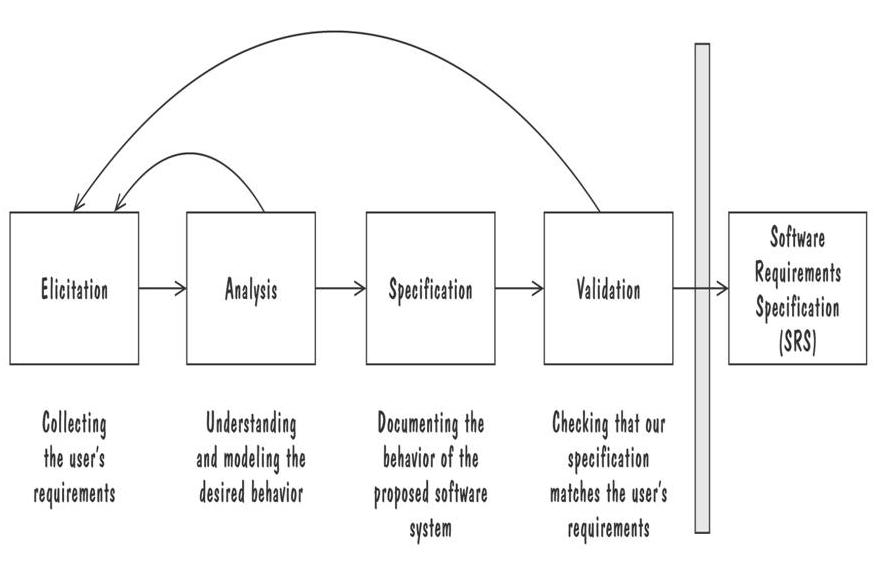
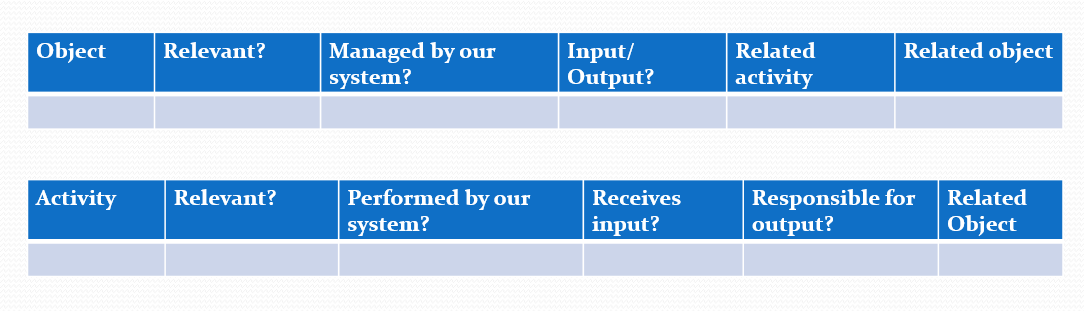
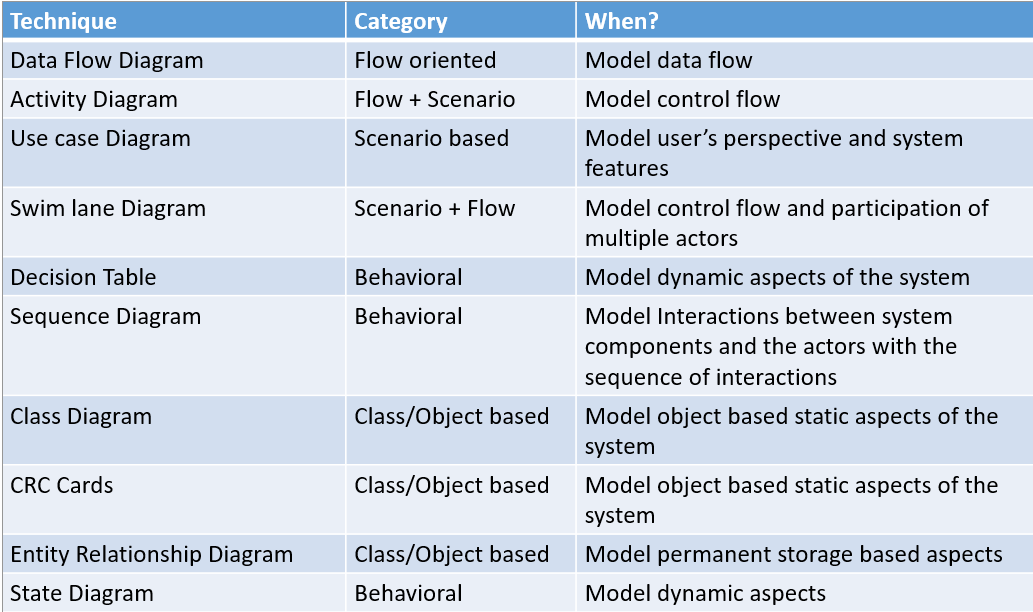
**Requirements Engineering**



* **Requirement Elicitation: Object and Activity Modeling**



* **Requirement Analysis**
* Data Flow Diagram (DFD) (context-level, level1, level2, … )
* Use case Diagram
* Object Models (ER Diagram, Abstract class diagram, CRC cards)
* Decision Tables
* State Diagram (State-charts)
* Sequence Diagrams (Message Sequence Charts)
* Activity Diagram
* Swim-lane Diagram

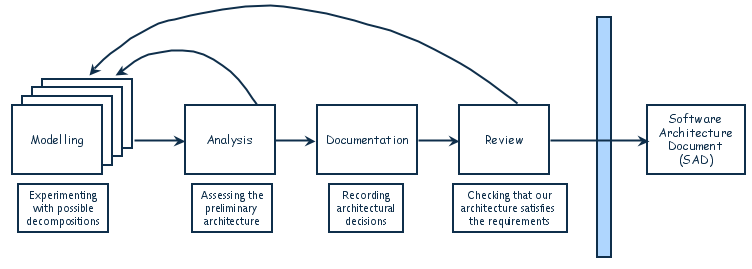


* **Requirement Specification**
  + **Standard Format:** [identifier] The [noun phrase] shall (not) [verb phrase] [constraint phrase]
  + **Functional Requirements:** describes interaction between the system and its environment, how should the system behave given certain stimuli, required behaviour in terms of required activities
  + **Non-functional Requirements:** describes some quality characteristic that the software must posses, a restriction on the system that limits our choices for constructing a solution.

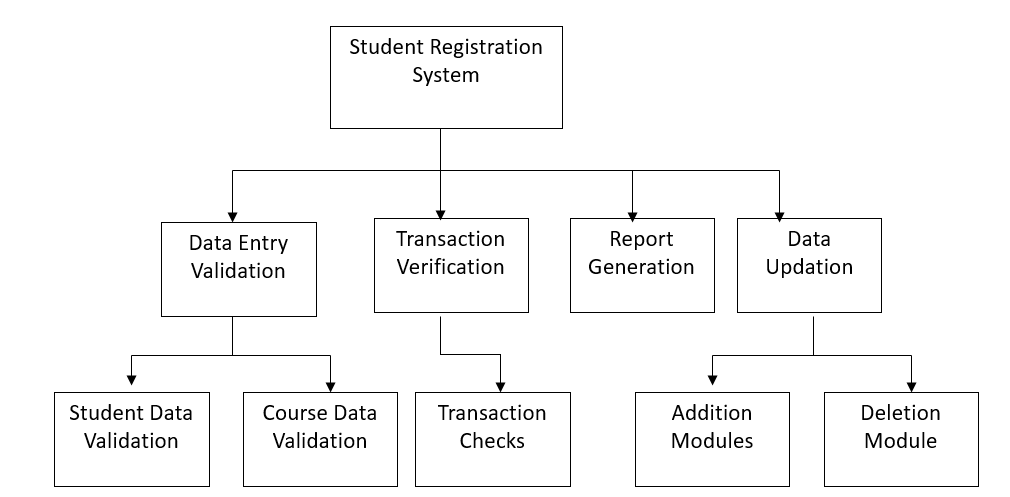
Constraints could be:

* + - **Design constraint**: a design decision such as choice of platform or interface components
    - **Process constraint**: a restriction on the techniques or resources that can be used to build the system

**Design Engineering**



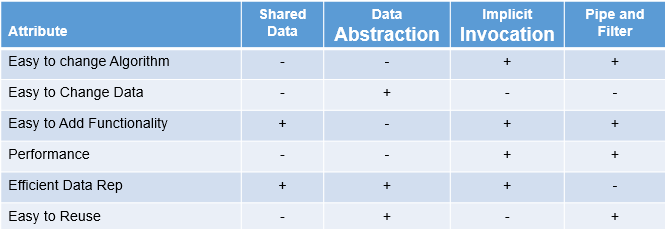
* **Decomposition**
  + Functional Decomposition
    - partitions functions or requirements into modules
    - begins with the functions that are listed in the requirements specification
    - lower-level designs divide these functions into subfunctions, which are then assigned to smaller modules
    - describes which modules (subfunctions) call each other



* + Feature-oriented decomposition
  + Data-oriented decomposition
  + Event-oriented decomposition
  + Object-oriented decomposition
* **Architectural Styles and Strategies:**

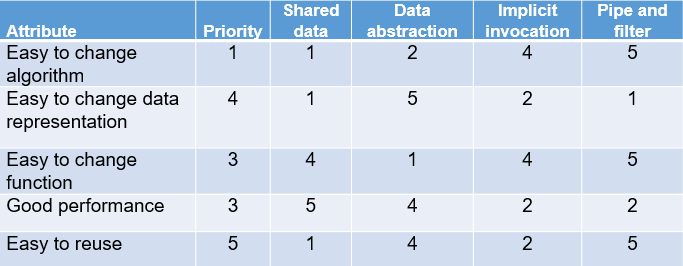
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Style** | **Properties** | **Advantages** | **Drawbacks** | **Identification Point / key** |
| Pipe-and-Filter | * The system has * Streams of data (pipe) for input and output * Transformation of the data (filter) | * The filters can be reused easily on other systems * Evolution is simple * Allow concurrent execution of filters | * Not good for handling interactive application * poor recording of filters may increase cost. | filter application  (sort, |
| Client-Server | Two types of components:   * Server components offer services * Clients access them using a request/reply protocol * Client may send the server an executable function, called a callback   The server subsequently calls under specific circumstances |  | * cannot handle two-way communication | request/reply |
| Peer-to-Peer | * Each component acts as its own process and acts as both a client and a server to other peer components. * Any component can initiate a request to any other peer component. | * Scale up well * Increased system capabilities * Highly tolerant of failures | Cannot use when:   * When file contents change frequently * When sharing speed has importance File quality is critical * When trust between peers is required | two-way communication |
| Publish-Subscribe | * Component expresses interest in an event by subscribing to it * When another component announces (publishes) that event has taken place, subscribing components are notified * Implicit invocation is a common form of publish-subscribe architecture | * Strong support for evolution and customization * Easy to reuse components in other event-driven systems | * Need shared repository for components to share persistent data * Difficult to test all subscribers | implicit invocation |
| Repositories | Two components   * A central data store * A collection of components that operate on it to store, retrieve, and update information (data accessors)   The challenge is deciding how the components will interact | * data storage and access is easy * all entities can interact with the data store. | * the data format must be acceptable to all components | a central data store (e.g. database)  read/write |
| Layering | Layers are hierarchical   * Each layer provides service to the one outside it and acts as a client to the layer inside it * Layer bridging: allowing a layer to access the services of layers below its lower neighbor | * High levels of abstraction * Relatively easy to add and modify a layer * more secure * easily modifiable (layers can be added easily) | * cannot bypass layers (except using bridging but it is not recommended) * can be difficult to implement is layers are not made correctly | hierarchical system |
| Call-Return | * Main program/sub program (like functions) * Remote procedure call |  | * limited to stand-alone architectures | multiple tasks |
| Micro-services | * A link between two communicating entities to make them compatible * Usually helps while using repositories |  |  |  |

* **Architecture Evaluation**
  + **Modifiability:** Design must be easy to change
    - Directly affected: change in responsibility (Anticipate expected changes, Value cohesion, Maintain generality)
    - Indirectly affected: change in implementation (Lower coupling, Interact through interfaces, Employ multiple interfaces)
  + **Performance:** System speed and capacity
    - depends on response time, throughput (requests per minute), load
    - Tactics for improving performance include:
      * Manage resource allocation (e.g. use structures such as First-come/first-served, Explicit priority, Earliest deadline first)
      * Reduce demand for resources
  + **Security:** Immunity and Resilience
    - Immunity: ability to thwart an attempted attack (Ensuring all security features are included in the design, Minimizing exploitable security weaknesses )
    - Resilience: ability to recover quickly and easily from an attack (Segmenting functionality to contain attack, Enabling the system to quickly restore functionality)
  + **Reliability:** performs correctly under assumed conditions
    - handle faults: A fault is the result of human error, compared to a
      * Passive fault detection: wait until fault occurs during execution
      * Active fault detection: periodically check for symptoms or try to anticipate when failures will occur
      * recovery tactics: undoing transactions, backup, correct and continue, report etc
    - handle failure: failure is an observable departure from required behaviour
    - handle exceptions using exception handling: exceptions are situations that cause the system to deviate from its desired behaviour (e.g. Failing to provide a service, Providing the wrong service, Corrupting data, Violating a system invariant (e.g.; security property), Deadlocking)
  + **Robustness:** mechanism for recovery
    - Mutual suspicion: each software unit assumes that the other units contain faults
  + **Usability:** Ease of use (for the user)
    - mostly related to UI
  + **Business goals:** quality attributes the system is expected to exhibit
    - Buy vs. Build: Time, cost, and reliability factors. (Dev time/costs, reliability, constraints from existing components)
    - Initial vs. maintenance costs: Balancing upfront savings and adaptability. (Dev costs, system modifiability, potential market loss)
    - New vs. known technologies: Weighing expertise and time trade-offs. (Expertise costs, release delays, need for in-house skills)
  + **Trade-off analysis**
    - One specification, many designs: KWIC Solutions
    - Based on attributes



+ means that the design has the attribute, - means that the attribute is not an aspect of the design

* + - Based on weights



Attribute priority ranges from 1 to 5. Column 3-6 rates design satisfaction, with 1 being lowest, 5 highest. Multiply the priorities with ratings and sum over the design to get score for each design For example score for pipe and filter = 1x5 + 4x1 + 3x5 + 3x2 + 5x5 = 55. Pick the design with the highest score. Here data abstraction has the highest score (57)

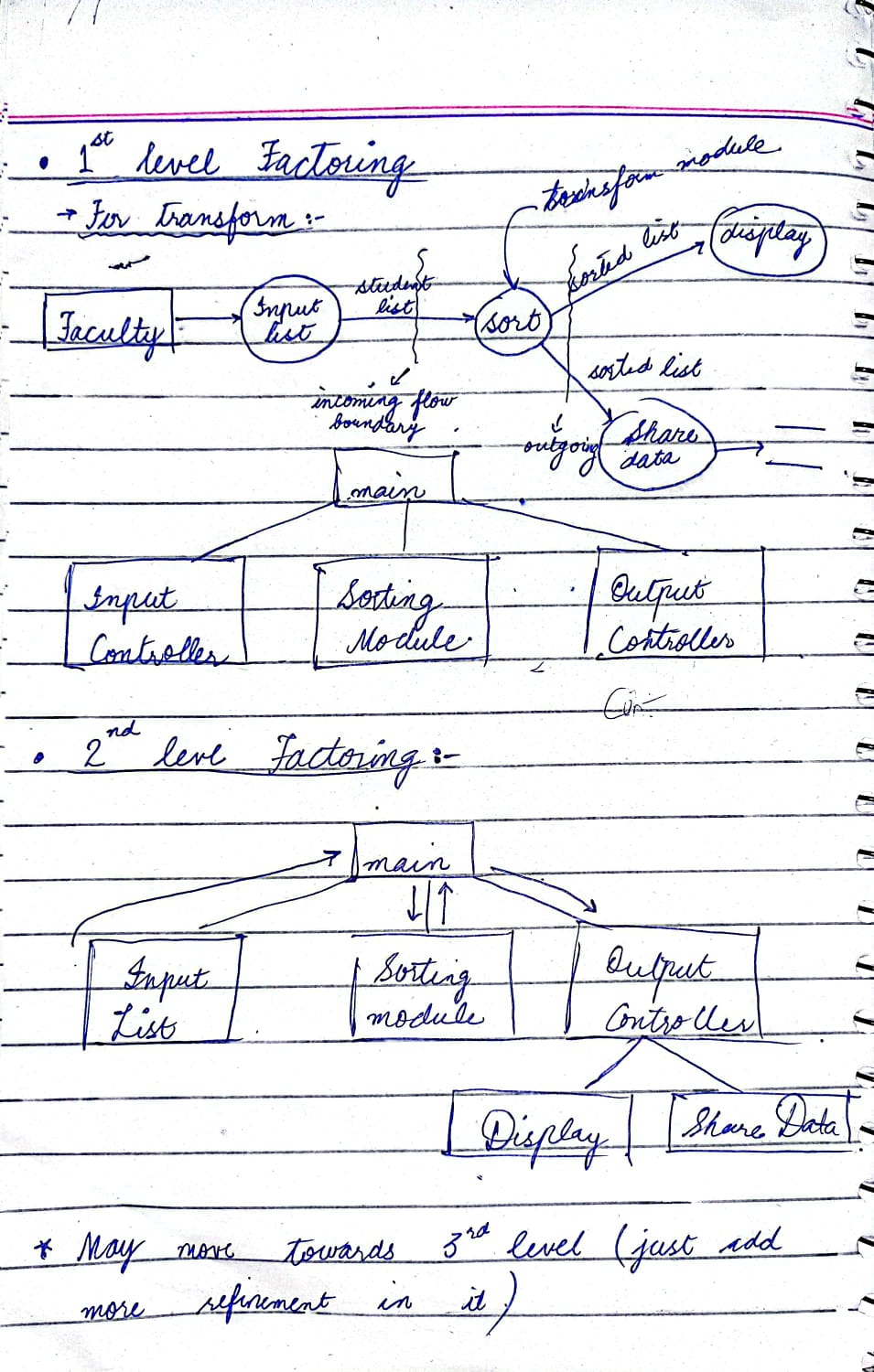
* + **Cost-benefit analysis**

ROI (return on investment

* + ROI = Benefits/Cost, ROI > 1 is desired
  + %age gain ROI = (Benefits – Cost) / Cost x 100
* **Mapping Flow to Architecture**
* **Determine flow**
  + **Transform**
    - Conversion of data from external to internal form (incoming flow)
    - Transform center
    - Conversion from internal to external form (outgoing flow)
  + **Transfer**
    - Transfer of data from external to internal form (incoming flow)
    - Transfer center
    - Transfer from internal to external form (outgoing flow)
  + **Transaction**
    - Single data item triggers data flows along one of many paths
* **Mark Input/output Flow Boundaries and map:**
  + **Transform/Transfer**

Three units

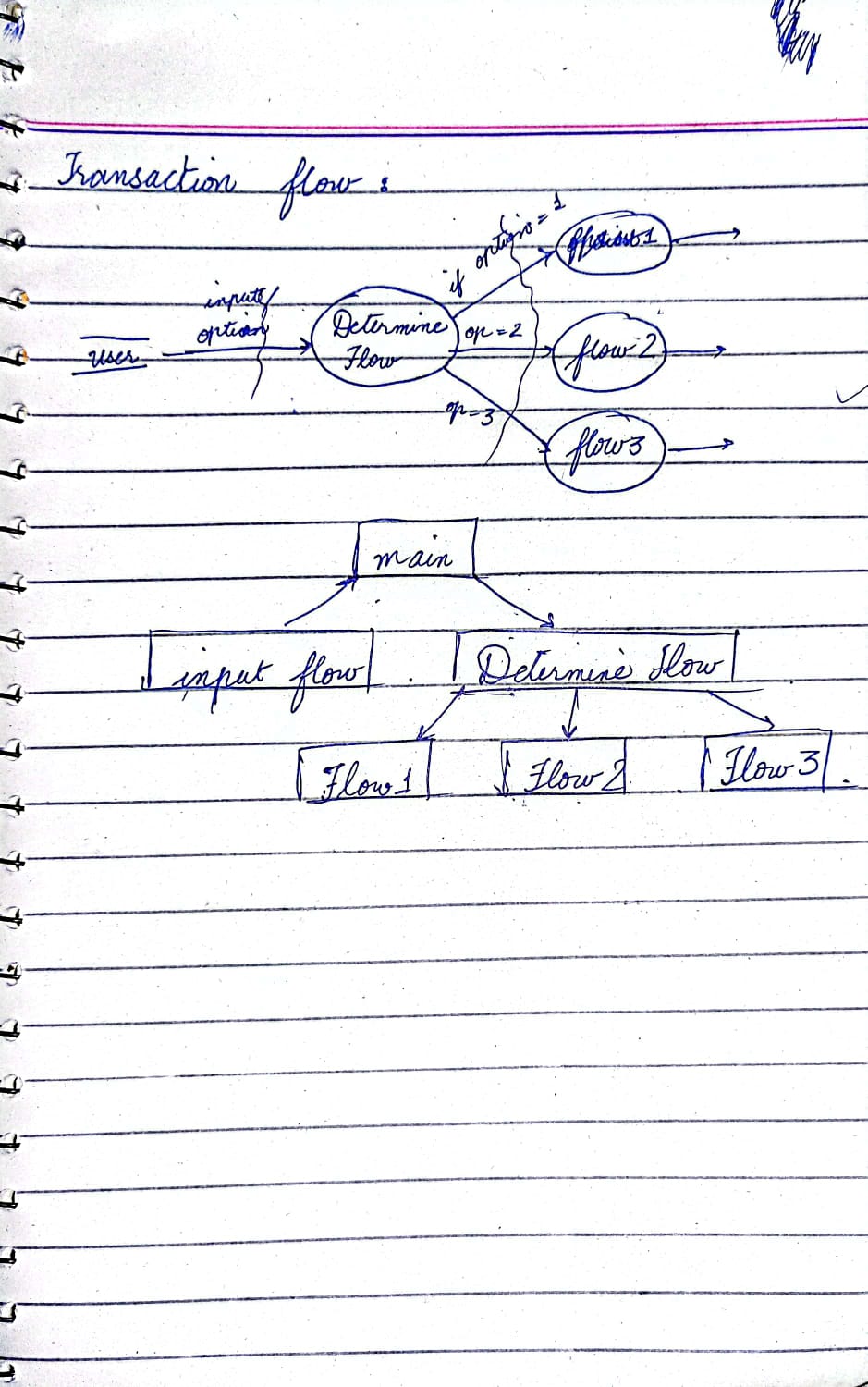
* + - Input controller
    - main transform/transfer module
    - output controller



* + **Transaction**

Two units

* + - Input controller
    - main Flow determining module

****