Chapter 4

Domain Analysis is the process by which a software engineer learns background info.

Faster development: communicate with the stakeholders; establish requirements more rapidly; focus on the most important issues

Better System: subtleties; make fewer mistakes; better abstractions and hence improved designs

Anticipation of extensions: insights into emerging trends and notice opportunities for future development

BENEFITS OF DOMAIN ANALYSIS

- Faster development
- Better system
- Anticipation of extensions

DOMAIN ANALYSIS TEMPLATE

Summary
- Glossary
- General Knowledge
- Customers and Users
- The Environment
- Tasks and Procedures
- Competing Software
- Similarities to other domains

4.2The starting point for software projects

A **requirement** is a singular documented **physical** or **functional need** that a particular design, **product** or process **aims to satisfy**. It is a broad concept that could speak to any necessary (or sometimes desired) **function**, **attribute**, **capability**, **characteristic**, or **quality of a system** for it to have **value** and utility to a **customer**, organization, internal user, or other stakeholder.

Types of Requirements

FUNCTIONAL REQUIREMENTS

- Inputs
- Outputs
- Data (storage / other systems)

File format used to temporarily backup
- Computations
- Timing / Synchronization

QUALITY REQUIREMENTS

- Response time

Ex: result must be calculated in <3s
- Throughput

Computations or transactions per minute
- Resource usage

No more than 50MB of memory is used/CPU
- Reliability

A continuously running server must not suffer >1 failure in 6 months
- Availability

Server available over 99% no period of downtime >1 minute 6-9s
- Recovery from failure

Allow users to continue work after a failure of no more than 20 words of typing
- Allowances for maintainability and enhancement

Describe changes
- Allowances for reusability

40% must be designed generically so it can be reused

Platform

Constraints on environment and tech of system

Process

Constraints on project plan and development methods

TEMPLATE OF A USE CASE

A. **Name**: Short and descriptive.
B. **Actors**: Types of users that will do this
C. **Goals**: What are the actors trying to achieve.
D. **Preconditions**: State of the system before the use case.

E. **Summary**: Short informal description.
F. **Related use cases**.
G. **Steps**: Describe each step using a 2-column format.
H. **Postconditions**: State of the system after completion.

EXTENSIONS

- Used to make optional interactions explicit or to handle exceptional cases.
- Keep the description of the basic use case simple.

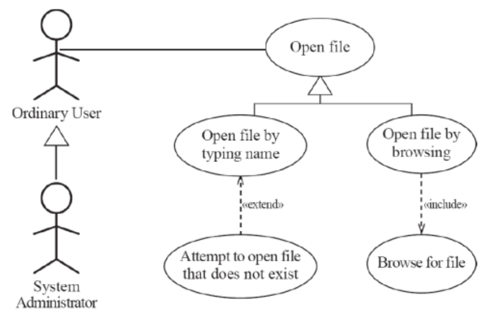
GENERALIZATIONS

- Much like super classes in a class diagram.
- A generalized use case represents several similar uses cases.
- One or more specializations provide details of the similar use cases.

INCLUSIONS

- Commonality between several different use cases.
- Are included in other use cases
- Even very different use cases can share sequence of actions.
- Enable you to avoid repeating details in multiple use cases.
- Lower-level task with a lower-level goal.

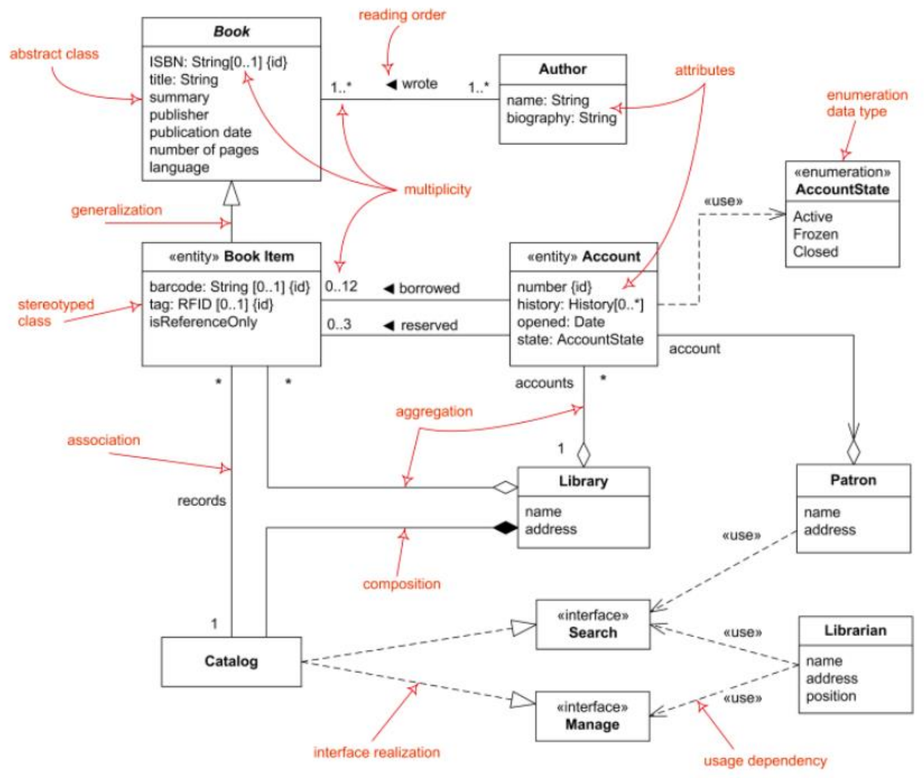
EXAMPLE OF GENERALIZATION, EXTENSION AND INCLUSION



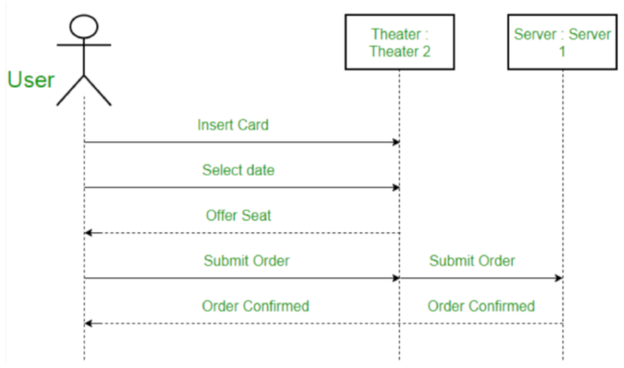
https://creately.com/blog/diagrams/use-case-diagram-relationships/

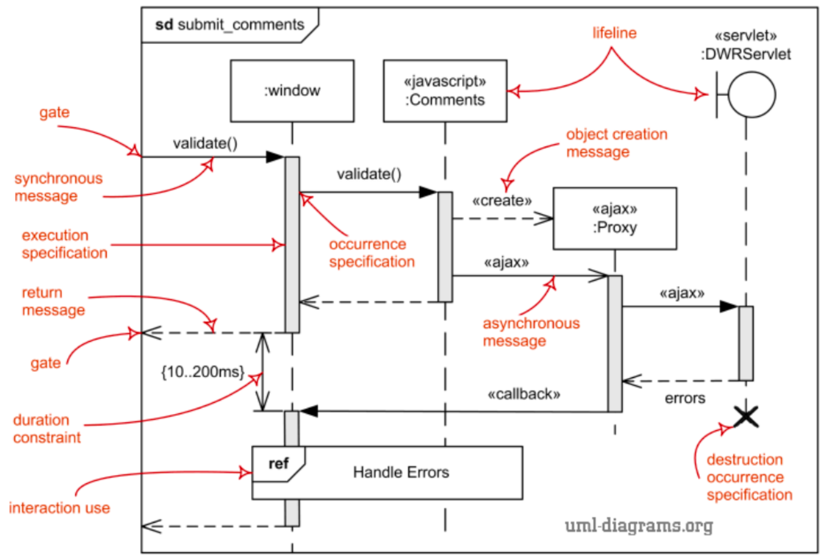
https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/

Chapter5

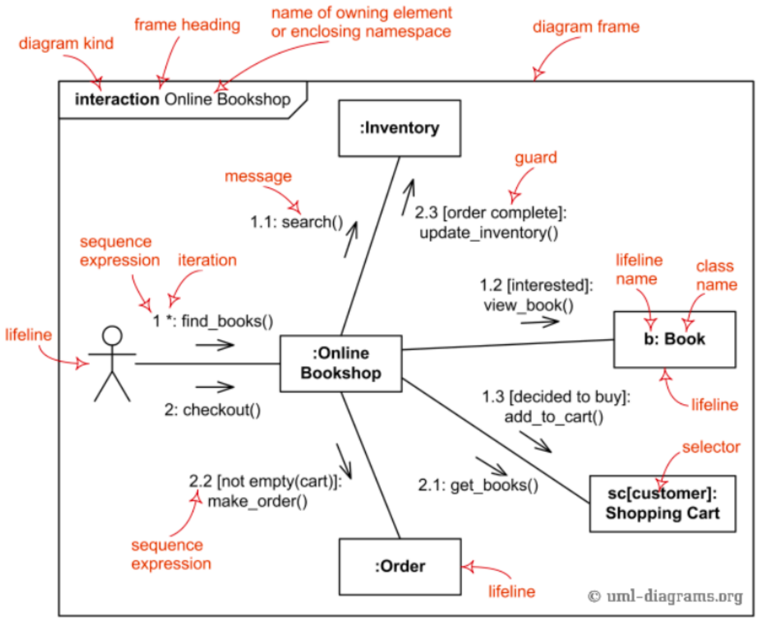


SEQUENCE DIAGRAMS (INTERACTIONS)

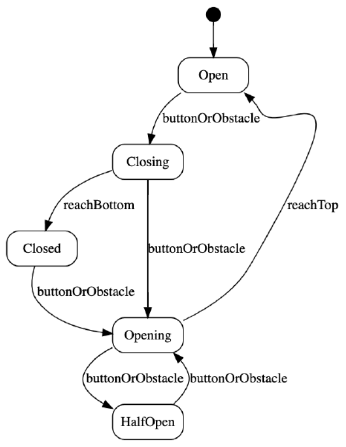




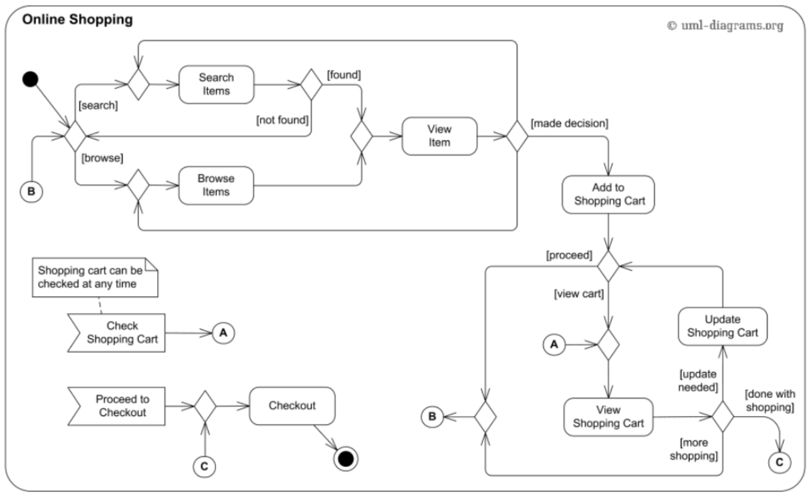
Communication Diagrams (Interactions)

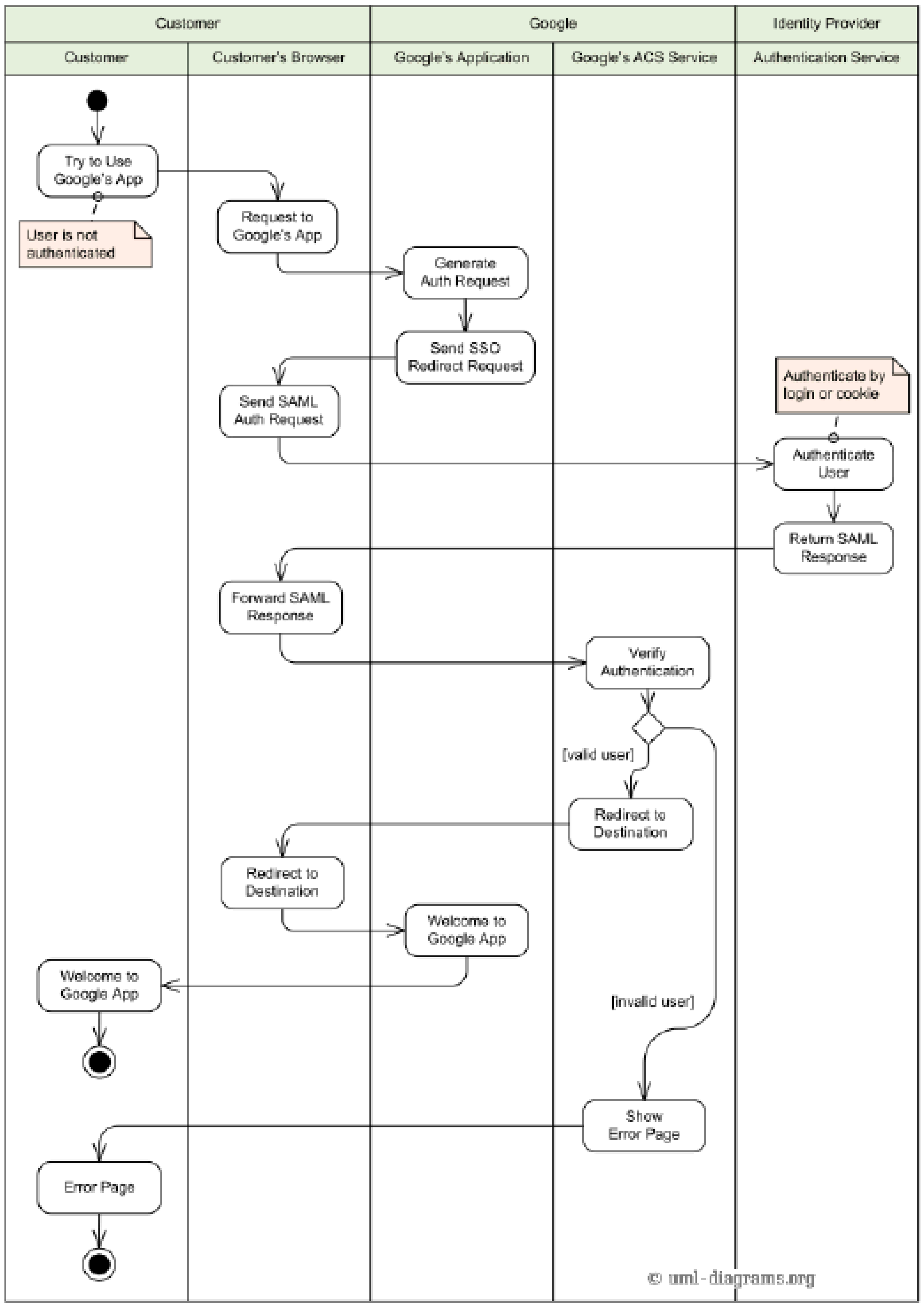


STATE DIAGRAMS

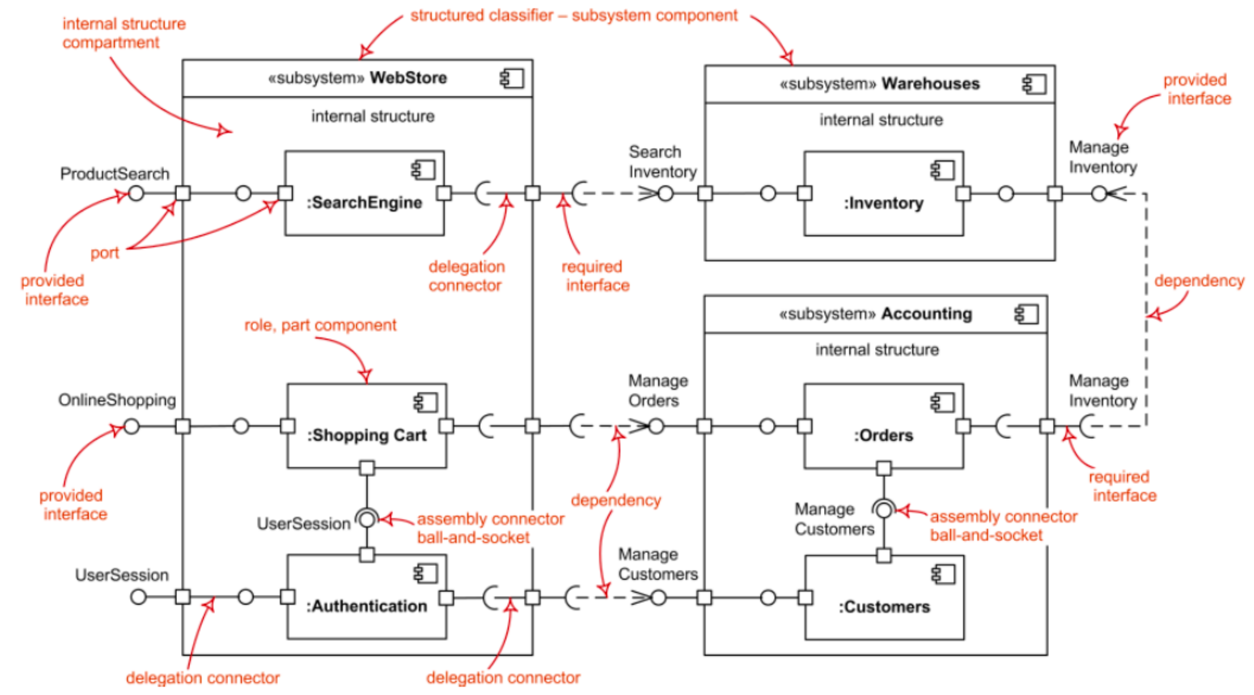


ACTIVITY DIAGRAMS

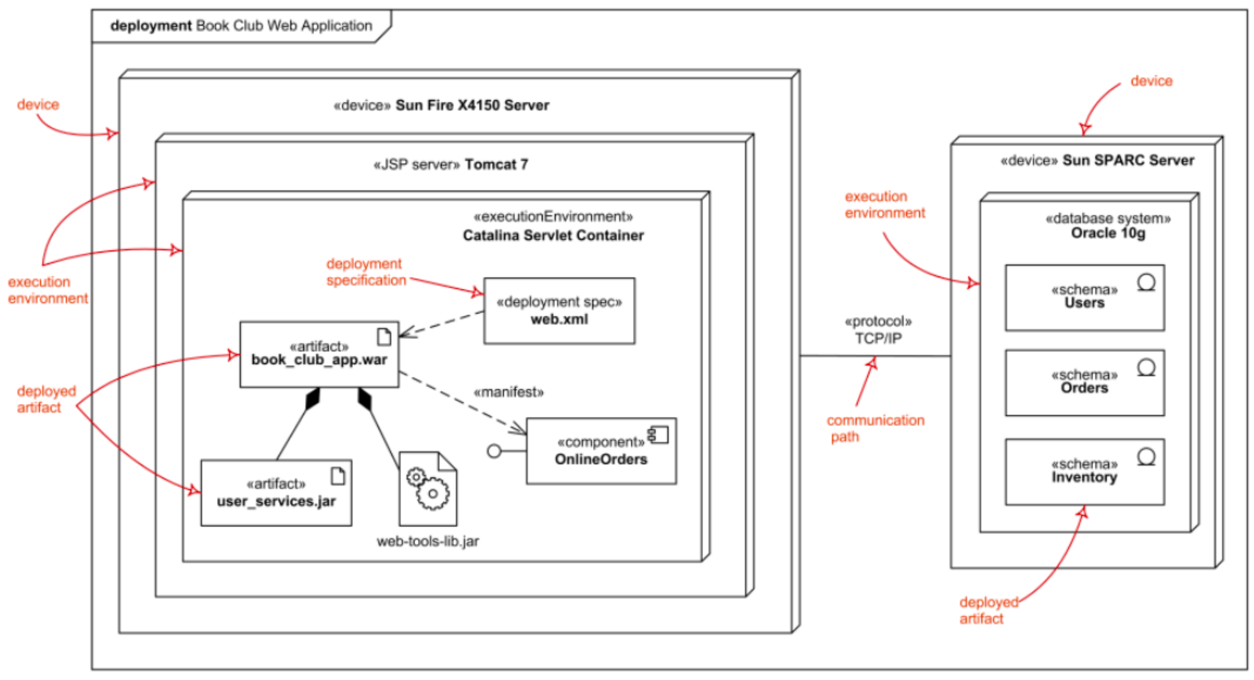



Customer
Google
Identity Provider
Customer
Customer's Browser
Google's Application
Google's ACS Service
Authentication Service
Try to Use Google's App
User is not authenticated
Request to Google's App
Generate Auth Request
Send SSO Redirect Request
Send SAML Auth Request
Authenticate by login or cookie
Authenticate User
Return SAML Response
Forward SAML Response
Verify Authentication
[valid user]
Redirect to Destination
Redirect to Destination
Welcome to Google App
Welcome to Google App
[invalid user]
Show Error Page
Error Page
© uml-diagrams.org

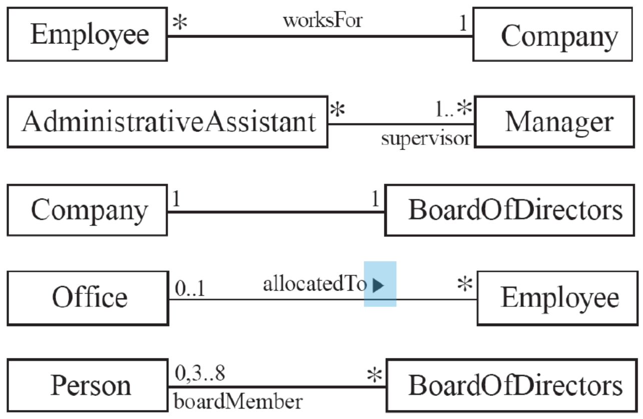
Component diagrams



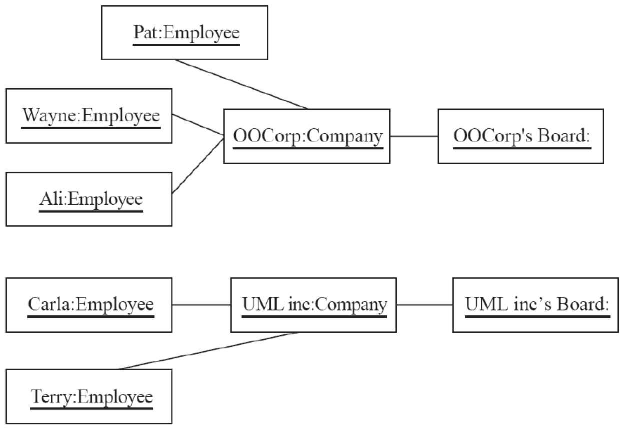
Deployment Diagram



Labelling associations



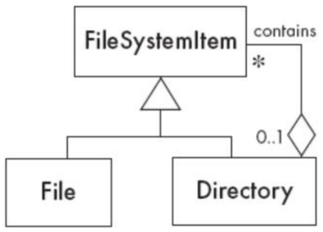
Object Diagrams



Chapter 6

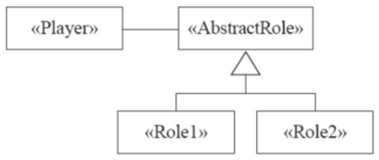
GENERAL HIERARCHY

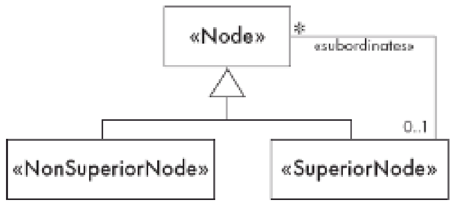
Example: File System

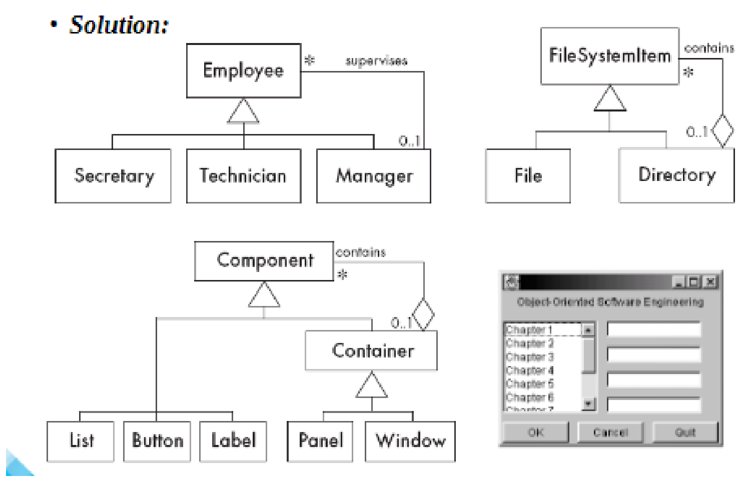
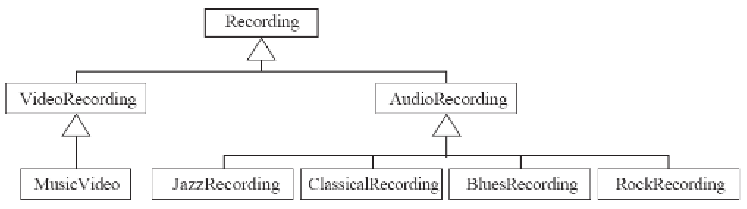


Play-Role

SOLUTION



General Hierarchy Pattern	
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 any 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
Solutions	• ***Solution:***  «Node» * «subordinates» 0..1 «NonSuperiorNode» «SuperiorNode»

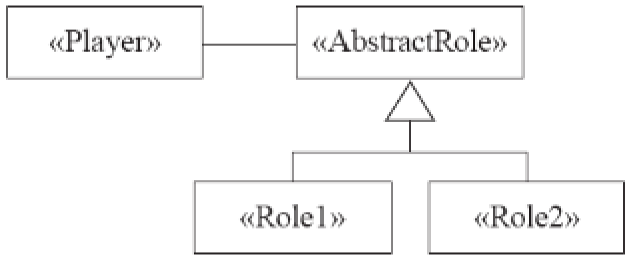
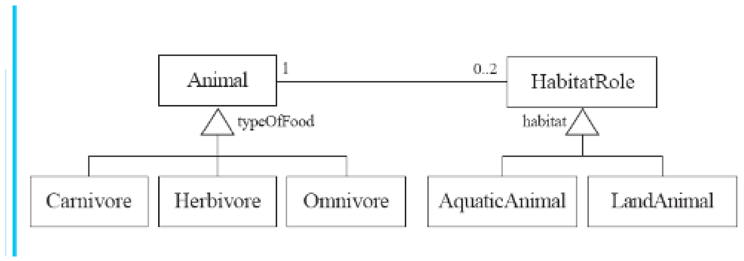
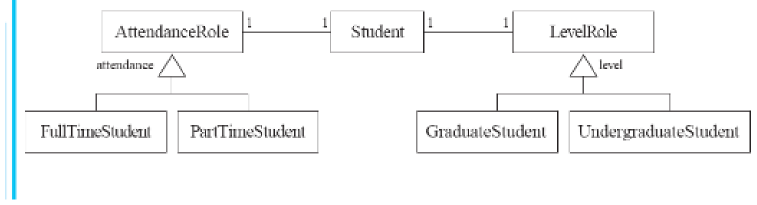
	 • Solution: Employee supervises * 0..1 Secretary Technician Manager FileSystemItem contains * 0..1 File Directory Component contains * 0..1 Container List Button Label Panel Window Object-Oriented Software Engineering Chapter 1 Chapter 2 Chapter 3 Chapter 4 Chapter 5 Chapter 6 OK Cancel Quit
Antipattern	**Antipattern:**  Recording VideoRecording AudioRecording MusicVideo JazzRecording ClassicalRecording BluesRecording RockRecording

The Player-Role Pattern

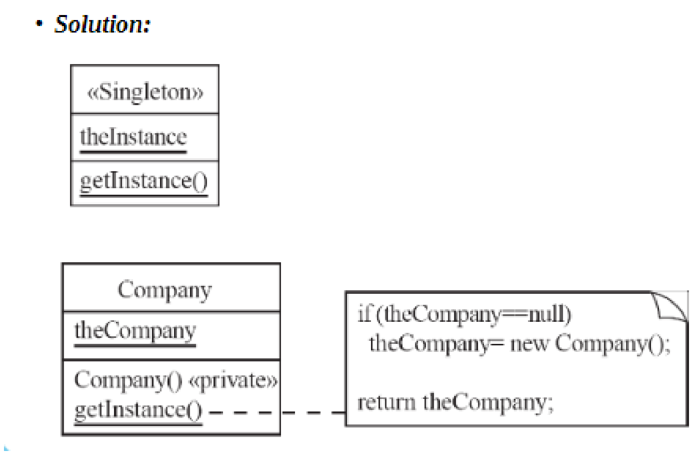
Context	A role is a particular set of properties associated with an object in a particular context An object may play different roles in different context

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 information associated w

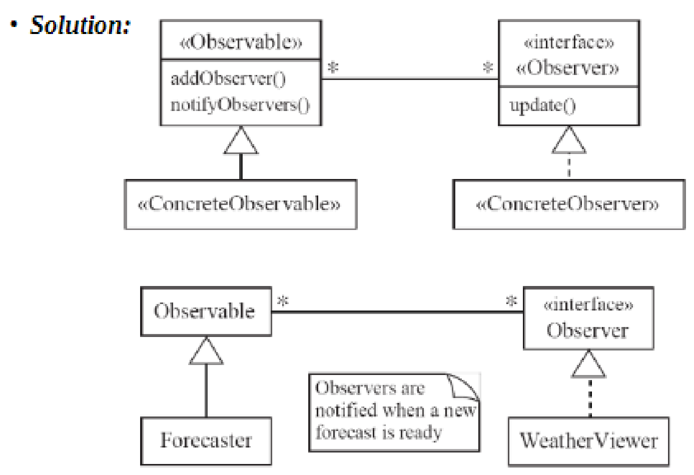
	You want to avoid multiple inheritance	You cannot allow an inst

Solution	
Examples	 Animal 1 0..2 HabitatRole typeOfFood habitat Carnivore Herbivore Omnivore AquaticAnimal LandAnimal **Example 2:**  AttendanceRole 1 1 Student 1 1 LevelRole attendance level FullTimeStudent PartTimeStudent GraduateStudent UndergraduateStudent
Antipatterns	Merge all properties and behaviours into a single "Player" class and not have "Role" classes at all Create roles as subclasses of the Player class

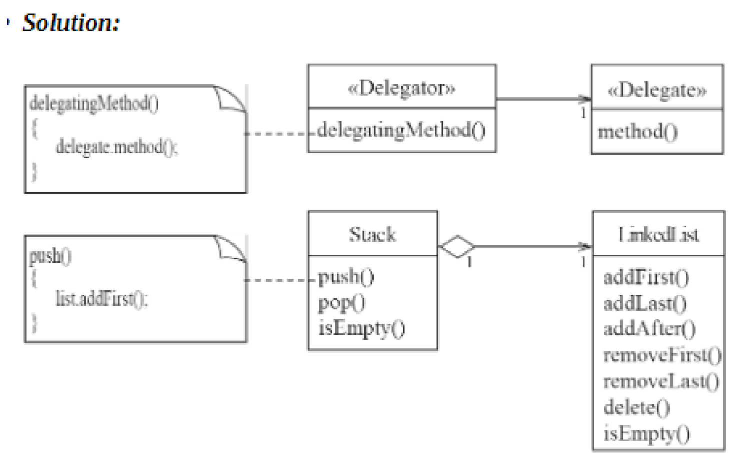
The Singleton Pattern

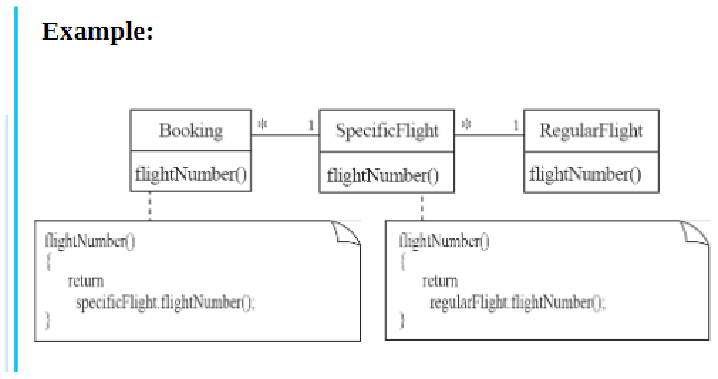
Context	It is very common to find classes for which only one instance should exist (singleton)
Problem	How do you ensure that it is never possible to create more than one instance of a singleton class
Forces	The use of a public cannot guarantee that no more than one instance will be created The singleton instance must also be accessible to all classes that require it
Solution	 Solution: «Singleton» theInstance getInstance() Company theCompany Company() «private» getInstance() if (theCompany==null) theCompany= new Company(); return theCompany;

The Observer Pattern	
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

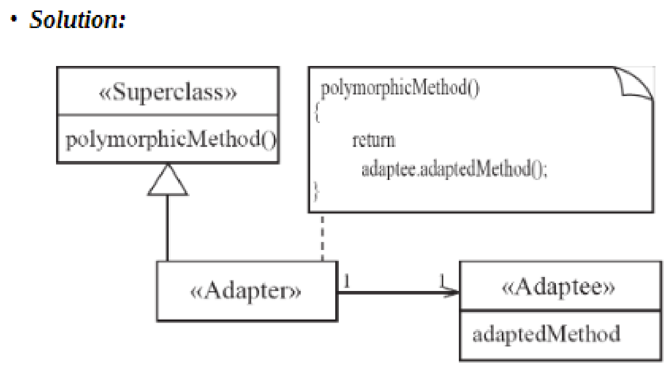
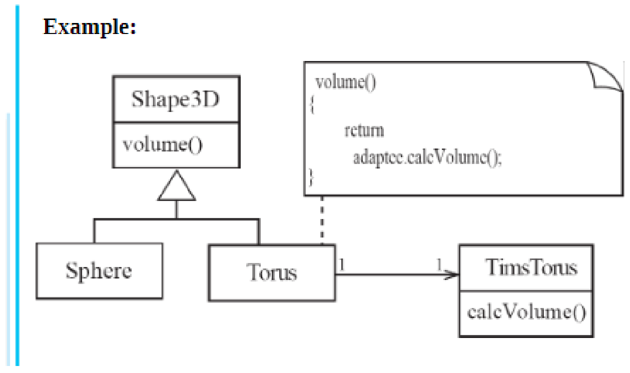
Forces	You want to maximize the flexibility of the system to the greatest extent possibble
Solution	 • Solution: «Observable» addObserver() notifyObservers() * * «interface» «Observer» update() «ConcreteObservable» «ConcreteObserver» Observable * * «interface» Observer Forecaster Observers are notified when a new forecast is ready WeatherViewer
Antipatterns	Connect an observer directly to an observable so that they both have references to each other Make the observers subclasses of the observable

The Delegation Pattern	
Context	You are designing a method in a class You realize that another class has a method which provides the required service Inheritance is not appropriate (ie. isa rule does not apply)
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

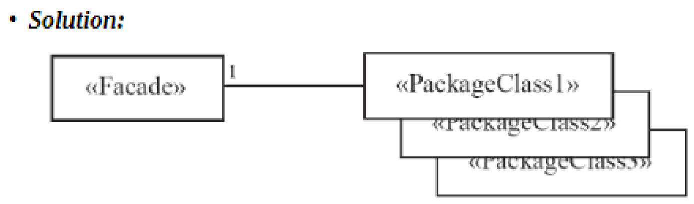
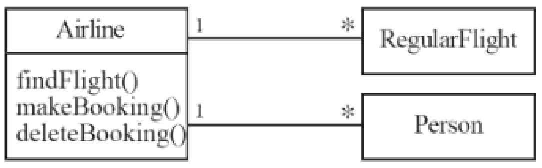
Solutions	 Solution:
	delegatingMethod() { delegate.method(); }
	«Delegator»
	delegatingMethod()
	«Delegate»
	method()
	1
	push() { list.addFirst(); }
	Stack
	push()
	pop()
	isEmpty()
	1
	1
	LinkedList
	addFirst()
	addLast()
	addAfter()
	removeFirst()
	removeLast()
	delete()
	isEmpty()

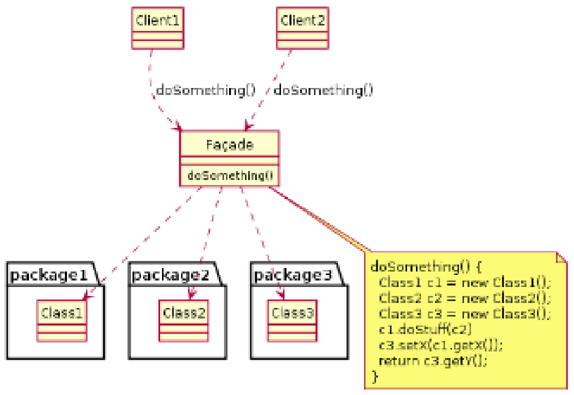
Example	 Example: Booking flightNumber() * 1 SpecificFlight flightNumber() * 1 RegularFlight flightNumber() flightNumber() { return specificFlight.flightNumber(); } flightNumber() { return regularFlight.flightNumber(); }
Delegation Antipatterns	Overuse generalization and inherit the method is to be reused Instead of creating a single method in the Delegator that does nothing other than call a method in the Delegate - Consider having many different methods in the Delegator call the delegate's method Access non-neighboring classes

The Adapter Pattern	
Context	You are building an inheritance hierarchy and want to incorporate it into 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 inheritance or you do not want to use it
Adapter	 • Solution: «Superclass» polymorphicMethod() polymorphicMethod() { return adaptee.adaptedMethod(); } «Adapter» 1 1 «Adaptee» adaptedMethod
Example	 Example: Shape3D volume() volume() { return adaptee.calcVolume(); } Sphere Torus 1 1 TimsTorus calcVolume()

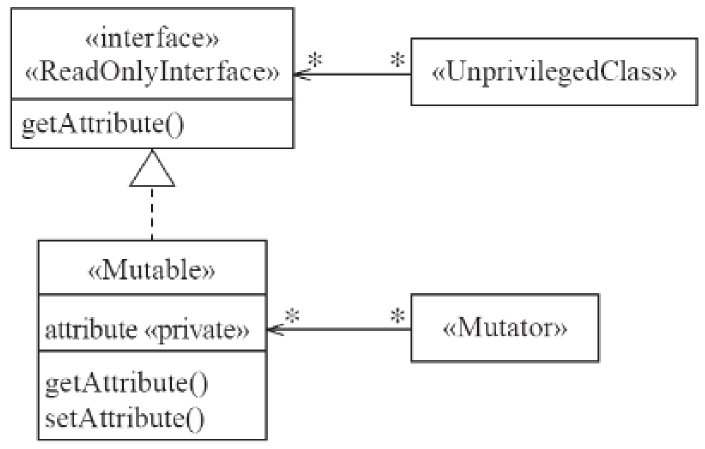
The Facade Pattern	
Context	Often, an application contains several complex packages

	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
Solution	 • Solution: «Facade» 1 «PackageClass1» «PackageClass2» «PackageClass3»  Airline findFlight() makeBooking() deleteBooking() 1 * RegularFlight 1 * Person

Facade	 Client1
	Client2
	doSomething()
	doSomething()
	Façade
	doSomething()
	package1
	Class1
	package2
	Class2
	package3
	Class3
	`doSomething() {`
	`  Class1 c1 = new Class1();`
	`  Class2 c2 = new Class2();`
	`  Class3 c3 = new Class3();`
	`  c1.doStuff(c2)`
	`  c3.setX(c1.getX());`
	`  return c3.getY();`
	`}`

Immutable Pattern	
Context	An immutable object is an object that has a state that never changes after creation
Problem	How do you create a 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 which access properties must not have side effect If a method that would otherwise modify an instance variable is required then it has to return a new instance of the class

Read-Only Interface Pattern	
Context	You sometimes want certain privileged classes to be able to modify attributes that are otherwise immutable
Problem	How do you create a situation where some classes see a class as read-only whereas others are able to make modifications
Forces	Restricting access to using the public, protected private keywords is not adequately selective Make access public makes it public for both reading and writing

Solution	• ***Solution:***  «interface» «ReadOnlyInterface» getAttribute() * * «UnprivilegedClass» «Mutable» attribute «private» getAttribute() setAttribute() * * «Mutator»

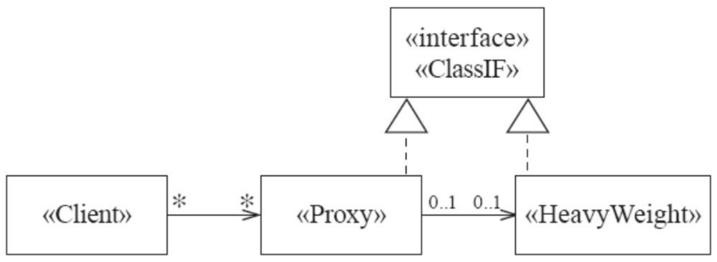
PROXY

CONTEXT ‣ Often, it is time-consuming and complicated to create instances of a class (heavyweight classes). ‣ There is a time delay and a complex mechanism involved in creating the object in memory

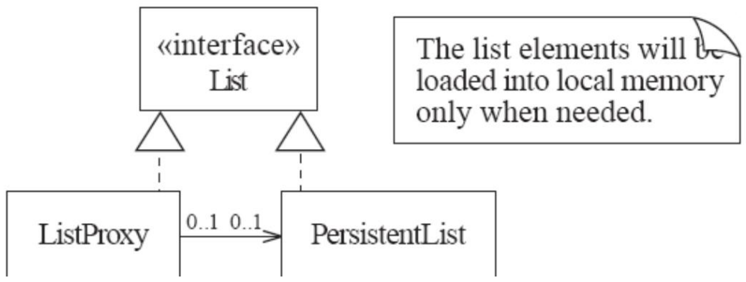
PROBLEM ‣ How to reduce the need to create instances of a heavyweight class?

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

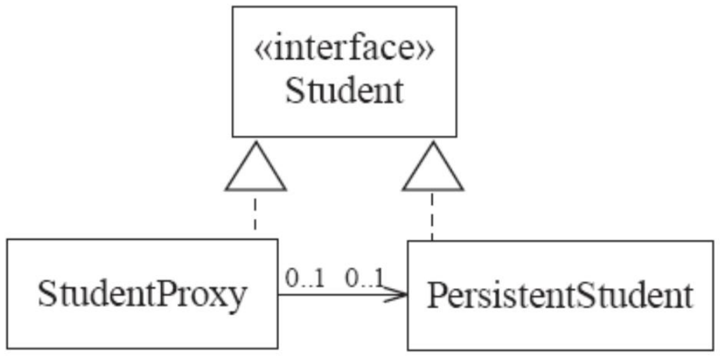
Solution



Example: Lists


«interface»
List
The list elements will be loaded into local memory only when needed.
ListProxy
0..1
0..1
PersistentList

Example: Student



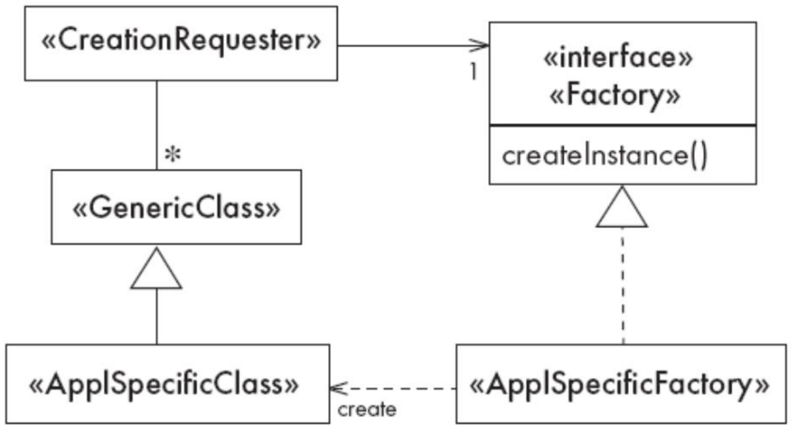
FACTORY

CONTEXT ‣ A reusable framework needs to create objecs; however the class of the created objects 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?

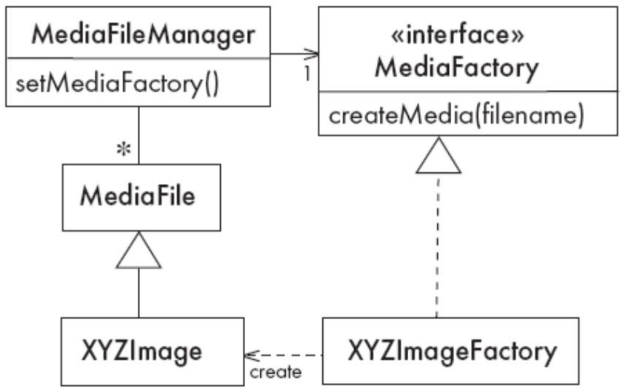
FORCES ‣ We want to have the framework create and work with application-specific classes that the framework does not yet know about.

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. ‣ The factory interface declares a method whose purpose is to create some subclass of a generic class.

Example: Media



Chapter 8

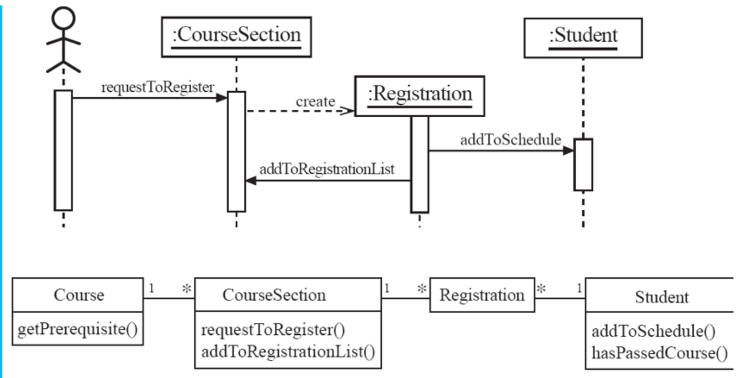
Activity Diagram

https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/

Interaction diagrams are used to model the dynamic aspects of a software system

- They help you to visualize how the system runs.
- An interaction diagram is often built from a use case and a class diagram.
  - The objective is to show how a set of objects accomplish the required interactions with an actor.

Sequence diagrams – an example

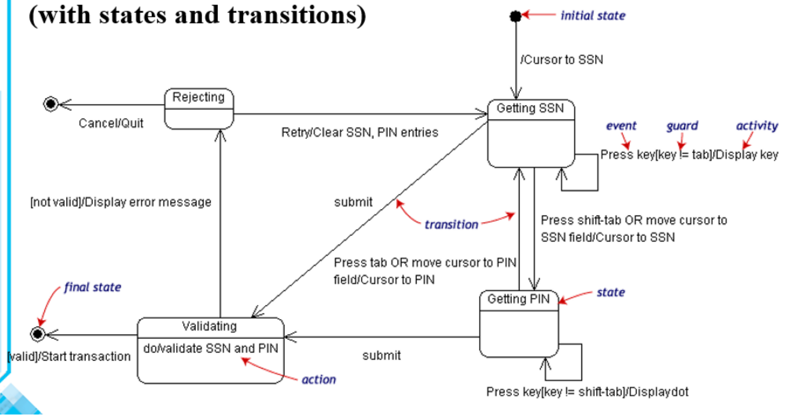


Allow multiple sequences to be represented in compact form (may involve all participants or just a subset).

- alt, for alternatives with conditions
- opt, for optional behavior
- loop(lower bound, upper bound), for loops
- par, for concurrent behavior Two or more operands that execute in parallel
- ref, for referencing other sequence diagrams

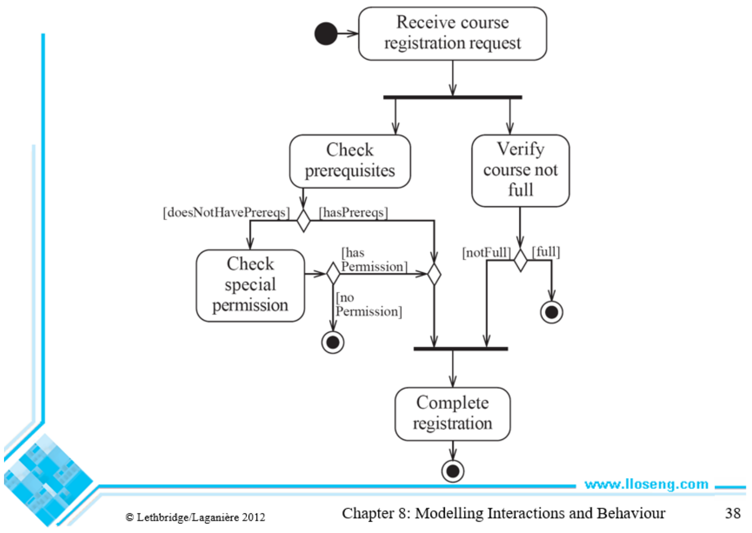
Basic Notational Elements of State Machine Diagrams

Describe the dynamic behavior of an individual object (with states and transitions)



initial state
/Cursor to SSN
Rejecting
Cancel/Quit
Retry/Clear SSN, PIN entries
Getting SSN
event
guard
activity
Press key[key != tab]/Display key
[not valid]/Display error message
submit
transition
Press shift-tab OR move cursor to SSN field/Cursor to SSN
Press tab OR move cursor to PIN field/Cursor to PIN
final state
state
Getting PIN
Validating
do/validate SSN and PIN
[valid]/Start transaction
submit
action
Press key[key != shift-tab]/Displaydot

Activity diagrams – an example



Concurrency is shown using forks, joins and rendezvous.

- A *fork* has one incoming transition and multiple outgoing transitions.

- The execution splits into two concurrent threads.

- A *rendezvous* has multiple incoming and multiple outgoing transitions.

- Once all the incoming transitions occur all the outgoing transitions may occur.

Chapter 9

DESIGN SPACE 6 ‣ The space of possible designs that could be achieved by choosing different sets of alternatives is often called the **design space**

COMPONENT ‣ Software or Hardware that has a clear role ‣ Isolated to allow its replacement ‣ Much harder in software (e.g. MySql vs Postgres) ‣ Much easier in hardware (e.g. Replacing servers) ‣ Designed to be re-usable ‣ Some are special-purpose **8**

MODULE ‣ A component at the programming language level ‣ In Java, ‣ Packages, ‣ Classes, and ‣ Methods

SYSTEM ‣ Logical entity ‣ Definable responsibilities (or objectives) ‣ Hardware, software or both ‣ Can have ‣ Specification (and then implemented by a components) ‣ Exists even if components are changed (or replaced) ‣ Requirements analysis identifies the responsibilities of a system

TOP-DOWN DESIGN ‣ Design very high level structure ‣ Work down to detailed decisions about low-level constructs ‣ Arriving at detailed decisions such as: ‣ Data formats ‣ Algorithms ‣ API Interfaces

BOTTOM-UP DESIGN ‣ Decisions about re-usable low-level utilities ‣ Decisions on piecing them together ‣ Creating higher-level constructs

MIXING TOP-DOWN AND BOTTOM-UP ‣ A mixture of both normally used ‣ Top-Down gives a good structure ‣ Bottom-Up promotes re-usable components

FUNCTIONAL COHESION This is achieved when all the code that computes a particular result is kept together - and everything else is kept out ‣ i.e. when a module only performs a single computation, and returns a result, without having side-effects. ‣ Benefits to the system: ‣ Easier to understand ‣ More reusable ‣ Easier to replace ‣ Modules that update a database, create a new file or interact with the user are not functionally cohesive

LAYER COHESION All the facilities for providing or accessing a set of related services are kept together, and everything else is kept out ‣ The layers should form a hierarchy ‣ Higher layers can access services of lower layers, ‣ Lower layers do not access higher layers ‣ The set of procedures through which a layer provides its services is the application programming interface (API) ‣ You can replace a layer without having any impact on the other layers ‣ You just replicate the API

COMMUNICATIONAL COHESION All the modules that access or manipulate certain data are kept together (e.g. in the same class) - and everything else is kept out ‣ A class would have good communicational cohesion ‣ if all the system’s facilities for storing and manipulating its data are contained in this class. ‣ if the class does not do anything other than manage its data. ‣ Main advantage: When you need to make changes to the data, you find all the code in one place

SEQUENTIAL COHESION Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out ‣ You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

PROCEDURAL COHESION Procedures that are used one after another are kept together ‣ Even if one does not necessarily provide input to the next. ‣ Weaker than sequential cohesion.

TEMPORAL COHESION Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out ‣ For example, placing together the code used during system start-up or initialization. ‣ Weaker than procedural cohesion.

UTILITY COHESION When related utilities which cannot be logically placed in other cohesive units are kept together ‣ A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable. ‣ For example, the **java.lang.Math** class

CONTENT COUPLING Occurs when one component surreptitiously modifies data that is internal to another component ‣ To reduce content coupling you should therefore encapsulate all instance variables ‣ declare them private ‣ and provide get and set methods ‣ A worse form of content coupling occurs when you directly modify an instance variable of an instance variable

COMMON COUPLING Using a global variable ‣ All components using the global are now coupled ‣ A weaker form is when a variable can be accessed by a subset of the system's classes ‣ e.g. a Java package ‣ Acceptable for global variables of system-wide default values ‣ The Singleton pattern encapsulates global access to an object, but it is still a global.

CONTROL COUPLING Calling a procedure with a '**flag**' or '**command**' to control its behaviour ‣ To make a change you have to change both the ‣ Calling method (that send the command) and ‣ Called method (that interprets the command) ‣ Polymorphic operations a good way to avoid control coupling ‣ One way to reduce control coupling ‣ Using a look-up table ‣ Commands are mapped to a the callable method

STAMP COUPLING An application class is declared as the type of a method argument ‣ Since one class now uses the other, ‣ changing either becomes harder ‣ Reusing one class requires reusing the other ‣ Two ways to reduce stamp coupling, ‣ Use an interface as the argument type ‣ Passing simple variables (beware of content coupling) **41**

DATA COUPLING The types of method arguments are primitive or simple library classes ‣ The more arguments, the higher the coupling ‣ Using that method means passing all those arguments ‣ Avoid it by limiting unnecessary arguments ‣ There is a trade-off (increasing one often decreases the other) between ‣ Data Coupling and ‣ Stamp coupling

ROUTINE CALL COUPLING One routine (or method) calls another ‣ The routines are coupled as they depend on each other's behaviour ‣ Routine call coupling is always present in any system. ‣ But should still be considered as coupling ‣ Is sequences of methods are use repeatedly then reduce call coupling by encapsulating into a single method

TYPE USE COUPLING A module uses a data type defined in another module ‣ Any time a class declares a variable of another class' type ‣ If the type definition changes, then the declaring classes might be affected ‣ Always declare the type of a variable to be the most general possible class or interface that contains the required operations

INCLUSION OR IMPORT COUPLING One component imports a package (as in Java) or includes another (as in C++). ‣ The including/importing component is now ‣ exposed to everything in the included or imported component. ‣ If the included/imported component changes ‣ The including/importing component might have to change ‣ Can cause conflict forcing a change (e.g. duplicate names)

EXTERNAL COUPLING A module has a dependency on external things like the operating system, shared libraries or hardware ‣ Reduce the number of places where such dependencies exist (high cohesion). ‣ The Façade design pattern helps

Chapter 10

Failure: an unacceptable behavior exhibited by a system

Error: decision by a software developer leads to a defect

Defect: a fault, flaw

Black-box vs white box testing: white box is more time consuming than black box but removes much of the guesswork and allows the tester to be more thorough.

Control flow-graph:

BLACK-BOX TESTERS HAVE ACCESS TO … THE SYSTEM ‣ Provide the system with inputs ‣ Observer outputs ‣ No access to ‣ Source code ‣ Internal data ‣ Documentation relating to the systems internals

QUIVALENCE CLASSES ‣ Impossible to test by brute force using all input values ‣ e.g. every integer ‣ Instead, divide the possible inputs into groups ‣ Should be treated similarly ‣ Known as **equivalence classes** ‣ A tester needs only to run 1-3 tests per equivalence class ‣ You must understand the required input ‣ Appreciate how software may have been designed

BIG BANG TESTING VERSUS INTEGRATION TESTING

‣ In big bang testing,

‣ you take the entire system and test it as a unit

‣ A better strategy in most cases is incremental testing:

‣ You test each individual subsystem in isolation

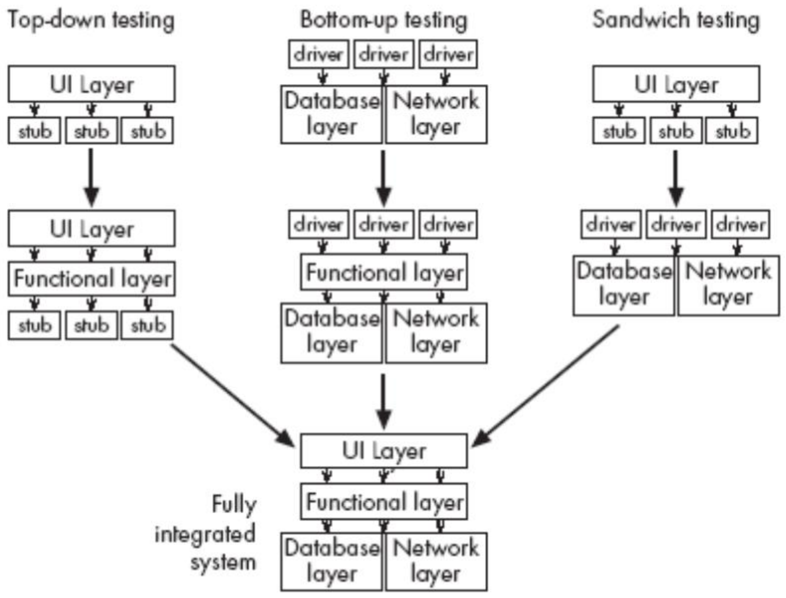
‣ Continue testing as you add more and more subsystems to the final product

‣ Incremental testing can be performed horizontally or vertically, depending

on the architecture

‣ Horizontal testing can be used when the system is divided into separate sub-applications

VERTICAL STRATEGIES FOR INCREMENTAL INTEGRATION TESTING



REGRESSION TESTING

‣ When testing manually ‣ Test a well-chosen subset of the previously successful test cases ‣ When testing automatically ‣ Run all tests, since it is generally fast