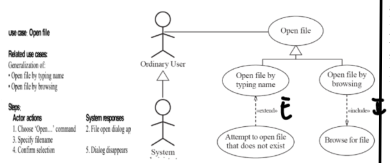
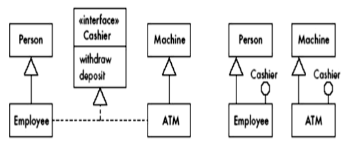
Functional: describes what the system should do: accepted inputs, produced outputs, what might be stored for other systems, computations performed. Timing-Synchronous
Non Functional: Quality: Constraints on the design to meet specific levels of quality.| Platform: what environment and technology.| Quality must be quantifiable, like response time, recovery from failure, availability.
Use Case: Typical sequence of actions that a user performs in order to complete a given task. The objective is to model the system from the point of view of how users activities in a requirement analysis. | **Should: 1.** cover sequence steps **2.** describe the user's interaction with the system (no computations) **3.** be independent from any particular UI design. Only include actions in which the actor interacts with the computer. | **How to:** Name—Actors—Goals of user—preconditions—summary – related UC's—steps—z_column-postconditions
Extensions: Used to make optional interactions explicit or to handle exceptional cases. Be very specific with quantities.
Scenario: a more specific use-case that shows amounts. Which UC to focus on. Problematic, high political/commercial value, central prototyping, system UI mock-up, doesn't have any computation. Simplest: paper prototype,

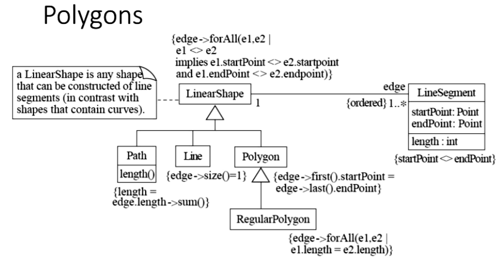


Chapter 5: modelling with classes/OCL
UML Diagrams: Classes: describe classes and their relationships | Interactions: sequence and communication diagrams, show behavior of the system in terms of how objects interact with each other. | State: show how systems or classes behave internally | Component/deployment: show how various components are arranged logically and physically.
Class diagrams: box with names inside, attributes and operations in the sub-box.
Association: Shows how two classes are related to each other
Multiplicity: shown at each end of the association | Many-one: many employees but a worker can only work for one company. | Many-many: assistant can work for many managers, managers can have one-to-many assistants | One-one: for each company, there is one board of directors, Each board can only work for one company.
Association classes: used when an attribute that connects two association classes but cannot be placed in either of them. [Student]1-*[Registration]*-1[courseSection]
Reflexive association: class is associated with itself. | symmetric: basically the same semantics, like cousin, friends, ect. | asymmetric: semantically different, like parent-child.
Directionality: by default bi-directional. Possible to limit the direction by adding an array to the end. The arrow incoming means that class doesn't know about what classes the association is coming from.
Generalization: superclass in java. labelled group of generalizations of classes. Avoid unecessary generalization. You can have a hierarchy of generalizations. Part time vs full -time students as separate classes
Object diagrams: Diagram showing all the instances using objects and links underlines the title. | Generalization do not appear in instance diagrams
Aggregation: is apart of, use aggregation when parts are part of the aggregation [vehicle]<>1-*[vehiclePart], [Country]<>1-*[Region]
Composition: Strong part of aggregation: when aggregation is destroyed the two parts are destroyed as well, [building]<B>-*[rooms]
Propagation: mechanisms where an aggregate is implemented by having the aggregate perform that operation on its parts. Propagation is to aggregation as inheritance is to generalization
Interface: describes portion of visible behavior like superclass with no attributes and implemented methods.

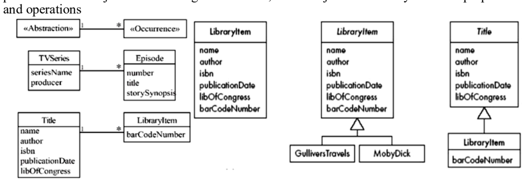


OCL: Object constraints language: OCL expression simply specifies a logical fact about the system that must remain true, cannot have side effects or compute data, in class diagrams and specify the values of attributes and associations.
Can be build from: references to role names, association names, attributes and the result of operations. Have logical values true or false. Operators like and, or, =, <, > ,<>, !=, string values like 'a string', integers, real numbers, arithmetic operations **Places to use:** to specify invariants n classe.s to specify type invariant for stereotypes. Describe, pre and post conditions. To describe guards, as a navigation language specify constraints on operations.
Each expression is written in the context of instance of a specific type. Self refers to a contextuatual instance. If using own attributes and opeartions , then use a dot. If using OCL operation, use -> instead
Exploratory domains: developed in domain analysis to learn about the domain.
System domain model: aspects of the domain represented by the system. Omit many classes that are needed to complete the system.
System model: Includes classes ued to build UI and usual experience components –domain model UI classes, utility classes, architectural classes
For domain models, you can discover classes. For system model, you create classes for UI, ect.
Associations: should be created if a classes possesses control, is connected to, is related to, is part of, member of, or has parts/members.
Actions are not associations!
Attribute: information that must be maintained about each class and contains simple values, like string or number.

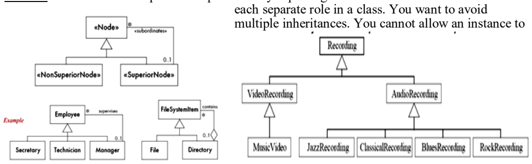
An example: constraints on Polygons



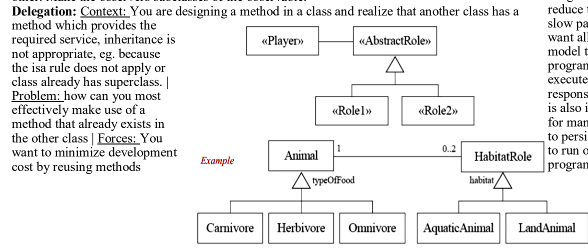
Chapter 6: Design Patterns
It is a recurring aspect of design that outline reusable solutions to general problems. They are as general as possible. Systematically documented.
Abstration-Occurrence: Context: often in a domain model you find a set of related objects (occurrences). The members of such a set share common information but also differ from each other in important ways. | Problem: what is the best way to represent such sets of occurrences in a class diagram | Forces: you want to represent the members of each set of occurrences without duplicating the common information (1: solution, 2: antipattern)
General hierarchy: Context: objects in a hierarchy can have one or more objects above them (superiors), and one or more objects below them(subordinates). Some objects cannot have subordinates | Problem: How do you represent a hierarchy of objects, in which some objects cannot have subordinates. | Forces: you want a flexible way of representing the hierarchy that prevents certain objects from having subordinates, all the objects have many common properties and operations



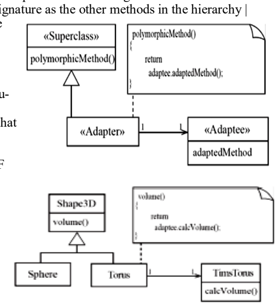
(1: solution, 2: antipattern)
Player Role: Context: a role is a particular set of properties associated with an object in a particular context. An object may play different roles in different contexts. | Problem: how do you best model players and roles so that a player can change roles or possess multiple roles? | Forces: it is desirable to improve encapsulation by capturing the information associated with each separate role in a class. You want to avoid multiple inheritances. You cannot allow an instance to



change class. | Antipattern: Merge all properties into a single player class and not have roles at all, create roles as subclasses of the player class.
Singleton: Context: it is very common to find classes for which only one instance should exist Problem: how do you ensure that it is never possible to create more than one instance of a singleton class
Forces: the use of a public constructor cannot guarantee that no more than one instance will be created. The singleton instance must also be accessible to all classes that require it.
Observer: Context: when an association is created between two classes, the code for the classes becomes inseparable. If you want to reuse one class, then you also have to reuse the other | Problem: How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
Forces: you want to maximize the flexibility of the system to the greatest extent possible | Anti-pattern: connect an observer directly to an observable so that they both have references to each other. Make the observers subclasses of the observable.
Delegation: Context: You are designing a method in a class and realize that another class has a method which provides the required service, inheritance is not appropriate, eg. because the isa rule does not apply or class already has superclass. | Problem: how can you most effectively make use of a method that already exists in the other class | Forces: You want to minimize development cost by reusing methods

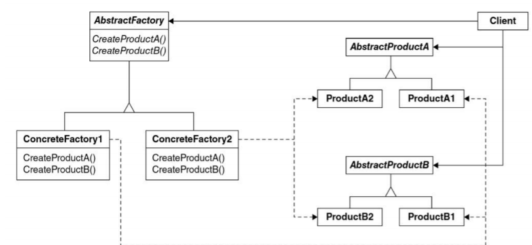


Adaptor: Context: you are building an inheritance hierarchy and want to incorporate an existing class. The reused class is also often already part of its own inheritance hierarchy | Problem: How to obtain the power of polymorphism when reusing a class whose methods have the same function but not the same signature as the other methods in the hierarchy | Forces: you do not have access to multiple inheritances or you do not want to use it .

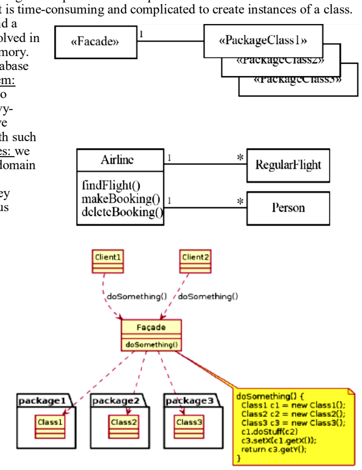


Façade:
Context: Often, an application contains several complex packages. A programmer working with such packages has to manipulate many different classes
Problem: How do you simplify the view that programmers have of a complex package | Forces: It is hard for a programmer to understand and use an entire subsystem. IF several different application classes call methods of the complex package, then any modification made to the package will necessitate a complete review of all these classes.
Immutable: Context: an immutable object is an object that has a state that never changes after creation | Problem: how do you create a class whose instances are immutable | Forces: there must be no loopholes that would allow illegal modification of an immutable object | Solution: Ensure that the constructor of the immutable class is the only place where the values of instance variables are set or modified. Instance methods must not have side effects. If a method that would otherwise modify an instance variable is required, then it has to return a new instance of the class.

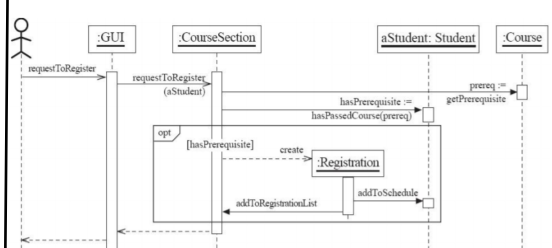
Factory: Context: A reusable framework needs to create objects; however the class of the created object depends on the application | Problem: how do you enable a programmer to add new application-specific class into a system built on such a framework | Forces: we want to have the framework create and work with application-specific classes that the framework does not yet know about. | Solution: the framework delegates the creation of application-specific classes to a specialized class, the factory. (*The factory is a generic interface defined in the framework and declares a method whose purpose is to create some subclass of a generic class.)*
Difficulties: patterns are not a solution to all our problems. You can be tempted to insert a pattern randomly when you see an indication which can lead to unwise design decisions. Justify each decision you make. Developing patterns is hard, a poor pattern can be hard to apply correctly.

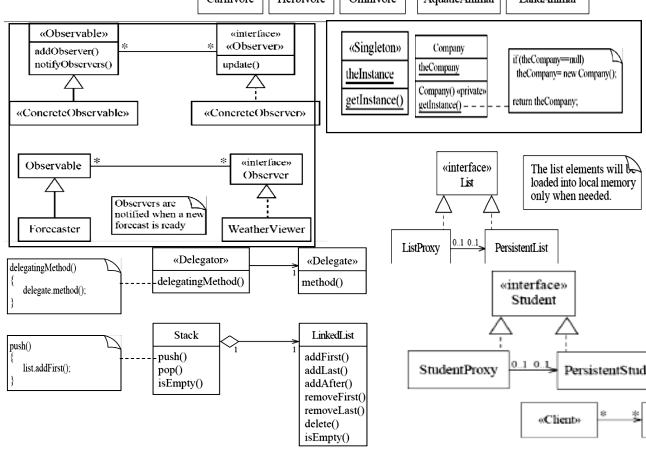


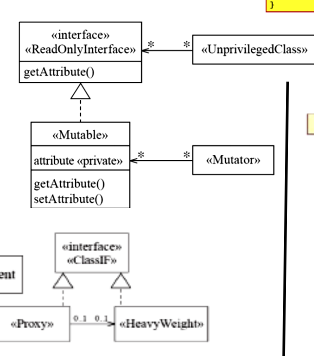
Read-Only: Context: you sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable | Problem: How do you create a situation where some classes see a class as real-only whereas others are able to make modifications | Forces: Restricting access by using the public, protected, and private keywords are not adequately selective. Making access public makes it public both for read and write.
Proxy: Context: Often, it is time-consuming and complicated to create instances of a class. There are a time delay and a complex mechanism involved in creating the object in memory. Class may reside in a database or remote server. | Problem: How to reduce the need to create instances of a heavy-weight class? How can we reduce the interaction with such slow paste classes | Forces: we want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities. It is also important for many objects to persist from run to run of the same program

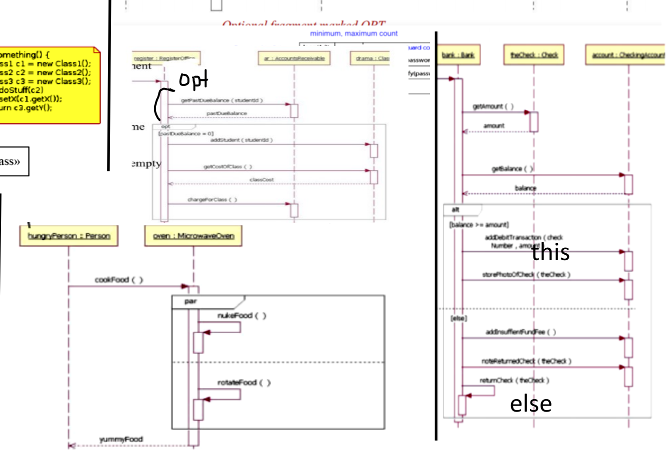


Chapter 8: Modeling interactions and behaviours
Help you visualize how the system runs, often built from a use case and a class diagram
Elements: instances of classes shown as boxes with classes and objects identified by underline. | **Actors:** use the stick person symbol | **MSGS:** shown as arrows from actor to object or object to object
Sequence diagrams: objects are arranged horizontally, actors initating the interaction is often shown on the left. The verticle dimension represents the time, vertical line called a lifeline is attached to each object or actor, a lifeline becomes a broad box, called an activation box during activation period. A message is represented as an arrow between activation boxes of the sender and receiver. A message is labelled and can have an argument list and a return value.
Fragments:
Alt: for alternatives with conditions. Multiple operations, separated by dashes
OP: optional, to specify a guard condition behavior fragment with no alternative special case of alt. Equivalent to an alt but with two operands. First is save operand of the opt. Second is empty operand with else guard
Loop: May be executed several times. At least executed minimum times and up to maximum count as long as the guard condition is true
Par: parallel: concurrency: two or more operands executed in parallel

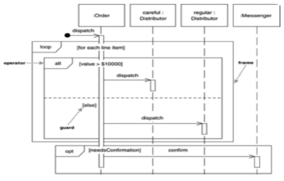


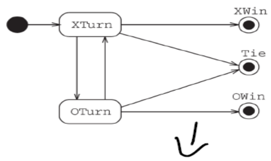




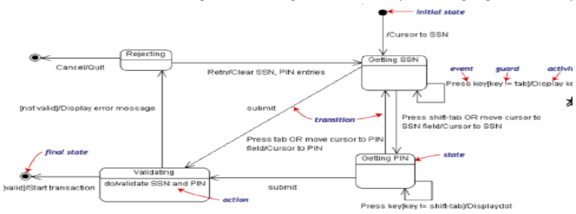


Nested fragments: you can have nested fragments just as you have nested blocks of code

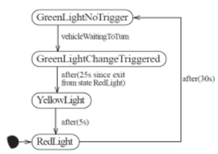


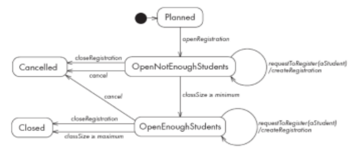


State Diagrams: Describes the behavior of a system, some part of a system, or an individual object. At any given point in time, the system or object is in a state. Being in a state means that it will behave in a certain way in response to any event that occurs. Some events will cause the system to change states. In the new state, the system will nbehave in a different way to events. A state diagram is a directed graph where the nodes are states and the arcs are transitions. At any given point in time, the system is in one state, it will remain in this state until an event occurs that causes it to change states, a state represented by a rounded rectangle containing the name of the table, special state filled in circle is initial, circled circle is end state.
Transitions: represents a change of states in response to an event. It is considered to ovvur instantaneously. The label on each transition is the event that causes change of states. They also have time-outs and conditions
Activities: Something that takes place while the system is in a state. It takes a period of time, the system may take a transition out of the state in response to the completion of the activity, some outgoing transitions may

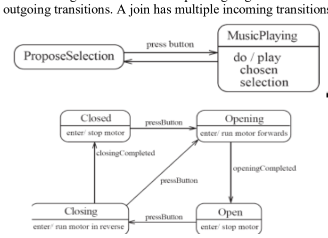


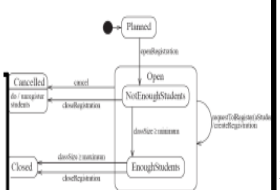
result in-the interruption of the activity-or-an early exit from the state.
Actions: an action is something that takes place effectively instantaneously: when a particular transition is take-upon entry into a particular state-or-opon exit from a particular state. It consumes no time
Nested sub-states and guard conditions: A diagram can be nested in a state –inner states are called

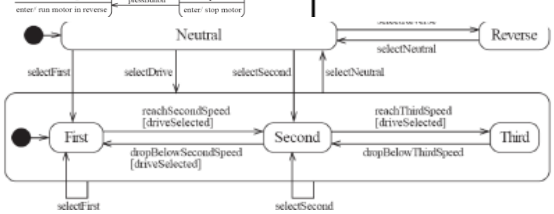




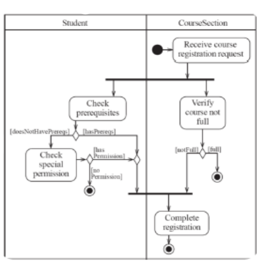
substates (nested is this diagram below)
Activity Diagram: like a state diagram. However, most transitions are caused by internal events, such as the completion of a computation. Can be used to understand the flow of work that an object/component performs. Can be used to visualize the interrelation and interaction between different use cases. Often associated with several classes-> using swim lanes: a partition of a class
These diagrams represent concurrency: This is shown using forks, joins, and a rendezvous. A fork was one incoming transition and multiple outgoing transitions. A rendezvous has multiple incoming and multiple outgoing transitions. A join has multiple incoming transitions and one outgoing transition. A folk splits into



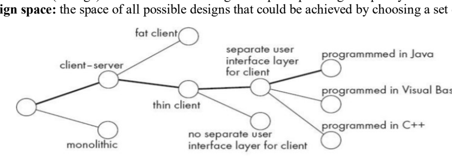




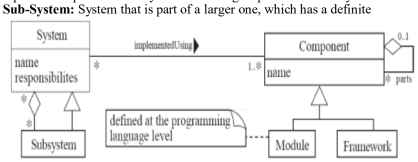
threads and the join/rendezvous must wait for each separate thread to finish before executing and continuing. You can implement complex classes by using activity diagrams first. Interaction activity, and state diagrams help you create a correct implementation. Good to use when behavior is distributed across several use cases, like how state diagrams are useful when different conditions cause instances to respond differently to the same event.
Difficulties: Dynamic modeling is a difficult skill. In a large system, there is a large number of paths a system can tke. It is hard to choose classes to which to allocate each behavior . Skilled developers lead the process work iteratively: class diagram, use cases, responsibilities, interaction diagram and state diagrams. Draw different diagrams capturing related information to highlight problems.



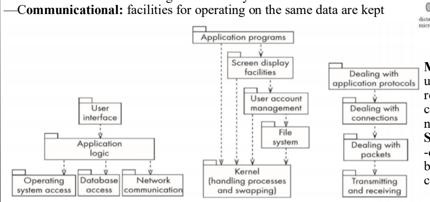
Chapter 9.
Design: problem-solving process whose objective is to find and describe a way to implement a system's functional requirements while respecting the constraints imposed by the quality, platform and process requirements (+budge) and also while adhering to the principles of good quality
Design space: the space of all possible designs that could be achieved by choosing a set of alternatives



Design issues: sub-problems in the overall design problem. Each issue normally has different options for solutions where the engineer needs to make a design decisions. For this, the eng. Needs knowledge of: the requirements, the design created so far, the technology available, software design principles and best practice, what has worked well in the past
Components: any piece of hardware or software that has a clear role. A component can be isolated allowing you to replace it with a different component that has equivalent functionality. They ar designed to be reusable. Some, though, perform special-purpose functions
Modules: A component that is defined at the programming language level, like classes or methods or packages.
System: A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software, or both. Can have a specification which is then implemented by a collection of components. A system continues to exit, even if its components are changed or replaced. Requirement analysis-determining responsibilities of system.
Sub-System: System that is part of a larger one, which has a definite



interface
Top down: first design the very high-level structure of the system. Then gradually work down to detailed decisions about low-level construction. Finally, arrive at detailed decisions such as the format of particular data items; the individual algorithms that will be used,
Bottom-up: make decisions about reusable low-level utilities, then decide how these will be put together to create high-level construction
Hybrid: is usually used to give a good structure using top-down and then a bottom-up approach to create reusable components.
Aspects of good design:
—Software architecture: The division into subsystems and components, how those will be connected, how they will interact, and their interfaces.
—Class Design: Various features of the class.
—UI Design: the process of making interfaces in software or computerized devices with a focus on looks or style
—Algorithm design: design of computational mechanism
—Protocal design: Designing of communication protocol
—Goals: of good desing, increase profit, reduce cost, accelerate development, conforming to requirements, increase: usability, efficiency, reliability, maintainability, reusability.
Design principles:
1. Divide and conquer: dealing with something big all at once is normally much harder than dealing with a series of smaller things
—A distributed system is divided up into clients and servers
—A system is divided up into subsystems, a system can be divided up into one or more packages, a package is divided up into classes, a class is divided up into methods
2. Increase cohesion: a subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things. This makes the system as a whole easier to understand and change. Types of cohesion:
—**Functional:** facilities are kept together that perform only one computation with no side effect. Everything else is kept out
—**Layer:** related services are kept together, everything else is kept out, and there is a strict hierarchy in which higher-level services can access only lower-level services. Accessing a service may result in side effects
—**Communicational:** facilities for operating on the same data are kept



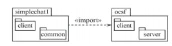
(a) Typical layers in an application program
(b) Typical layers in an operating system
(c) Simplified view of layers in a communication system

together, and everything else is kept out. Good classes exhibit this.
—Sequential: a set of procedures, which work in sequence to perform some computation, is kept together. The output from one is input to the next, everything else is kept out.
—**Procedural:** a set of procedures, which are called one after another is kept together, everything else is kept out.
—**Temporal:** procedures used in the same general phase of execution such as initialization or termination, are kept together.
—**Utility:** related utilities are kept together when there is no way to group them using a strong form of cohesion.
3.Reduce coupling: coupling occurs when there are interdependencies between one module and another interdependency making changes in one place will require changes somewhere else. Types of coupling:
—Content: a component surreptitiously modifying internal data of another component. Avoid this.
—**Common**: the use of global variables. Severely restrict this
—**Control**: one procedure directly controlling another using a flag. Reduce this with polymorphism.
—**Stamp**: one of the argument types of a method is one of your application classes. If it simplifies the system, replace each such argument with a simpler argument (an interface, a superclass or few simple data items)
—Data: the use of method arguments that are simple data. If possible, reduce the number of arguments
—routine call: A routine calling another. Reduce the total number of separate calls by encapsulating repeated sequences
—type use: The use of a globally defined data type. Use simpler types where possible(superclass or interface)
—**inclusion**/export: including a file or importing a package.
—**External:** A dependency exists to elements outside the scope of the system, such as the operating system, shared libraries or the hardware. Reduce the number of dependencies
4.Increase abstraction: ensure that your design allows you to hide or defer consideration of detail, thus reducing complexity. Abstraction allows you to understand the essence of a subsystem without having to know unnecessary details
5.Increase reusability: design the various aspects of your system so that they can be used again in other contexts.
—Generalize your design—Follow the preceding three steps—Design your system to contain hooks—Simplify your design—Reuse: actively attempt to reuse code
6. Reuse where possible: design with reuse is complementary to design for reusability Actively reusing designs or code allows you to take advantage of the previous investment in reusable code(cloning is not reuse)
7.Design for flexibility: anticipate changes and prepare, do not hard code and leave all options open.
8.Anticipate obsolescense:
—Avoid using early released technology
—Avoid libraries not specific to particular environments
—Avoid undocumented features
—Avoid companies that do not provide long term support
—Use standard languages and technology
9.Design for portability: having the software run on as many platforms as possible. Avoid the use of facilities that are specific to one particular environment. **10.Design for testability:**
—Design program to automatically test
—Ensure that all the functionality of the case can be driven by external programs bypassing a graphical user interface
—In java, you can create a main() method in each class in order to exercise the other methods
—Use Junit or similar frameworks
11.Design defensively: handle all cases where other code might attempt to use your components inappropriately. Check that all the inputs to your components 'preconditions" are valid
—Design by contract: a technique that allows you to design defensively in an efficient and systematic way
—Each method has an explicit contract with its caller

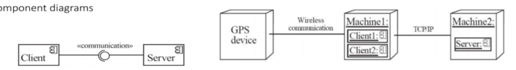
Use priorities and objectives to decide among alternatives: list possibilities, list pro/cons, determine if any alternative impedes on objectives, choose best alternatives, adjust priorities.
Cost-benefit analysis: list costs for soft.eng work, cost of development and end-user cost, and place up against benefits, like the incremental software dev time saved, increased sales, ect.
Software architecture: design global organization of the software system. You need to develop an architectural model to enable everyone to better understand the system, allow people to work on individual pieces, prepare for extension of system, and facilitate reusability.
Stable architecture: new features can be added with only small changes to the architecture
Steps: skeptch an outline of the architecture using requirements analysis, determine main components, consider each use ase and adust the architecture mature the architecture
Architecture modeling: specifically UML(all can be useful too)

Package diagrams

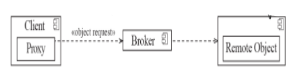
Deployment diagrams

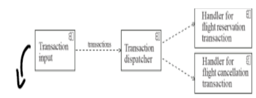


Component diagrams

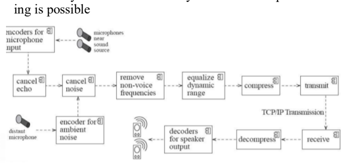


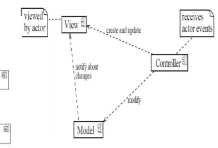
Architectural pattern/style:
Multi-layer: In a layer system, each layer only communicates wit the layer immediately below it. Each layer has a well-defined interface used by the layer immediately above. The higher layer sees the lower layer as a set of services. A complex system can be build by super posing layers at increasing levels of abstraction. Important to have a separate layer for the UI. Layers immediately below UI provide use-Case app functions. Bottom layers proide general services like net-com and DB access. **Principles:** 1. layer can be independently designed. 2. well designed layers are cohesive. 3. lower layers do not know about upper layers. 4. you do not need to know about lower level implementation. 5. lower layers can be designed generically. 6. reuse layer built by others that provide needed services. 7. add new facilities built on lower-level services, or replace higher-level layers. 8. isolate components in separate layers to become resistant to obsolescence. 9. all dependent facilities can be isolated in one of the lower layers. 10. layers can be tested independently. 11. API's of layers are natural places to build rigorous assertions-checking.
Distributed/client-server: at least one component has the role of server, waiting for and then handling connections. At least one component has the role of the client, initiating connections in order to obtain some services.
Broker: transparently distribute aspects of software system to different nodes. An object can call methods of another object without knowing that this object is remotely located. CORBA is an open standard for this.



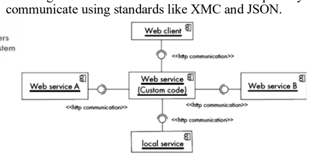


Transaction-processing: process needs series of inputs one-by-one. Each input describes a transaction-a command that changes the data stored by the system. There is a transaction dispatcher component that decides what to do with each transaction. This dispatches a procedure call or message to one of a series of components that will handle the transaction.
Pipe-and-filter: a steam of data, in relatively simple format is passed through a series of processes, each of which transforms it in some way. Data is constantly fed into the pipeline. Processes work concurently . Architecture is very flexible-->component removal, replacing, insertion, and reordering is possible





Model-view-controller(MUC): Separate UI layer from other parts of system. Model contains underlying classes whose instances are to be viewed/manipulated. The view contains objects used to render the appearance of the data from the model in the UI. The controller contains the objects that control and handle user's interaction with the view and the model. Observable design pattern normally used.
Service-oriented: Organizes an application as a collection of services that communicates using well-defined interfaces. Online, they're called web services-> an application, accessible online, that can be intergrated with other services to form a complete system. The different components generally communicate using standards like XMC and JSON.



	1	2	3	4	5	6	7	8	9	10	11
Multi-layers	■	■	■	■	■	■	■	■	■	■	■
Client-server	■	■	■	■		■	■		■	■	
Broker	■				■	■	■		■		■
Transaction processing	■	■	■				■				■
Pipe-and-filter	■	■	■	■	■	■	■			■	■
MVC	■	■	■			■	■			■	
Service-oriented	■	■	■	■	■	■		■	■	■	■
Message-oriented	■	■	■	■	■	■	■			■	■

Why: write deisng documents to help people known your design decision. It require constant effort to ensure a software systems design remains good through its lifetime. Make original design flexible. Ensure docs are usable and detailed.