Owen Daigle - SEG2105 Final Cheat Sheet

CH1

Real time embedded software

- E.g. control and monitoring systems
- Must react immediately
- Safety often a concern

Data processing software

- Used to run businesses
- Accuracy and security of data are key

Requirements and specification Includes

1. Domain analysis
2. Defining the problem
3. Requirements gathering
    1. Obtaining input from as many sources as possible
4. Requirements analysis
    1. Organizing the information
5. Requirements specification
    1. Writing detailed instructions about how the software should behave

CH2

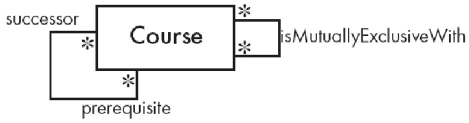
Dynamic binding occurs when we need to wait till runtime to decide which method to run.

- A variable is declared to have a superclass as its type, and
- There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses

public - Any class can access
protected - Only code in the package, or subclasses can access
(blank) [**package**] - Only code in the package can access
private - Only code written in the class can access
- Inheritance still occurs!

CH5

A class can be associated to itself (**reflexive**)



A course can have multiple successors, and multiple prereqs. It is not **symmetric** (if A is prereq for B, B is not prereq for A).
MutuallyExclusive is symmetric since if A is mutuallyExclusive with B, then B is ME with A.

We create associations if we see:

- __ possesses/controls __
- __ is connected/related to __
- __ is a part of __
- __ is a member of __
- __ is a subdivision of __

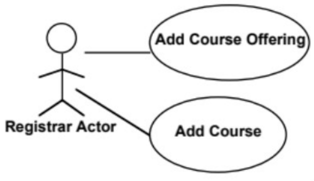
We create classes if:

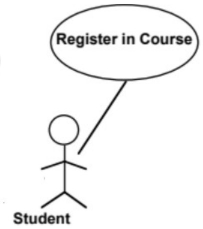
- __ is a __

CH4

A requirement is something the system must do, or a constraint on the system.
Such as: *The online banking system must allow the user to access their acc balance in less than 5s.*
Functional Reqs are what I/O the system takes, what data should it store ETC.
Non Functioal reqs are quality constraints such as response time, OS compatibility, etc.

A use case is a sequence of actions from a user to do a task.





Design Patterns are generic solutions to problems.

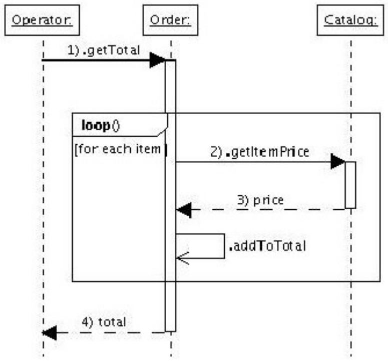
CH6

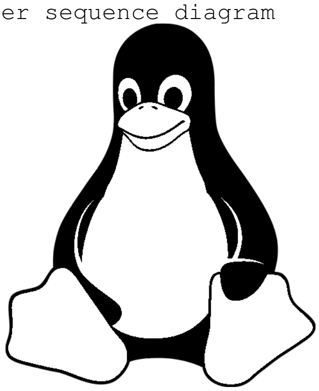
Abstraction Occurence means one item has multiple sub items: *A TV series has multiple episodes.* (Series has many associations to episodes)
General Hierarchy means an object has superiors of same type and sub of same type: *Employees have a manager that is also employee.* (Manager is subclass of employee, manager has multiple employee associations)
Player Role means someone can play multiple roles. We have an abstractRole class with multiple role subclasses. Player can have multiple roles.
Singleton means only possible to create 1 instance
Observer means one class (observer) waits. The observable class will ping the observer when it has more data. * observers, * observables.
Delegation means one class calls another to do a heavy operation: *A linked stack delegates ops to a linked list class.*
Adapter means have a function that calls another function and adapts its signature. *F1 has signature INT, we need sig STR, so make F2 that calls F1 and casts STR to INT.*
Facade means have one class that interacts with the whole system and exposes a standard API.
Immutable means ensure constructor is only place that vars can be changed.
Read Only Interface means have multiple interfaces to access the object, each one exposes different methods: *A read only interface not expose setters.*
Proxy means we have a heavyweight class with complex computations and simple data. To access the data, we go through a proxy class that does not load the heavyweight into memory.
Factory means a class who is designed to create subclasses of a generic class.

CH8

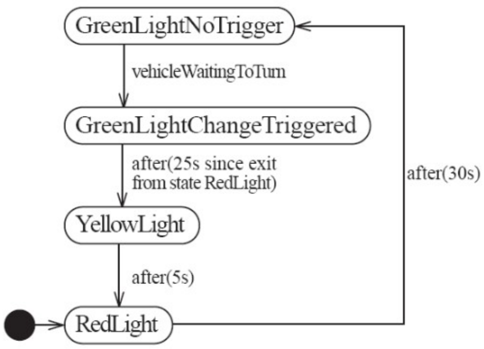
A **sequence diagram** shows messages exchanged by objects doing a task.
e have the following optional fragments

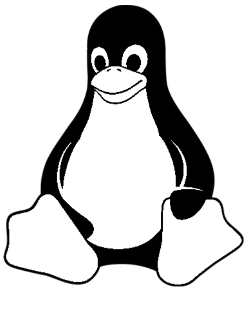
- **alt** for alternatives w conditions (if else)
- **opt** for optional behaviour (if)
- **loop** from _ to __ loop
- **par** for concurrent (parallel) behaviour
- **ref** to reference another sequence diagram





State Diagram show the state of **1** part in a system





An **Activity diagram** is like a state diagram but transitions are caused by internal events like computations.

CH9

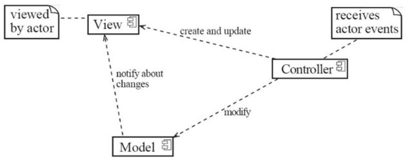
A **Component** is a piece of hardware or software with a clear role.
A component can be isolated to be replaced with one of similar functionality.

With **bottom up** design, we start with low level implementations.
With **top down** desgign we start with general structure, then work down to the exact implementation later on.

We have a list of design principles.

1. **Divide and Conker** means seperate into many parts. Parts are modular with agreed upon sig
2. **Increase Cohesion: Functional** (*code that performs same computations together*), **Layer** (*All facilities providing access to a service are together*), **Communicational** (*All modules that access certain data are together*), **Sequential** (*One procedure provides input to another, these go together*), **Procedural** (*Procedures one after another are together*), **Temporal** (*Operations during same phase of execution are together*), **Utility** (*related utilities are together*).
3. **Reduce Coupling: Content** (*one component modifies internal data to another one, fix by private vars*), **Common** (*Using global variables, fix by singleton*), **Control** (*We use a "flag" to control a method, fix by using polymorphic methods*), **Stamp** (*Use a class as a parameter for another method, fix by either using interface, or using simple variables*), **Data** (*More method arguments increases data coupling, fix by passing an object [tradeoff with Stamp]*), **Routine Call** (*One routine calls another*), **Type Use** (*Module uses a data type defined elsewhere, fix by defining a variable to be the most generic possible*), **Import** (*When we import a package, fix by only importing what we use*), **External** (*Dependancy on OS/hardware/etc, fix by reducing this*)
4. **Increase Abstraction**: use classes, interfaces, etc. These make it easier to understand the system.
5. **Increase Reusability**: generalize design, add hooks, use a simple design.
6. **Reuse**: reuse designs, frameworks, and code snippets rather than redoing the work.
7. **Flexibility**: Leave all options open for future.
8. **Anticipate Obscelecence**: Avoid early releases, or legacy software. Use standard popular software.
9. **Portability**: Dont use utilities only for one platform like ms windoze 😱😱😱
10. **Testability**: design program to test
11. **Defensively**: Never trust a client, or even other developers.

This comapres the arch pattern to the design prnpl
1.**MultiLayer**: UI layer, Logic layer, DB layer
2.**ClientServer**: Some clients, Some servers, can be p2p.
3.**Broker**: Client interacts with broker which interacts with remote. Broker does serializing.
4.**TransactionProcessing**: There is an imput which goes to a **dispatcher**. This dispatcher will route the transaction to the correct place.
5.**PipeFilter**: Data goes though a series of processes, each transforms in some way.



6.**MVC(ModelViewController)**: Seperates UI from rest of system. View is front end, controller receives events and modifies model, model is all the classes and db...

6.**WebServiceOriented**: An application is accessible over the internet using APIs, and stuff.

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

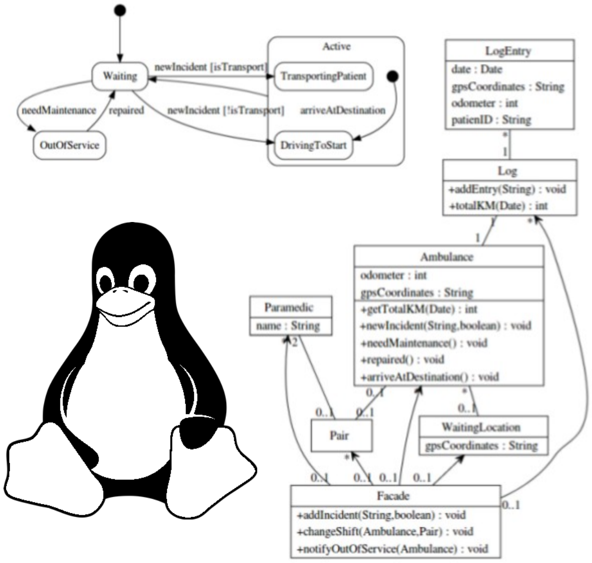
EXAMPLES

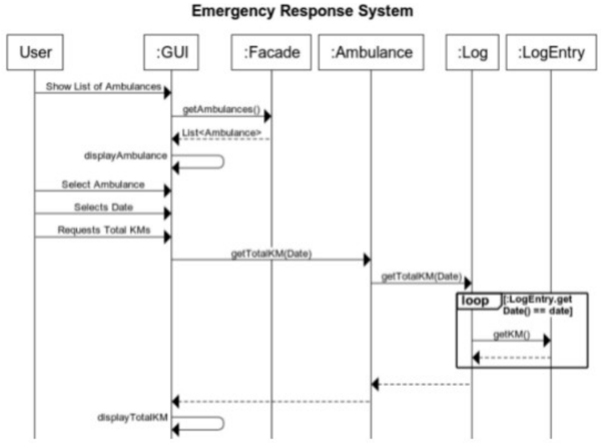
• An emergency response service has a system to manage a fleet of ambulances. • The person in charge of the system assigns two paramedics as the crew of each ambulance that is in service. Each pair of paramedics works a shift in an ambulance and then passes the ambulance to the next pair of paramedics. • Sometimes an ambulance goes 'out of service' when it needs maintenance. Ambulances constantly inform the system of their location as determined by a GPS receiver. • There are two types of tasks that an ambulance crew may have to perform. The first is to respond to an emergency call – to transport a patient from an initial point (for example the site of an accident) to a hospital. The second type of task is to transfer a sick patient from one hospital to another. When performing a task, the ambulance is said to be 'active'. When active, the ambulance is either 'driving to the starting point' or 'transporting a patient'. • When they are not active on a task, each ambulance crew is told to drive to a waiting place somewhere in the city. There are 15 waiting places, and the system chooses the best waiting place so as to best spread the ambulances around the city. When an ambulance is required for a task, the nearest inactive ambulance is assigned that task automatically. • For each step in a task, the emergency response service maintains a permanent record of the date, the time, the location, the odometer reading of the ambulance and the file number of any patient being transported.

Design a state diagram for the ambulance.

Create use case to find how many km one ambulance drove in 1 day

1 - The user asks for the list of ambulances	2 - The system displays the list
3 - The user selects one of the ambulances	
4 - The user specifies a date and	5 - The system displays the number of kms


Waiting
newIncident [isTransport]
Active
TransportingPatient
needMaintenance
repaired
newIncident [!isTransport]
arriveAtDestination
OutOfService
DrivingToStart
LogEntry
date : Date
gpsCoordinates : String
odometer : int
patienID : String
*
1
Log
+addEntry(String) : void
+totalKM(Date) : int
1
1
*
Ambulance
odometer : int
gpsCoordinates : String
+getTotalKM(Date) : int
+newIncident(String,boolean) : void
+needMaintenance() : void
+repaired() : void
+arriveAtDestination() : void
Paramedic
name : String
2
0..1
0..1
0..1
*
0..1
Pair
WaitingLocation
gpsCoordinates : String
*
0..1
0..1
0..1
0..1
0..1
Facade
+addIncident(String,boolean) : void
+changeShift(Ambulance,Pair) : void
+notifyOutOfService(Ambulance) : void


Emergency Response System
User
:GUI
:Facade
:Ambulance
:Log
:LogEntry
Show List of Ambulances
getAmbulances()
List<Ambulance>
displayAmbulance
Select Ambulance
Selects Date
Requests Total KMs
getTotalKM(Date)
getTotalKM(Date)
loop
[:LogEntry.getDate() == date]
getKM()
displayTotalKM