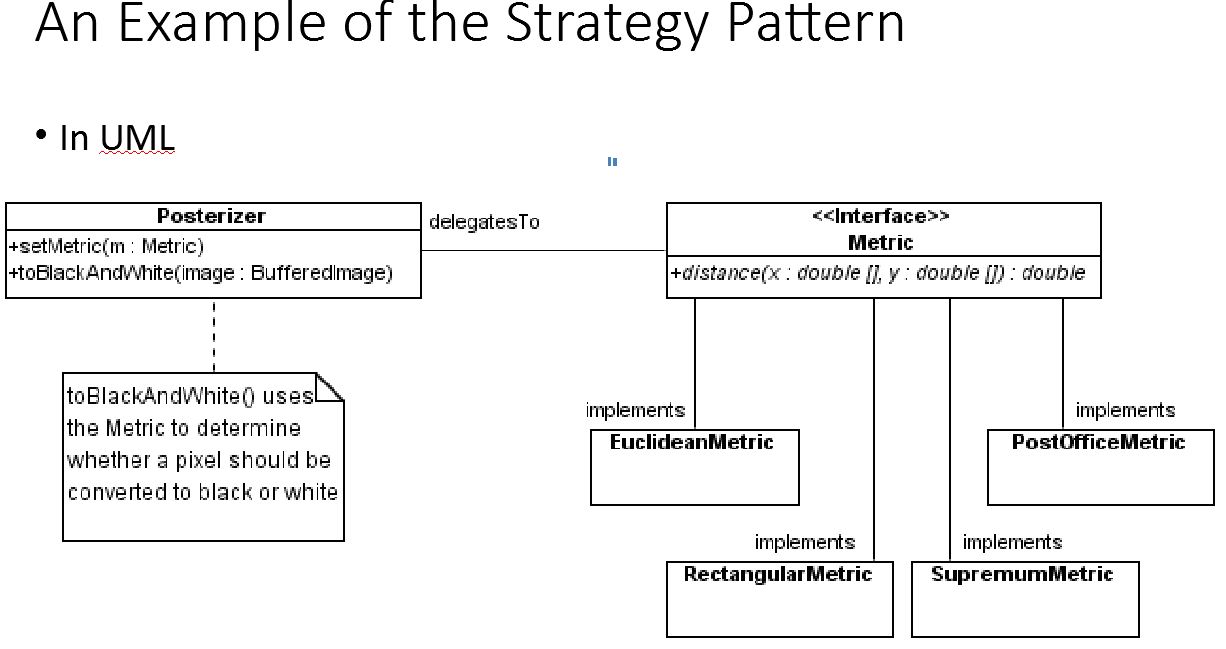
***Composite*** (Structural Design Pattern) - tree structure groups composition of objects. ***Decorator* (**Structural**)** - ex: GUI; adds responsibilities to objects dynamically.

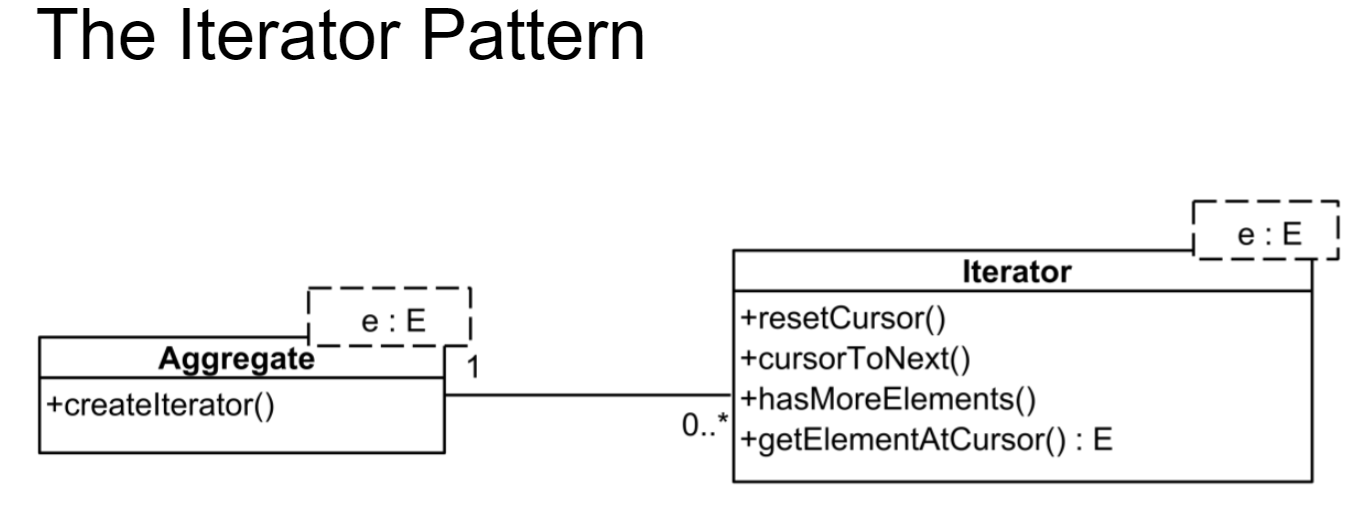










****

**3. REST Architectur**e

**Representational state transfer \*Webservices -**the communication services between the server and the computer. Not the user. ***Standard formats* -**  JSON and XML. **REST \***An architectural style protocol. \*Uses XML or JSON to send and receive data. \*Simply calls services via URL path. \*Result is readable which is just plain XML or JSON. \*Transfer is over HTTP only. \*Easy to call from Javascript. \*Performance is much better compared to SOAP - less CPU intensive, leaner code ect.

**HTTP calls (C.R.U.D) -- \***GET (create) - read 2000. \*POST (read) - write + read 201. \*PUT - (update) - write + read 200. \*DELETE (delete)- write 204. \***RESPONSE HTTP STATUS CODES: -- \***500+: Server error, 500 Internal Server Error. \*400+: Client error, 400 Bad Request, 401 Unauthorized, 404 Not Found, 409 Conflict. \*300+: Redirection, 304 Not Modified. \*200+: Success, 200 OK, 201 Created, 204 No Content. \*100+: Informational**,** 100 Continue, 101 Switching Protocols, 102 Processing.

**JSON:**

{

“Id”: 430252,

“Name”: “Kawhi Leonard”

}

**XML:**

<Athlete>

<id> 430252 </id>

<name> Kawhi Leonard </name>

</Athlete>

XML

<e>

<a>some</a>

<b>textual</b>

<a>content</a>

</e>

JSON

"e": {

"a": [ "some", "content" ],

"b": "textual"

}

**POSTMAN: GET REQUEST (get data):** Input url, add key: content-type, value: application/json into headers. Hit send to get json or xml response depending on choice. **POST REQUEST (add data):** Input url, add key: accept, value: application json into headers. Go to body, inside body go to raw, in “text” dropdown, select Json, type in or paste your json and hit send. **PUT REQUEST: (updating or inserting data into already existing):** Same as post just change request to put. **DELETE REQUEST:** go to url of item you want to delete and hit send.

**HATOEAS:**

**Level 0** -> Not RESTFul API, Plain Old XML (POX) SOAP

<Message>

<id> 32 </id>

<mesg> Howdy </mesg>

</Message>

**Level 1** -> Support Resource URI (Individual URI for each Resource)

https://localhost:5000/messages/1/comments/2

**Level 2** -> Support Status Codes and HTTP Methods (Get, Post, Post)

2xx, 4xx, 5xx

**Level 3** -> Support Responses which contains that a Client can use

{

“id” : 1,

“mesg” : “Howdy”,

“links”: [{

“link”: “insert hyperlink here”,

“ref”: “self”

}]

}

**4. Information Hiding, Cohesion and Coupling**

**Info Hiding:**

**Information Hiding**

The Principle: hide the internal details of a component from all

The Rational: prevents damage from errant external code(you can't hurt what you can't see), makes components easier to understand/use

(it enhances abstraction), simplifies modification and repair(changing

internal details should not have any impact on other components), facilitates re-use

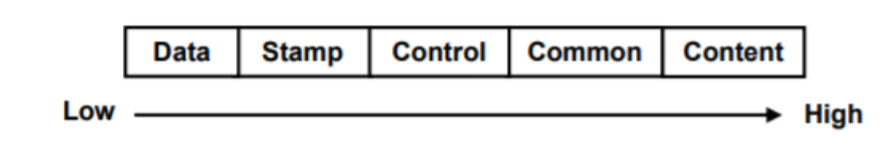
Coupling:

The Principle: minimize the coupling between modules

The Rationale: strongly coupled modules are difficult to change and debug and weakly coupled/decoupled modules are easier to understand and more likely to be re-used

**Private Info:** Local variables, data types, structures, and algorithms. **Public Info:** Interface Info(inputs and outputs), Behavior/Functionality, Errors.  **Coupling:** Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity.

The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.**: 1) Data**(parameter passing), **2) Stamp** (Composite Passing data structures)eg: objects. **3) Contro**l (method dependent on some other method) **4) Common** (dependent on global entities. **5) Content** (handling instances and Behavior is not well defined.



**Coupling Techniques:**

Info hiding - decreases coupling

Global entities - increases coupling

Complex data communicated through modules - tightly coupled

**Levels of Coupling:**

Direct: modules share variables

Common: modules share global variables

Import/Export: share variables/operations that are explicitly exported

**Cohesion:** A module having high cohesion and low coupling is said to be functionally independent of other modules. A functionally independent module has minimal interaction with other modules.

**Techniques:** separate input, validation, calculations, output

**Cohesion:**

**The principle:** The parts of a module should be closely related to each other.

**The Rationale:**

-Highly cohesive modules are easier to understand

-High cohesion tends to reduce coupling between modules

-Separation of concerns

Modules don’t cross levels of abstractions

Design modules that relate to real-world entities

**Levels of Cohesion:**

Coincidental - no meaningful connection

Logical - Collection of functionally related components

**Temporal:** Functions are executed together

Ex: Initialization

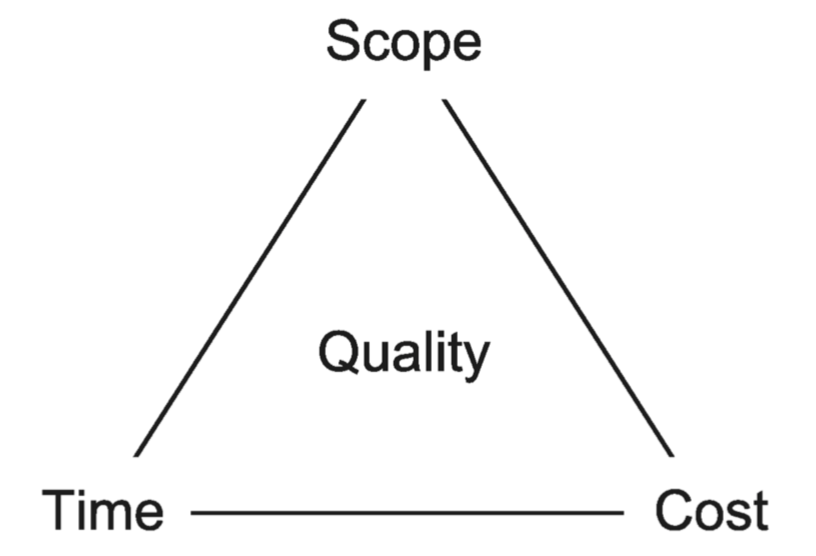
**Abstract:** Modules provide services associated with a single data type

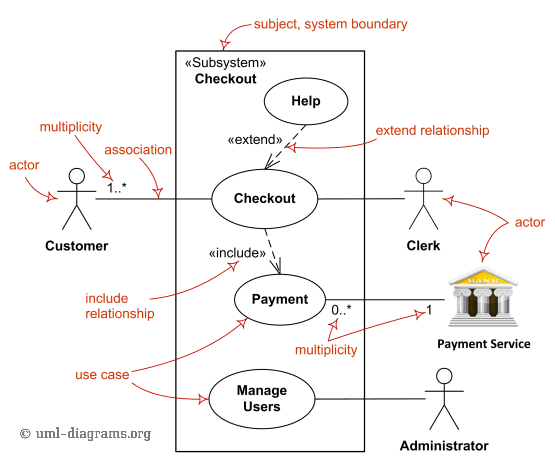
**5. UML Use Case, Class and Sequence Diagrams**

**Purpose:** \*Describe a task/procedure/process. To model a physical architecture **Elements: \***Artifacts - a physical manifestation of a component of a software system (e.g., files) \*Nodes - computational resources (i.e., a physical device or an execution environment such as a virtual machine) **Components:** \*Activity - a non-atomic (i.e., decomposable) task. \*Action - an atomic (i.e., not decomposable) task. \*Data - inputs to or outputs from actions (and may include state information in square brackets). \*Flow - flow of control and flow of data. **Representation and Interpretation** ***Directed Graph:*** \*Nodes/ Vertexes - represent actions or objects. \*Edges/Arcs/Links - represent control flows and/or data flows. ***Tokens: \****Produced and consumed by nodes \*Flow instantaneously along edges ***Execution:*** \*An action node begins executing when tokens are available on all incoming edges \*When an action node begins execution it consumes all incoming tokens \*While/after an action node executes it produces outgoing tokens **Pin: \***A terminator for data flows into (input pins) or out of (output pins) an action node **Activity Parameter: \***An object node on the boundary of an activity \*Contains the name of a particular object and the name of its type **Syntax:** \*name:type **Abstraction in Activity** **Diagrams: \***As with any model, a decision must be made about how much detail to include. **Communication Paths: \***A solid line between nodes **Deployment: \***An artifact can be placed in a node \*A node can contain a list of artifact names \*A dependency arrow (with the stereotype <<deploy>>) can connect the artifact to the node. **Methods of Providing/Delivering Support *Channel:*** Telephone, Electronic Mail, Chat/Messaging, Social Network, WWW "Forms", ***Interaction:*** Synchronous, Asynchronous.  **Support Tracking/Management *Purpose: \****Accept, prioritize, assign, track, and log requests for support ***Common Terminology: \****Call Management System \*Ticketing System **Class Attributes:** Public (+), Private (-), Protected (#), Package/Default (~), Derived (/), Static (underlined) **Relationships: \***Inheritance ( ---> ) Subclasses inherit all the attributes and methods of the superclass. \*Association ( ---- ) No dependency, just interaction (eats, calls, etc). \*Aggregation ( ----<> ) Specifies a whole and its parts, when a part CAN exist outside the whole. \*Composition ( ----◄►) When a part CAN’T exist outside the whole. ***Multiplicity:*** Numerical constraints on a relationship ( 0..1 zero to one, n specific number, n..\* number to many, m..n specific number range )

**6. Static Analysis & Refactoring Kinds of Static Analysis:**

**Management Issues: \*Scope:** The work to be done in the project. **\*Time:**  A constraint (and, hence, a resource to be monitored).  **\*Cost:** A constraint (and, hence, a resource to be monitored). **\*Project Management Iron Triangle** Brooks’ Law states that adding programmers to a late project makes it later.





Syntax Checking - ensure software obeys grammatical rules of language it is written in

Style Checking - Syntactically correct but stylistically inappropriate

Construct Checking - suspicious constructs ex: unused variables; non-portable; memory allocation inconsistencies Software Metrics: Thousand Lines of code; source lines of code; logical lines of code. Ex: E.g., for (i = 0; i < 100; i += 1) printf("hello"); /\* How many lines of code is this? \*/

**Cyclomatic Complexity:**  Calculated from the nodes and edges in control flow graph. Independent path: at least one path not yet traversed. **Halstead Complexity:**  Calculated from number of operators and operands. \*To reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. ***Formal Methods:*** •Model Checking (determine if the program satisfies all requirements). •Data Flow Analysis (calculate the possible set of values using the control flow graph). •Symbolic evaluation (tracing the execution of a program using symbolic values). **Refactoring:** Changing code without changing its external behavior. ***Ex:*** renaming variables

Improves structure, presentation, or performance of the code - can only do so on existing code/tests.

**7. GitHub**

GIT: A version control system. \*Distributed. \*Every peer has a repository. \*Repositories must be synchronized (by exchanging "change sets" or "patches"). \*Built around commit objects that can be manipulated in a variety of ways. **Repositories:** \*is an archive. \*The main development path is usually called “master”. **Working Trees**: \*A directory/folder within a repository. **Index:** An intermediate collection of changes (that need to be confirmed) before they can be moved from a working tree to a repository. **Git Functions:** \*Git checkout - to checkout a working tree. \*Git add - to add to the index. \*Git commit - to commit to the local repository from index. \*Git push - to update the central repository. \*Git pull - get changes that others have made.

8. Project Management (traditional) **Week11\_Project Management.pptx (only slides on Time, Scope and Cost triangle)**

**9. Flying Donut.** Your simple Flying Donut team project practice principles. Online Scrum software.

**Setting up project:** Click on projects in dropdown and select create. Specify the title, start/end dates, project visibility (private by default). **Collaborate:** Through invite of members only (sent through email). Roles: **Follower:** -view project. **Project member:** -create sprints, tasks and items. -move items b/t sprints or backlog. -assign items/tasks to other members. -invite members. -can’t delete project. **Administrator:** -member privileges + full rights to project. -only one that can delete projects. **Create a backlog** with abstract user stories for what we want to accomplish. Create user stories in the format of: As a <role>, I want to <activity> so that <benefit>. Stories start out as Epics, Features, then sprintables after repeated grooming. \*User stories entail the Card, Conversation, then Confirmation to make sure that the proper concern for the need is met. \*Assign importance and effort values to user stories when they get created to figure out what needs to be worked on in what order. **Organize Sprint Planning** to groom the PB with the team to determine what PBIs will be worked on for the sprint. \*Select the appropriate PBIs and add them to the sprint backlog. \*Develop doable tasks and assign them to team developers so that the team members know what tasks and assignments to focus on (in the ***‘To Do’*** column). \*When a team developer begins working on a task, move it into the ***‘Doing’*** column. \*When a task gets completed, move it over into the ***‘Done’*** column. \*When a team developer finishes a task, they can begin working on another task. \*The **Scrum Master** makes sure that the team developers are working on tasks & adhering to SCRUM rules, and if any problems arise, the **SM** helps the team member overcome the obstacle through various ways such as elaboration, explanation, and communicating with other team members to figure out who can assist with the obstacle to ensure that the workflow is happening smoothly and that developers do not get hung up on problems. \*The team gathers for meetings periodically every week to discuss what they have been working on, what they plan to work on, and what they have been struggling on. \*During meetings, further development of the user stories takes place, and the breaking those user stories down into feasible tasks to prepare for the next sprint. \*During meetings, more elaboration and explanation takes place to iron out any confusions about the project to make sure that everyone is on the same page, everyone inputs and contributes to what kind of behaviour and details need to be met in the development process, and to ensure that everyone has an assignment and is able to contribute to the project goal. This is done iteratively until the sprint comes to a finish. \*After a sprint concludes, the team discusses what was accomplished, what can be improved on, how the next sprint can be adjusted to produce better results for the future, and what else is there left to do to complete the project.

**10. Architecture Styles**

**Architecture:** Abstract model of a system. **Architecture Style:** Some architectures have similar entities, attributes

and relationships which allow them to be grouped into a a

style or idiom (i.e., when we abstract even further we see commonalities). **Layered (like an onion or wedding cake):** Entities are grouped based on services provided: [Presentation] - (Front end). [Application Logic]. [Storage] - (Back end). ***Pros***: \*Improved cohesion, \*decreased coupling, \*specialized expertised. ***Cons:*** Layer to layer communication can be inefficient. \*Can lead to cheating. ***Example:*** HVAC Unit - Ui -> Activity Scheduling -> Device interface. **Client-Server:** Entities have either one of these roles: \*Service provider: wait for and then response to requests. \*Service consumers: initiate requests. ***Pros:***  Same as layered + centralized control. ***Cons:***  Single point of failure, difficult to scale**. *Example:*** Website - Browser is the client, website is the server**. Pipe and Filter: \***Treats input/output as streams (flows in one path infinitely). \*Groups entities as either: \*Filters: processes streams. \*Pipes: connect sources and sinks. ***Ex:*** Input -> Action -> More Action -> Output. \*Can branch into two actions. ***Pros:***  \*Simple, \*easy to extend and/or modify. ***Cons:***  \*Forces lcd data transmission, \*Queue overflows, \*Unidirectional data flow, \*not good for interactive systems. ***Example:***  \*reading a file and printing a certain phrase in file, \*sorted alphabetically. ***In unix:*** cat file.txt -> grep “phrase” -> sort > out.txt

**Event Driven Architecture**: \*Entities grouped: \*Event generators/producers: post/announces events \*Event Receivers/handlers \*The entities communicate through an intermediary called an event queue \*Listening: receivers/handlers inform the queue (registration process) that they want to be notified of specific events ***-Pros:*** Promotes reuse, easy to modify ***-Cons:*** makes synchronization difficult ***Example:*** **MVC:** \*Models (the data): to realize problem-domain function \*Views: to display data to users \*Controllers: receive and carry out commands from users ***Pros:*** \*Views and controllers can be added, removed, changed without disturbing the model. \*Views can be added or changed even during execution. \*Components of the ui can be changed even at runtime. ***Cons:*** \*Views and controllers are often hard to separate \*Frequent updates may slow data display and degrade ui \*Ui components highly depend on model components ***Example:*** Calculator \*Model (observing the input) \*Controller (changing and updating) \*View (Java swing)

**11. Interaction Design *\*\*(NAK)\*\****

Is concerned with specifying the user experience for a software product. Is concerned with the visual and aural appearance of the program and with the sequence of events that occur as a user and a program exchange data. Specifies both, the appearance and behaviour of a product. The **Quality** of a user interface (How to evaluate a Design) : Effectiveness, Efficiency, Safety, Learnability, Memorability, Enjoyability, Beauty. **Related Fields:**  Ergonomics, Perceptual physiology & psychology, Cognitive psychology, Graphic Design.. In **Traditional** approaches : Done as an afterthought or a mini-phase between requirements specification and engineering design. A **Better** approach (**current** in this day) : Use interaction design as a driver of requirements specification. In SCRUM, interaction design should be conducted during a sprint for the features to be implemented in the following sprint. **Models & Notations:**  Static model: Use case diagrams --- are an easy way to represent collection of features. Several diagrams can represent different collections of features as design alternatives. **Screen** or **Page Layout** Diagrams: Drawing of part of a products visual display when it is in a particular state. Are easy to draw and manipulate, so they are good tools for considering many design alternatives. Are a great help for stakeholders in visualizing what a product will look like. Can be **Drawn** at various levels of **Abstraction:**  Good starting point -- **Wireframe.**  A high fidelity diagram shows colors, fonts, controls, and so forth can be made as the basis for implementation. **Dynamic Model: \***Use case Name: To identify the use case. \*Actors: The agents participating in the use case. Stakeholders and Needs: What this use case does to meet stakeholder needs. \*Preconditions: What must be true before this use case begins. \*Post-Conditions: What will be true when this use case ends. \*Trigger: The events that cause this use case to begin. \*Basic Flow: The steps in a typical successful instance of this use case. \*Extensions: The steps in alternative instances of this use case occurring either because of variations in the normal flow or because of errors.

**Use Case Descriptions** Can: \*Provide the notation for the next step in product design after choosing features. \*Be used to describe the interactions realized by each feature. \*Be used to explore alternative interaction flows. \*Have a big influence over many interaction design criteria, such as efficiency, safety, learnability, memorability, and enjoyability. A **Storyboard** is a collections of screen layout diagrams linked by arrows depicting events or the passage of time. \*Shows how the screen changes over time or in response to various inputs. \*Storyboards are an excellent tool for investigating design alternatives and for helping stakeholders visualize designs before their implementation. \*They can model the interactions between a system and its users and other things with which it exchanges data. \*They are valuable for interaction design, and can document some requirements. **However: \***They are not able to model requirements that don’t have to do with interactions. (Ex: algorithms, non-functional requirements). \*They are organized to trace interactions and not to specify behaviours in relation to system features. \*Hence use case models cannot capture all requirements, and they may not always be the best vehicle for expressing interaction requirements. In **SCRUM: \***A PBI may specify some aspect of a products’ interaction with its environment. \*Use case descriptions can help capture details of interaction, and so they are a good tool for elaborating interaction PBIs during sprints. \*Provide supplementary information if a PBI is elaborated to a fairly low level of detail during backlog grooming.

**Interaction Design Processes: \***A top down activity: It begins with the most abstract models and gradually refines them until a detailed specification is complete. A use case diagram can be formulated to pin down product features and capabilities. \*Use case descriptions should follow as a means to determine the best overall flow of interaction. \*Screen layout diagrams and storyboards can be made to aid in design, to obtain feedback from stakeholders, and as specifications for implementation. \***Usability Testing:** It is empirical evaluation of parts of and interaction design to determine whether it meets interaction design goals. A testing scenario can be “executed” on anything from a rough paper prototype to a full feature product. ***Measurements:*** Pre & Post test questionnaire surveys, observation of user actions, how long it takes users to achieve goals, number of user mistakes… **Interaction Design Principles:** *General principles for product appearance and behaviour* **(SAC):**  ***Simplicity:*** Simpler designs are better. ***Accessibility:*** Designs that can be used by more people are better. ***Consistency:*** Designs that present similar data in similar ways, and providing similar ways of accomplishing similar tasks, are better. *Principles that apply particularly to appearance --* **(CAP): *Contrast:*** Designs that

make things that are different appear different are better. ***Alignment:*** Designs that line up elements in a grid are better. ***Proximity:*** Designs that group related items together spatially are better. *Principles that apply especially to behaviour --* **(FeVER):**  ***Feedback:***  Designs that acknowledge user actions are better. ***Visibility:*** Designs tha Prominently display their state and available operations are better. ***Error Prevention & Recovery:*** Designs that prevent user errors and provide error recovery mechanisms are better.

**12. Requirements ( user stories and requirements done in project) *\*\*(NAK)\*\****

**Requirements** - Making user stories that reflect what the project intention and interaction is. Making user stories that we can further dissect and have a rough idea of the requirements that would need to be involved in creating the feature for that user story. **Traditional:** Project launched with a product mission statement. Requirement analysis is the activity of eliciting, examining, and understanding stakeholder needs, developing requirements specifications, and evaluating them to make sure they are clear, complete, consistent, correct, and well formed. The output is a software requirements specification (SRS). **Agile Processes:** An agile project begins with some project mission statement in the form of epics or features through user stories. \*The product owner refines these high-level features into workable user stories. \*During each spring, user stories are refined into physical level requirements and then implemented. \*At the conclusion of each sprint, stakeholders and product owner validate requirements during sprint review. \*Requirement management is a core activity of the agile methods, it is incorporated into the broader processes of backlog grooming and sprinting, and not distinguished as a separate process.

**13. Software Quality *(\*\*NAK\*\*)***

**Quality Assurance** (QA) is a systematic pattern of activities intended to ensure that a product properly satisfies the needs and desires of its stakeholders.

**Activities:** Validation - is the process of determining if a product or its specifications satisfies stakeholder needs and desires. Are we building the **Right** product? Verification - the process of determining if a product or its specification satisfies those needs and desires properly. Are we building the product **Right?**

***Functional Suitability* -** Functional completeness, correctness, and appropriateness

***Performance Efficiency* -** Processing times and throughput rates, resources used, capacity

***Compatibility* -** The degree to which a product can co-exist with and interoperate with other products

***Usability* -** Understandability, learnability, operability

***Security* -** Confidentiality, integrity, authentication, and accountability

***Maintainability -*** Modifiability, adaptability, reusability

**Defect Elimination -** Defect - any undesirable aspect of a product. Defect Elimination activities - Detection & Removal, and Prevention.

**Organizing for Quality:** Root causes for quality problems: Management tends to blame workers, and workers tend to blame managers. Implications: Quality shouldn’t be the responsibility of a single department, quality must be part of the entire developmental process.

**Quality Tradeoff:** Costs of low quality software: Negligence and liability costs, security costs, disgruntled customers (hence lost sales). Costs of high quality software: Prevention costs, Appraisal costs, Delays to market (hence lost sales)

**Use Case Example with defined parts:**

Extends when optional (actors may not use) , includes when needed (actors will always use)

**Types of Coupling:**

***Data(parameter passing)***

Void main() {

int a=27;

call(a) }

Void call(int a)

{

print(a);

}

***Stamp(object)***

Void main() {

Alpha = new Alpha();

Alpha a = 27;

call(a);}

Void call(Alpha alpha)

{

print(alpha a);

}

Class alpha{

int a;}

**Control(control method that controls coupling)**

Void main() {

int a =27;

call(a);

}

Void call(int a) {

If (get() == 1)

print(a);}

Int get() {return 1;}

**Common(global variable)**

Int a = 27;

Void main() {

call();

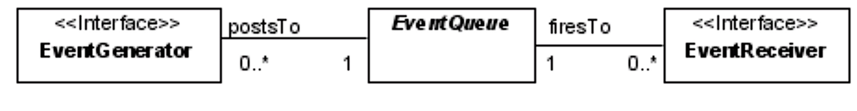
}

Void call(){print(a);}

**Stateless protocol:** A stateless protocol does not require

the server to retain information or status about each

user for the duration of multiple requests. What makes the protocol stateless is that the server is not required to track state over multiple requests, not that it cannot do so if it wants to. This simplifies the contract between client and server, and in many cases minimizes the amount of data that needs to be transferred. If servers were required to maintain the state of clients' visits the structure of issuing and responding to requests would be more complex. As it is, the simplicity of the model is one of its greatest features.

**Diagram For Event Driven Architecture:**  


**URI’s (Uniform Resource Identifier):**

Lookup/Action Based - mysite.com/courses?course=1

Resource Based - mysite.com/courses/course/1

**Restful URI’s:**

Instance Based - messages/1/comments/2

Collection Based - messages/1/comments

**Apache Maven:**

Apache Maven is a project management tool, based on Project Object Model (POM), and manages a project's build, reporting from a central piece of information.

**Design Patterns:**

Singleton - a design pattern that Create and return an instance of itself if one doesn't exist and Return the existing instance if one does exist

Code: Fileviewer

Public class Fileviewer {

Private static boolean exists = false;

Private static Fileviewer instance;

Private Fileviewer { exists = true; }}

Public static Fileviewer createInstance() {

if(!exists) instance = new Fileviewer;

Return instance;}

Public void valueChanged(ListSelectionEvent lse {

FileViewer fv;

String fn;

Fn = (String)list.getSelectedValue();

Fv = FileViewer.createInstance();

fv.load(fn);}

**Inheritance Vs. Composition:**  
Inheritance is an *"is-a"* relationship. Composition is a *"has-a"*. You do composition by having an instance of another class C as a field of your class, instead of extending C

class Engine {} // The Engine class.

class Automobile {} // Automobile class which is parent to Car class.

class Car extends Automobile { // Car is an Automobile, so Car class extends Automobile class.

private Engine engine; // Car has an Engine so, Car class has an instance of Engine class as its member.

}

**Encapsulation:**

//A Java class which is a fully encapsulated class.

//It has a private data member and getter and setter methods.

package com.javatpoint;

public class Student{

//private data member

private String name;

//getter method for name

public String getName(){

return name;

}

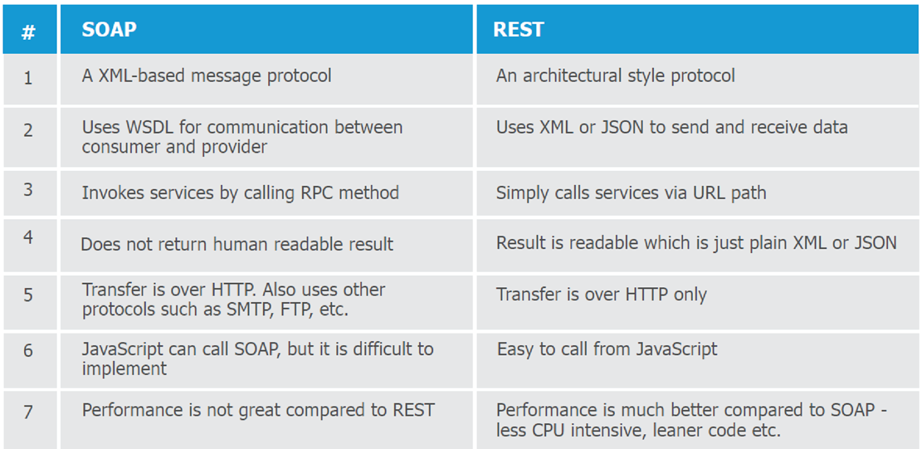
//setter method for name

public void setName(String name){

this.name=name

}

}



**Use Case:**  
In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system.

The purposes of use case diagram is

Used to gather the requirements of a system.

Used to get an outside view of a system.

Identify the external and internal factors influencing the system.

Show the interaction among the requirements are actors.