2212 Notes
2015, B. Locke

Topic 1: Software and Software Engineering

Software:
- intangible (not physical)
- reproducible
- hard to automate
- can be very “hacky”
- easily modified
- no deterioration, but hard to upgrade

Software Quality (seven aspects of software quality)
i) Usability (does it fulfil its purpose)
ii) Efficiency (does it waste a lot of computer resources)
iii) Compatibility (does it work across platforms)
iv) Reliability (is it fault-tolerant)
v) Maintainability (can it be changed w/o minimal effort)
vi) Security (is it secure)
vii) Reusability (is it reusable)

Types of Software:
i) Custom
In-house is most common (i.e. TD bank back-end software)
Customized software for a specific customer
ii) Generic
Open source example
iii) Embedded
Firmware
iv) Real-time software
Soft: performance degrades if deadline misses
Hard: deadlines MUST be met for performance (i.e. hospital software, defense software)
v) Data processing software
Security analysis of data

Difference between Software Engineering and real engineering is:
Software engineering is: solving customer problems w/ development of large, high quality systems within a cost/time and other constraints.

High quality software systems need:
i) Software engineering techniques
ii) Allocation of work
iii) System cohesion
iv) End product must be sufficient quality

Stakeholders in Software projects are:
Users
Customers/Clients
Developers
Development Managers

2212 Notes
2015, B. Locke

User Interests in Software:
- ease of use
- helps with efficiency in solving a task

Customer/Client Interests in Software:
- acceptable cost to solve a problem

Developer Interests In Software:
- selling of software, with a low maintenance and upgrade cost

Development Manager in Software:
- easy to design
- easy to maintain
- easy to reuse

Tradeoffs of Software Quality:
i) Improving efficiency comes at cost of:

ii) Improving reliability at cost of:

iii) Improving security at cost of:

^Check piratepad information here

Internal Quality Criteria:
i) SLOC (commenting of code)
ii) Complexity of code
iii) Coupling of code
- Does code rely on other parts? i.e. global variables etc

SLOC (commenting of code)
What are two metrics to measure quality?
Short term:
i) Does it meet immediate needs?
ii) Is it evident for volume of data presently

Long term:
i) Maintainability
ii) Continually meeting customer needs

What are Activities that need to be completed over software development?
i) Requirements and Specification
ii) Design
iii) Modelling
iv) Programming
v) Quality Assurance
vi) Deployment
vii) Management of development process

i) Requirements and Specification

- What is the program designed to do?
- What information is necessary before designing it?
- What are the requirements of the software?

ii) Design

- How are the requirements going to be implemented?
- What software/What hardware should be used?
- Dividing system into subsystem and determining their interaction
- How is data stored?
- How is the user going to interact with software?

iii) Modeling

- How do we represent how the software functions over a domain of inputs?

iv) Programming

- Self explanatory

v) Quality Assurance

- How to test the program
- How to review the program

vi) Management of development process

Answer to Study Questions (click)

Topic 2: Managing the Software Process

Part 1) Software Process Models

Jieni's Notes:
https://docs.google.com/document/d/1Af4ov9Irv0zOtVLun3rwHOsCMo3i9SZ9qMHvGNFEjrg/edit?usp=sharing

Project Management describes:

- All the activities needed to plan/execute a project

Models of Project Management Include:

i) Optimistic Approach
ii) Waterfall Model
iii) Spiral Model
iv) Agile Model
  a. eXtreme Programming (XP)
  b. Scrum

The Optimistic Approach:

Known as when organization does NOT follow good development practices (code and fix)

Drawbacks of the Optimistic Approach:
No recognition of planning before implementation
No aim for the program
No explicit goals therefore no explicit targets for quality assurance
Maintaining/developing costs are high

The Waterfall Model:

Classic way that suggests development occurs in a series of stages
Engineers work in series of stages, and perform quality assurance

Drawbacks of The Waterfall Model
Document driven, therefore lots of documentation
Low interaction with customers after requirements are set
Stages MUST be completed before progressing (hence waterfall)
Nothing is usable until program is complete
No prototyping
Implies that only maintenance is present after product is finished

The Spiral Model:

An iterative and prototyping driven approach to software development
Prototype generation, along with mini-waterfall process
Prior to each implementation, risk analysis is performed
Every iteration, project is implemented heavier

Drawbacks of The Spiral Model
(Check piratepad)

The Agile Model:

Development of small iterations

Benefits of Agile Model
Test-driven development helps to implement projects that are high-risk or have uncertain requirements
Users are involved at ALL stages
Customer profit is maximized over lifetime

Agile Example 1 (eXtreme Programming, XP)
User Stories are utilized instead of requirements
Stakeholders work together
1-3 weeks between releases
Test-driven (TDD)
Refactoring is used
Paired programming is utilized

Scrum
Iterations of agile are “sprints” and are 1-3 weeks
Potentially shippable products result after sprints are completed

Iterations of sprints build upon the previous, and are based on user stories

Scrum Team:
Product owner, who is manager
Scrum manager, manages sprints/dev team
Dev team, who are the developers
Sprint in Scrum
1 Development increment
Sprint planning meeting begins the sprint
Daily scrum meetings are held
Sprint retrospective looks back at the generated project

Drawbacks of Agile Development

- Emphasis on team communication/cohesion
- Smaller projects are easiest to manage
- Need customer to be present at most times
- Documentation really isn't generated

Topic 2: Managing the Software Process

Part 2) Cost Estimation and Risk Management
Elapsed time:

- time from start - end of a project

Development Effort:

- labour measured in person-months or person-days
- To estimate development effort to money amount, multiply average cost to how many engineers it will take to complete the job

The 7 principles of effective cost-estimation

1) Divide and Conquer
2) Include everything when making estimates
3) Base estimates on past experience + knowledge of project
4) Account for differences when extrapolating costs
5) Anticipate worst case
6) Combine independent estimates
7) Revise/refine estimates as work progresses

Divide and Conquer for cost-estimation
Project is split into subsystems
Divide subsystems into activities needed to develop them
Make detailed estimates for activity
Sum results for estimate

Include everything when making estimates for cost-estimation
Ensure that no cost is hidden or missed

Base estimates on past experience + current knowledge
Expect similar projects to take similar work
Base estimates on personal judgement
Use algorithmic models in software industry

Account for differences when extrapolating costs
Different developers, needs, complexity of requirements

Anticipate worst case when extrapolating costs
This includes planning for contingencies
More important functions should be implemented first
3 Estimates of cases:
Optimistic (O) (Everything is perfect)
Likely (L) (typical things go wrong)
Pessimistic (what can go wrong)

Combine multiple independent estimates for cost-estimation
Several techniques to average the cost
Examine reasons for discrepancies between estimates

Revise and refine estimates as work progresses
Update costs as detail is added
Update costs as requirements are different
Update costs as problems are discovered

Algorithmic models of cost analysis allow one to:
- Estimate user stories, requirements
- Estimate the amount of work necessary

Some examples of algorithmic models include:
i) Constructive Cost Model (COCOMO)
$E = a + bN^c$
ii) Functions Points
$S = W_1F_1 + W_2F_2 + W_3F_3 + \dots$

What is project scheduling and tracking?
Scheduling
- deciding what sequence a set of activities will be performed
- i.e. like a scheduler for a parallel program

Tracking
- how well you're sticking to a cost and schedule

Tasks define:
- what needs to be completed during a project
- example is user requirements, or system testing
- duration, of of a task describes how long it needs to be completed

- i.e. measured in person-days or person-weeks

A deliverable is:
- Something that can be “sent out” to the client or development team
- Describes a program, code or reports

Dependencies are:
- tasks that require another task having been completed first
- i.e. Code can not be tested without first being written

Milestones describe:
- achievements made during the project, i.e. deliverables or completing a task
- deliverable may be a milestone but not all milestones are deliverables

Deadlines refer to:
- times that milestones need to be met
- Can be set by both the user or the client

What is meant by “the deadline has slipped?”
- If a deadline is missed
- Project managers need to monitor whether or not past deadlines have slipped or whether future deadlines are going to slip
- Manage resources to meet deadlines

Some challenges in project management include:

i) Accurately estimate costs

ii) Measure progress and meet deadlines

Strategies to measure progress and meet deadlines include:

a. Improve cost estimation skills
b. Make relationships with members of team
c. Be realistic in requirements, and follow an iterative approach
d. Use earned-value charts to monitor progress

iii) Dealing with lack of human resources and/or technology
- Must consider resources available
- Must consider if you have the skill/talent pool available to complete your project

iv) Communication in a large project
- Need to sync requirements around many people
- Need to have effective meetings
- Make sure that project info is available
- Emphasize collaboration on groupware

v) Hard to obtain commitment and agreement from others
- Need to understand negotiation and leadership
- Need to ensure everyone is represented fairly
- Need to be assertive when there are disagreements

Risks describe:
- Situations where one is exposed to danger
- Risks are problems that are yet to occur

Risk management can be thought of as:
- Identifying, prioritizing and mitigating risks
- Not being negative or worrying

The 5 activities for risk management include:

i) Risk discovery (Identify risks)
ii) Exposure analysis (How probably is the risk to affect you?)
iii) Contingency planning (What happens if risk is materialized?)
iv) Mitigation (How to make contingency possible?)
v) Ongoing monitoring
Acronym: ERCMO

How might one identify a risk?
Brainstorming strategies
What is the worst problem for a strategy?
How to predict future?
Compare failures of other projects?
Consider partial failures?

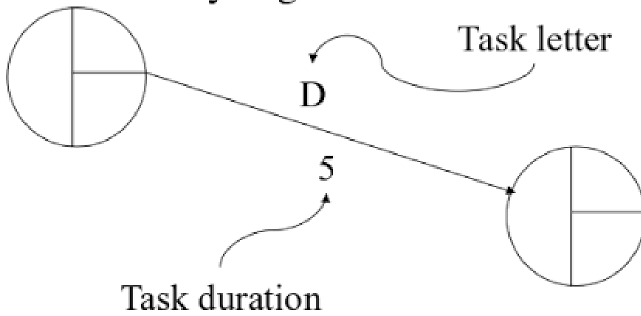
PERT Charts

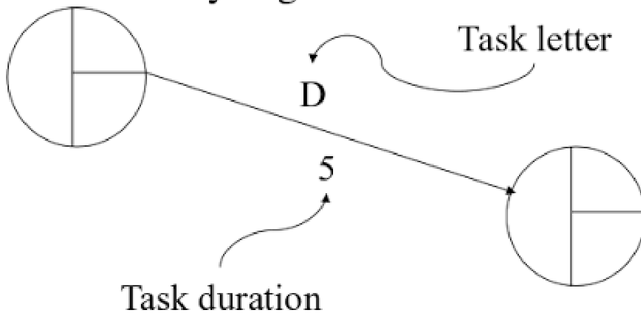
What does PERT stand for?
It is a **Project**, **Evaluation** and **Review Technique**

What is a PERT Chart?
It's a representation to schedule events in a project
It's represented as a DIRECTED **graph**

Nodes/Vertices are **events**
Edges are **tasks** to be completed
NE
ET

A task can only be completed until the Event preceding it has occurred.





PERT Node:

Sequence Number:

- Order in which event must be completed

Earliest completion Time (ECT)

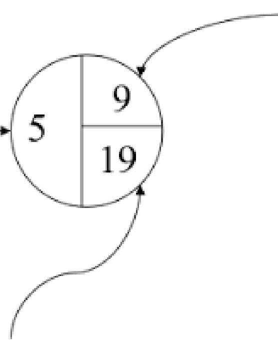
- Earliest time this event can be completed

Latest Completion Time (LCT)

- Latest time this event can be completed

Sequence number assigned

Only task edges indicate dependencies



Earliest Completion Time (ECT):

Earliest time this event can be achieved, given durations and dependencies

Latest Completion Time (LCT):

NOTE: For dependencies in a PERT chart, the DIRECTED EDGES indicate dependencies

The steps in order to build a PERT chart are:

i) Make list of project tasks/events
ii) Find dependencies within tasks
iii) Draw PERT without durations, ECTs or LCTs
iv) Estimate duration of tasks
v) Fill durations

vi) Calculate ECTs and LCTs

Dashed lines represent tasks that are?
Dummy tasks, to show dependency between two events, where no activity occurs
Dashed = Dependency

- The node that these tasks point to is dependent on the nodes preceding it

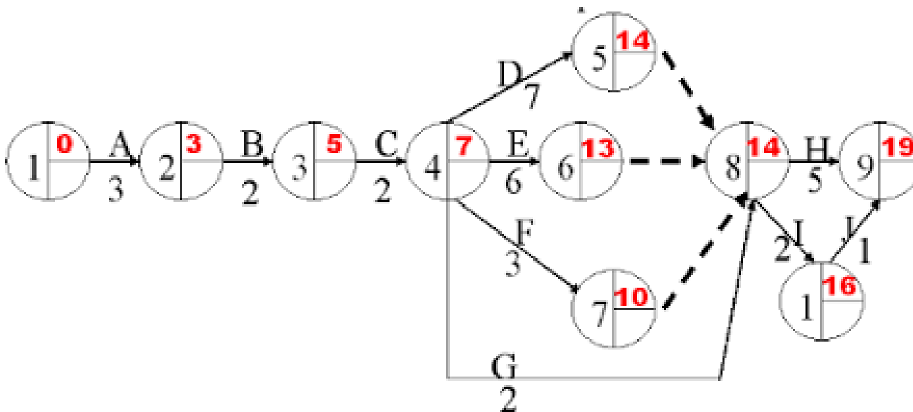
How are durations estimated during tasks?

- Estimating durations from previous tasks
- Identify difficulty and skill level

Factors to estimate durations are:
i) Difficulty of task
ii) Size of team
iii) Experience
iv) Availability of users
v) Management commitment
vi) Other project issues

How might you calculate ECTs?
Find an event that's not dependent on others

NOTE: For dependent tasks, sum all the totals of tasks before it
i.e.



Node 2 has a ECT of 3 because the task before it requires a time duration of 3
Node 3 has a ECT of 5 because the task(s) before it require a time duration of 5

Critical path of a PERT chart determines longest time for project to complete on due date

Some useful advantages of PERT charts are:
i) Determining time estimates
ii) Deriving dates from the chart
iii) Finding out what tasks should need more resources
iv) Identifying risks

- Determining what tasks are the most important to complete

GANTT Charts

What are GANTT Charts?

- Graph representation of a schedule

Y axis of Gantt chart is:

- Tasks to be performed

X axis of Gantt chart is:

- Timeline for tasks to be performed

SHADED areas in Gantt charts are progress
Un-shaded areas in Gantt charts are tasks to be done
Partially shaded areas in Gantt charts are tasks in progress*
(% shaded represents % done)

What are the major differences between PERT and GANTT charts?

Differences	
PERT	GANTT
- Dependencies are indicated by verticies - Critical path is “obvious” - Shows how tasks must be completed in order	- Represent overlapping tasks - See what tasks are falling behind - Show progress of entire project
Similarities	
- Can easily see flow of tasks/dependencies - See stage of project	

NOTE: Setting task length to 0 in MS Project will specify a milestone

Answers to Study Questions(click)

Topic 3: Requirements

What are requirements?

- Something that describes (stakeholders) needs and desires
- Short, info that describes system
- Something agreed upon by stakeholders
- Solves the original problem

Requirements Engineering involves:

i) Elicitation
ii) Analysis/prioritization
iii) Definition
iv) Prototyping
v) Review
vi) Agreement

i) Elicitation

- Elicitation is process of collecting information from a user

Why might domain analysis be important?

- In order to determine the field of business that certain software will be used in
    - Help to develop requirements better

Scope definition serves the purpose of:

- Defining the problem you're trying to solve faster

NOTE: a more precise problem definition narrows the "scope"

ii) Analysis of Requirements:

Need to check accuracy and detail of requirements

ii) Prioritization of Requirements:

Used to maximize value given constraints

- NOTE: The $100 test is assigning 100 to the client, and asking where they would spend the money

iii) Prototyping of Requirements

Generation of stages of a program, and implementing new stages
Refinement of a program is done in "stages" or prototypes

iv) Review of Requirements

Different project management strategies will have different requirements
i.e. Waterfall generates many documents prior to prototyping
i.e. Agile generates minimal documents as you go along

v) Agreement of Requirements

Stakeholders and users should decide on:
Format and information needs of functions
Interface of program functionality
Interaction between system

What are 3 non-functional requirements of a system?

1) Quality
2) Platform
3) Process

1) Quality:

How reliable, maintainable, testable and reusable is the program?
How secure, responsive, throughput and usable is the program?

2) Platform:
What platform will the program be executed on?
What environment will the program run in?

3) Process:
What is the project plan, and development methods?

The complexity of requirement specifications depend on:
i) Size/complexity of project
ii) Future release plans
iii) Customer support/maintenance
iv) How knowledgeable are the developers?

Risks and Difficulties to be expected in requirement definition?
i) No understanding domain
ii) Not evolving with requirements
iii) Attempting too much
iv) Not resolving conflicting requirements
v) Difficult to state requirements with precision

Topic 3: User Stories

Define User Stories:

- Documents that describe functionality that's valuable to users/customers
- A reminder to have a CONVERSATION with the "user" to get details
- Customer requirements

The three components of user stories include:

1) Card
2) Conversation
3) Confirmation

1) Card
A description of the user story

2) Conversation
A discussion about the requirement to get details

3) Confirmation
Test that tells details of the story

Who writes User Stories?
The USER! (And stakeholders)
Ensures that the software accurately represents what the user wants

Are user stories technical or more casual?

Casual, in order for the user to describe stories and prioritize them
Only include details that the user might care about

A physical example of a user story might be:

Front:
i) User Story + Points
Back:
ii) Acceptance tests to determine the user expectation

Points of a User Story describe:

- Size/complexity of a story relative to other stories
- SCALE LINEARLY, i.e. 4 pts takes 2x as long as 2 pts

Who determines the "value" of a story point?

DEVELOPERS

- They can set boundaries of what 1 point is worth (i.e. 1 evening, 1 week etc)

Acceptance Tests help to:

- Determine aspects about a user story
- Verify that the user story is supposed to work how it's supposed to

NOTE: Usually automated

Story-Driven Development is:

- A form of developing where milestones are measured by number of stories completed
- Responsible for producing deliverables at the end of each iteration

How are releases planned in story driven development?

- Highest priority releases are assigned first*
- Therefore, adjustments to development will not affect the overall critical development

User-Stories are easily prioritized. What are they based on?:

- Desirability to users and customers
- How "complementary" the story is to another story
    - i.e. A map software with a "zoom out" story might not be priority #1, but coupled with "zoom in", it becomes very high priority

NOTE: User-Stories are prioritized to MAXIMIZE VALUE to organization

The INVEST principle is used for good user stories. Define INVEST

i) Independent
ii) Negotiable
iii) Value
iv) Estimateable
v) Small
vi) Testable

i) **Independent**
Don't make user stories dependent on each other for flexibility

ii) **Negotiable**
Stories can be changed as they're not contracts or fixed requirements*
- Recall stories exist to have a conversation with stakeholder

iii)

INVEST Acronym Expanded	Example
Independent: Don't make stories dependent on each other.	Pay w/ visa Pay w/ Mastercard Pay w/ Amex 3 days to support Amex, but 1 day to support other two Therefore, prioritize 3 day support, and add other two as time permits (group 3 stories into 1)
Negotiable: Stories are negotiable by the user/stakeholders.	A company can pay for job posting w/ credit card. NOTE: Accept "x" cards. ^this is all the user wants to care about
Value: User stories are VALUABLE to users/customers.	Up to "x" users can use application with certain license. Errors are shown to user and logged consistently. ^ Note how it describes high-level user needs, and not classes/methods of implementation.
Estimateable: Developers can estimate size of story.	Can a story be estimated accurately? If it's too big, it should be decomposed into an **epic.**
Small: Stories should not be too small/large.	
Testable: Acceptance tests must be passed before the story is proved to have passed its tests.	

NOTE: User stories are to REMIND users to discuss things with developers, not have written communication
- User stories are written in language of user
- User stories fall within an acceptable size (not an epic, not too small)
- User stories work between iterations
- User stories defer detail due to discussion at later times

[Answer to study questions (Click)]

2212 Notes
2015, B. Locke

Topic 4: Software Design

Part 4.1)

Design can be used to describe:

- Process to implement **functional requirements**
- Follow constraints from non-functional requirements
- Adhering to quality

Design Space is defined as

- all possible designs and their combinations

Which two levels can design be accomplished?
At both the **software architecture** and **component level**

Contrast the Software level with the Component level

Software	Component
- High level overview - System is divided into subsystems - Subsystem interaction is defined	- Software or hardware with clear roles - Components are replaceable and reusable - Components can be special-purpose

How is modeling useful in software design?

- Aids design and allows for visualization of design
- Help generate documentation for design

UML diagrams are:
Class-based diagrams
Have interactions that indicate how classes talk to each other
Display the internal-connections of systems
Show logical arrangement of system

NOTE: **<u>UNDERLINES</u>** in UML diagrams mea "static" variables

Some Identifications of Object Oriented design include:
i) Identifying classes
ii) Identifying attributes
iii) Identifying inheritance and interfaces
iv) Identifying responsibilities
v) Identifying Operations

i) Identifying Classes:
Don't include redundant classes
Don't represent instances
Don't add vague classes

ii) Identifying Attributes:
Only add essential attributes to classes

iii) Identifying inheritance and interfaces
Bottom up Strategy:
Group similar classes, and give them a superclass (factor out their common attributes)

Top down Strategy:
Keep classes general, factor out those that are un-similar and give them subtypes

iv) Identifying Responsibilities
Ask, what the system needs to do?
What does a specific class do?

- 1 Requirement should be assigned to 1 class (don't give 1 class too many requirements)

v) Identifying Operations
What methods need to be added into our classes?

KNOW THIS TABLE AS RULES TO GENERATE A CLASS DIAGRAM

Attributes	Instance Variables
Generalizations	keyword: **extends**
Interfaces	keyword: **implements**
Associations	Instance Variables
- One way	Declare a reference variable to that class
- Two way	Use two one-way associations to link classes
- Multiplicity	Use other class type (list, vector)

i.e.
In a One to Many relationship (1 -> *)
(This class describes the ONE side)

```
class SpecificFlight
{
	private Calendar date;
	private RegularFlight regularFlight;

		SpecificFlight (Calendar date, RegularFlight, regularFlight)
		{
			this.date = date;
			this.regularFlight = regularFlight;
		}
}
```

In a One to Many relationship (1 -> *)
(This class describes the MANY side)

```
class RegularFlight
{
        private List <SpecificFlight> specificFlights;

        public void addSpecificFlight(Calendar dates)
        {
                SpecificFlight newSpecificFlight = new Specificflight(dates, this);
        }
        specificFlights.add (newSpecificflight);
}
```

NOTE: How there can be many **SpecificFlight** objects within the ReguliarFlight class with the multiplicity association. Many **SpecificFlights** can be stored in a RegularFlight's list, but each **SpecificFlight** belongs to one RegularFlight.

Two basic essentials of Design Principles are:

1) Low coupling
2) High cohesion

Define Coupling:

It is a measure of how **independent** 2 things are.

High/Strong coupling is:

- Sharing global variables
- 1 Object uses data of another object

Low/Weak coupling is:

- 1 Object requests data of another object through a method
- Data is shared through params and returns
- A measure of good coding practice because it practices encapsulating

Define Cohesion:

It's a measure of how data and functionality is related

High Cohesion is:

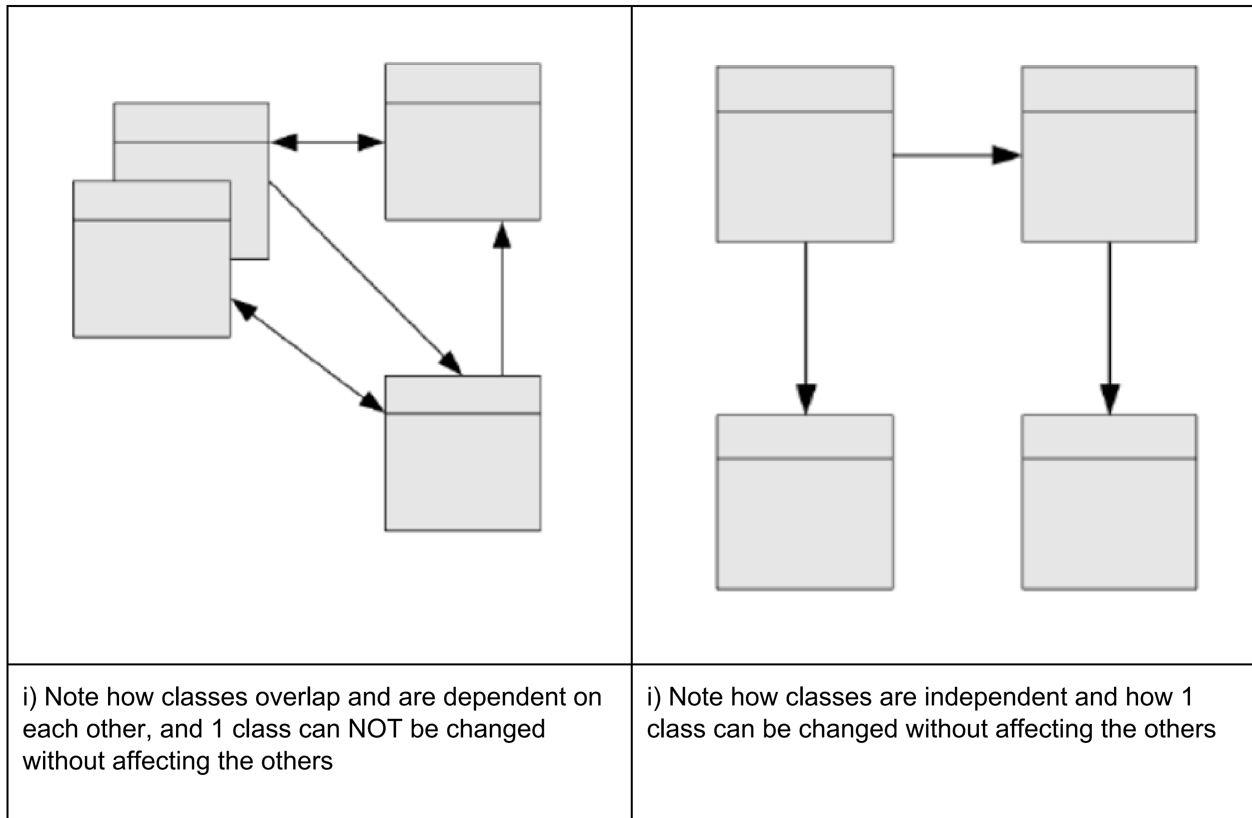
- Every class has a defined function that can compliment another class
- Cohesive classes work on the SAME data

Low Cohesion is:

- Grouping functions that don't work on the same data
- An example of a "god class"

i.e.

High Coupling & Low Cohesion	Low Coupling & High Cohesion



i) Note how classes overlap and are dependent on each other, and 1 class can NOT be changed without affecting the others	i) Note how classes are independent and how 1 class can be changed without affecting the others

Topic 4: Software Design

Part 4.2)

Basic OO Design Principles included:
i) Identifying Classes
ii) Identifying Attributes
iii) Identifying inheritance and interfaces
iv) Identifying Responsibilities
v) Identifying Operations

Advanced OO Design Principles are defined by:
vi) Encapsulating what Varies
vii) Code to interfaces
viii) Composition > Inheritance

vi) Encapsulating what Varies:

Encapsulation allow certain parts of a program to be altered/extended without affecting other classes

RECALL BETH'S DUCK HUNT:

Adding new functionalities (i.e. flying behaviours) is done by extending, and this in no way affects the original Duck

Duck Superclass

- MallardDuck (extends Duck, implements FlyBehaviour)
- RedheadDuck (extends Duck, implements FlyBehaviour)

Naive Solution	Improved Solution

MallardDuck
{
fly();
}
 	
RedheadDuck
{
fly();
}
 	
Rubberduck
{
fly();
}
	MallardDuck
	{
	FlyBehaviour newFly;
	}
	
	RedheadDuck
	{
	FlyBehaviour newFly;
	}
	
	Rubberduck
	{
	}

As noted, FlyBehaviour is factored out into a class that defines what flying actions are permitted by a certain duck. Rubberduck does NOT fly, therefore it does NOT need to declare an object of FlyBehaviour. Code duplication is removed, as each duck that has a FlyBehaviour object shares the same methods.	

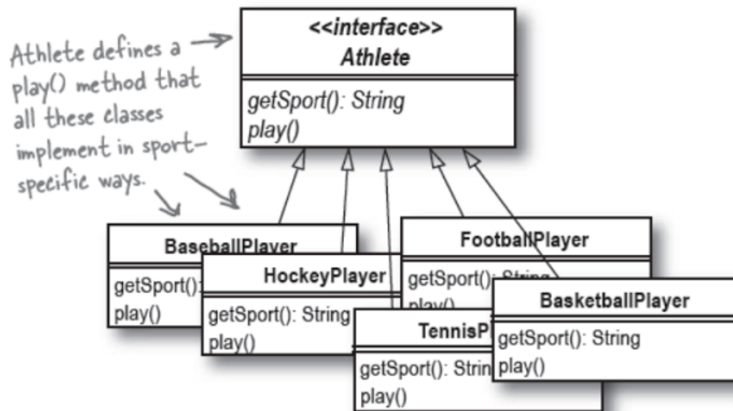
vii) Code to interfaces

Should subclasses interact or superclasses interact?

Superclasses, because then each sub-class of that superclass can be implemented and the interactions will still apply.

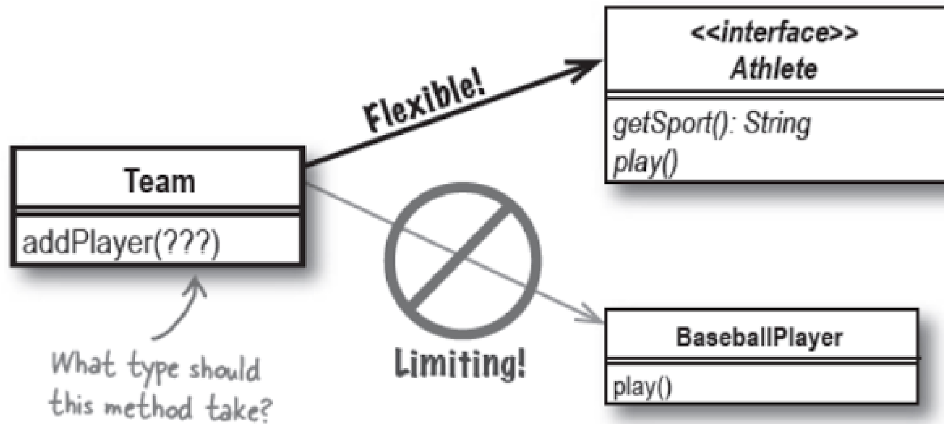
i.e.

If you had a Superclass Athlete that is extended by Sports Classes



If you wanted to add a AddPlayer method, where would you want to add it? Would you add it to each subclass or the superclass Athlete?

Obviously adding it to the Athlete superclass is best, as each subclass can still extend it and modify the method as necessary.



Each addPlayer method can just add a player of the type Athlete, and since each subclass extends Athlete, those subclasses may be added.

If the addPlayer method was added to the subclasses (BaseballPlayer, HockeyPlayer, BasketballPlayer etc...) there would be a lot of duplicate code and duplicate addPlayer methods. Each additional Player class that implemented Athlete would need a new addPlayer method.

viii) Composition > Inheritance
Inheritance refers to: **IS-A**
i.e. MallardDuck **IS-A** Duck
i.e. HockeyPlayer **IS-A** Athlete

Composition refers to: **OWNS-A**
Pirate **OWNS-A** Sword

i.e.
Unit* Pirate = new Unit();
pirate->setProperty("weapon", new Sword());
If you delete the Pirate unit, then the property of a sword is deleted as well
Composition is a strong linkage, 1 object needs the other to exist
In this case, object pirate owns-a object sword

Aggregation is a **HAS-A**
OWNS-A means that both objects may exist independently of each other

i.e.
Student jay = newStudent();
Class calculus = newClass();
calculus.addStudent (jay);
If you delete the class calculus, the student jay still exists outside of the class.
In this case, object calculus HAS-A student object Jay

NOTE: Composition means 2 objects linked, they can't exist independently

2212 Notes
2015, B. Locke

NOTE: Aggregation means 2 objects are linked, but CAN exist independently

Design Pattern Example: Model-View-Controller

Design Pattern's Components

i) Abstract	(language-independent)
ii) Context	(code design is common)
iii) Problem	(examine the goals)
iv) Solution	(how to achieve these goals)

MVC Abstract: No language specified
MVC Context: encapsulate interactive components and have a design pattern
MVC Problem: achieve user interaction software
MVC Solution: (listed below)

Model:
- Capture data
- Control conversion between persistent and dynamic data
- Compute data

View:
- Display elements for user
- GUI

Controller:
- Coordinates Model and View
- Application "flow" is controlled by controller

A singleton is an example of a design pattern.
- 1 class has 1 instance that is GLOBALLY ACCESSIBLE

To implement a singleton, what is necessary?
- Public static accessor (getter function)
- Protected/private constructors
- Private static attribute (reference instance)

Drawbacks of utilizing a singleton are:
- Bad habit, as it is a global variable with global access
- Difficult to delete as everything has reference to it

An alternative to singleton usage is: reference passing.
- Therefore, global access is still present, but values/variables are only returned when requested- not publically.

Topic 4: UML Class Diagrams

UML: Unified Modelling Language

- Diagrams to represent object oriented software

What is displayed on UML diagrams?

- Classes
  - Attributes within classes
  - Operations/methods within classes
- Associations between classes
- Generalizations (inheritance between classes)

UML Class:

A box, with name of class, attributes and methods
+/- indicate public/private respectively
<u>underlined</u> defines static variables

ClassName
- attribute: type + attribute: type
- method: (params): returnType + method: (params): void

UML Associations:

(X)*---1(Y) represents MANY to ONE
It means there can be MANY “X” to ONE “Y”
i.e. many employees X for one company Y

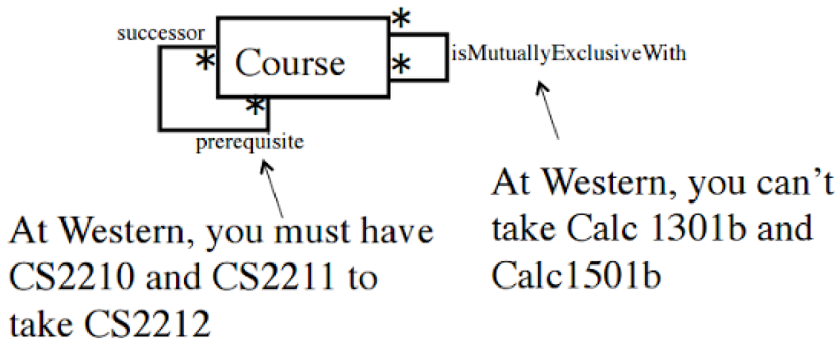
(X)*---*(Y) represents MANY to MANY
It means there can be MANY ”X” to MANY ”Y”
i.e. many users X for many computers Y

(X)1---1(Y) represents ONE to ONE
It means there can be ONE “X” to ONE “Y”
i.e. One User X to One account Y only

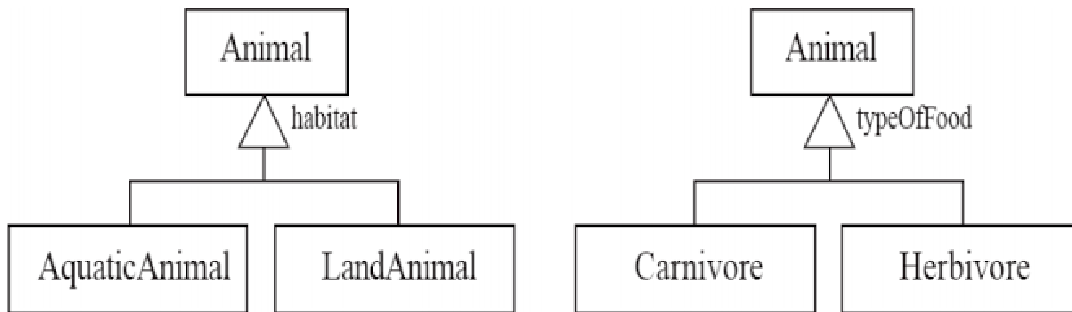
(X)0..1---1(Y) represents ONE or NONE to ONE
It means there may or may not be ONE “X” to ONE “Y”
i.e. A person Y may have a pet “X” or not have a pet “X”

Reflexive Associations within UML Diagrams refer to:

- Associations connecting to itself



Generalizations or Inheritance:
NOTE: Indicate superclasses with an arrow

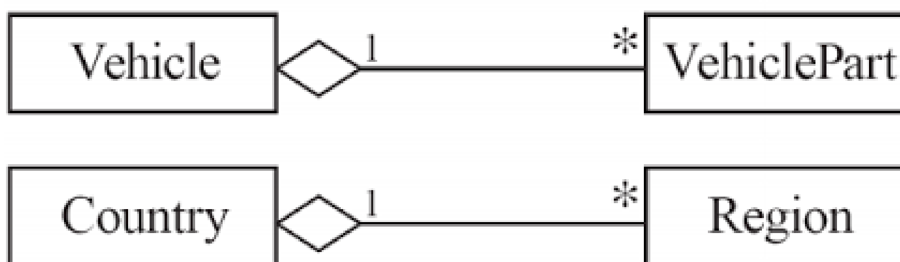


What is the difference between associations and generalizations?
Associations (lines with or without numbers indicating multiplicity) describe how 1 class is called by another at RUNTIME

Genralizations (inheritance) describe how classes belong to a superclas

Aggregation:
RECALL is a HAS-A relationship (isPartOf)
- Indicated by a clear diamond linking the left hand side of the X HAS-A Y component

i.e.



NOTE: Only use aggregations if you can state:
- x is "part of" y
- y is "composed of" x

Composition:
RECALL is a OWNS-A relationship
- Indicated by a BLACK diamond
- If aggregate is destroyed, everything is destroyed (parts are NOT independent)

You can't have a room without a building
You can't have a building without a room

Answer to study questions (Click)

Topic 5: User Interface Design

5.1)
User Interface Design is:
- GUI!

The main focus of User Interface (GUI) Design is?
- Utility (can the system accomplish what it's supposed to
- Usability (how easily are things accomplished?)

Usability Principles:
1. Do not rely on usability guidelines
2. Base UI design on the users' tasks
3. Tasks should be made achievable through simplicity
4. Interface should be informative
5. Offer error prevention/handling hints
6. Allow operations to be undone
7. Allow for adequate response time
8. Simplify display
9. Provide necessary help/support
10. Allow for consistency

1. Do not rely on usability guidelines
- Test with users, not just guidelines

2. Base UI design on users' tasks
- Consider how users will be using the GUI (are they n00bs or pros?)

3. Tasks should be made achievable through simplicity

- Group like tasks together to make usage intuitive
- Make shortcuts available to make usage ergonomic

4. Interface should be informative
- Define what certain commands do
- Make tasks redundant
- Tell users where they are in the program

5. Provide error prevention and handling hints
- Prompt user for correct format of input
- Tell user when errors occur

6. Permit Reversal of actions
- Let user navigate to area prior to problem

7. Ensure that response time is adequate
- Don’t make user wait for error response
- Response time should be <= 1 second

8. Simplify information presented
- Do not display too much info
- Organize info with appropriate fonts/images

9. Provide all necessary help
- Allow for support to be easily found

10. Be consistent
- Do not change layout too much
- Keep program navigation familiar

A heuristics-based approach to evaluating user interfaces is:
heuristic: Defined as using mental shortcuts for user interfaces
Tasks/Windows/Dialogues should have notes of defects and violations (along with proposed solutions)

How might focus groups evaluate User Interfaces?
- Pick group that works on the software the most
- Note difficulties experienced by users who want to use the program
- Implement changes

Human-Computer Interaction is:
The study of how people interact w/ computers
Combines cognition, graphic design, interface technology and computer graphics

Conceptual Modelling has 4 components:
i) Metaphors/Analogies
ii) User-level concepts
iii) Relationships between concepts
iv) Mapping of concepts to task-domains

Topic 5: User Interface Design

5.2)

WIMP definition of GUI:

i) Windows (run a self-contained program)
ii) Icons (symbols denote actions)
iii) Menus (Allow multiple options for selection)
iv) Pointers (Can choose what you want to see)

Post-WIMP:

- Touchscreens
- Button Based GUIs
- Gesture-based GUIs

Important terminology for GUIs:

Widget:

- Windows Gadget
- Describes components of a UI

Dialog:

- File that can be interacted with
- Not the main UI (i.e. a popup)

Focus:

- The current window that the user can add input to
- Click-To-Focus is most common OS model
- Follows a pointer or mouse

Focus Stealing

- refers to dialogs that redirect input to itself
- Popups

Mode:

- Programs can be split into modes
- Input is interpreted differently in differnt modes

Mode Errors:

- Refer to how Interactions defined for another mode are used incorrectly

Look and Feel:

- Look is the appearance and design
- Feel is how components interact/behave

Native look and feel vs pluggable look and feel

- Native is the OS default
- Pluggable is how the look can be swapped out

Answers to Study Questions (click)

Topic 6: Software Testing

6.1)

Purpose of Testing:

- See how system works, and catch bugs

Test Case:

- Input sets used to test program
- JUnit is used frequently to set up test cases
- A set of test cases is called a **Test Suite**

Good Testing involves:

- Conducting tests that may find bugs
- Test requirements thoroughly

Bug:

- A “problem” or unexpected error in the code

Failure:

- Program does something wrong

Fault:

- Incorrect code causing a failure

Error:

- Mistake that programmer made to cause a bug

What is the Pareto Principle and what does it describe?

80-20

- 80% of failures caused by 20% of faults
- 20% of failures caused by 80% of faults

Why is testing more important the larger programs go?

Difficult to catch bugs in huge files

Contrast Testing with Debugging

Testing = running test cases to find failures

Debugging = finding faults and correcting them

Unit Testing refers to what methodology of testing?

- Test individual methods/classes isolated from one another
- I.e. test classes independently

Integration testing refers to what style of testing?

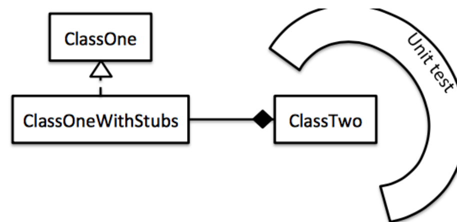
- Test compilation of interacting classes and making sure they operate as expected

Stub:

- Function/method not integrated yet

- Simple known values to test with
- "stubbed out" methods are invoked to run testing
- Stub method is usually used if a method we need to test
    - Uses other methods/classes that are not completed (stub out the unfinished methods/classes)
    - Uses methods/classes that work with external sources (files, database, network)
        - Unit tests need to be fast
        - unit tests need to be isolated (code + database = integration test)
        - Stub out these methods/classes
    - Uses Methods that return different values based on date/time
    - Uses stochastic (non-deterministic) methods

Stubbing Methods:
- Can be done manually
    - ex) can create a subclass of classOne with some methods implemented /overriden as stubs, and use this for tests



Stubbing Stochastic Methods:
- Stubs are easy to create for deterministic methods
- Non-Deterministic methods that exhibit unpredicatiable behavior?
    - always will return the same value
    - always return the same sequence of values
        - Use a static variable to track how many times the method has been called
        - On the first invocation, we'll return 5
        - On the second, return 99
        - etc.

Fake Objects:
- Object that has all methods implemented as stubs
- Object that takes some kind of shortcut making it unsuitable for the final product

Mock Objects:
- Objects mimic the behavior of real objects
- Used in behaviour verification
- can simulate the behaviour of complex objects
- implemented using a mocking library (ex) in java, EasyMock, JMock, JMockIt, Mockito)
- mocking libraries available in most languages

Unit vs Integration Tests:
- With only integration tests, we can't definitively say that:
    - the problem is in the code

    - the problem is in the database
    - We waste time finding the bug
- Unit + integration tests means:
    - The code works
    - code works with database
    - code works with your code

Mocks:

- Can often be confused with stubs
    - Mocks do allow us to stub methods
    - Also allow us to :
        - Verify that specific methods were called
        - verify that specific arguments were passed
        - thus, we can record and verify the interactions between the classes and its collaborators

Continuous Integration:

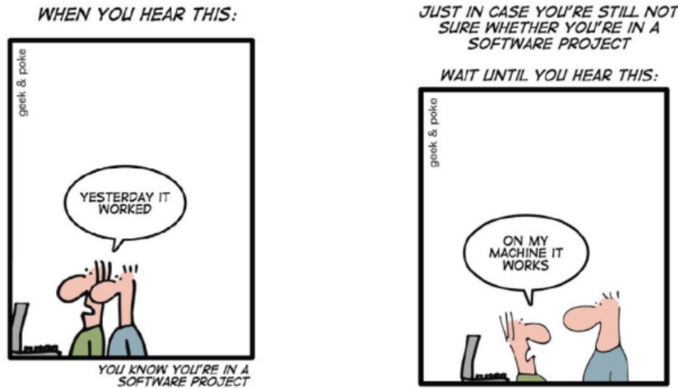
- Common Scenario:
    - Jerry and Long are working on a project
        - they each implement a few classes
        - code them
        - ensure they are well tested
    - When they're each done, they integrate them
        - Everything Breaks (:O)

Integration Hell:

- Extremely risky for a project
- Difficult to determine time required to resolve the integration problems
    - May (vastly) exceed our budget
    - May (vastly) exceed our schedule

Integration Hell

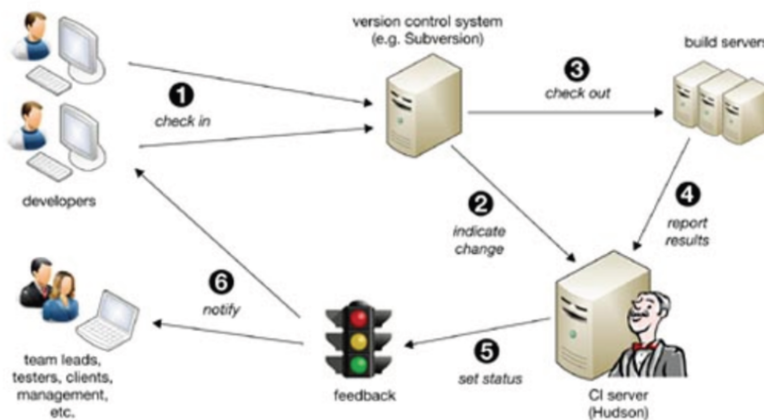
That awkward moment near the end of a project when everyone realizes that none of their classes interoperate correctly



Continuous Integration:
- Originally from the Extreme Programming (XP) development methodology
- Mitigates risks associated iwth integrating software
- Avoids integration hell
- -Integrate early and integrate often (on every change)

Continuous Integration Server:
- Automates building, testing and reoprting
- ex) Hudson, CruiseControl,TeamCity,Jenkins,etc
- Developers might forget to run the tests
- It might take too long to run the tests
- We might need to test the code in various environments
- Reports sent back provide useful insights to the team
- Can run all sorts of utilities on our code
- Can run higher level tests if they are automated

Continuous Integration Server



System Testing:

- Testing the entire system
    - End to end
        - Tests workflows or apths
            - Happy paths and unhappy paths
    - Acceptance
        - Tests done by the client in “accepting” that the requirements ofteh ocntract ar emet (so they pay you)
        - Suite of tests defining when a requirement or user store is “done”

Automating Acceptance Tests:

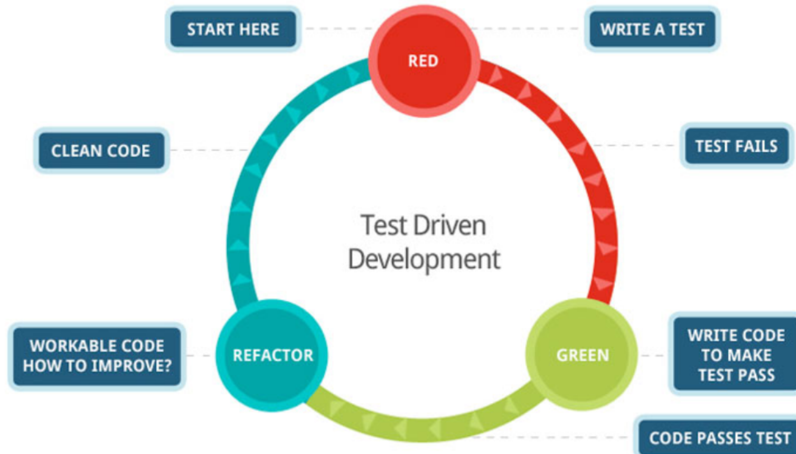
- Benefit of system implemented as MVC
    - Substitute a test driver class for the GUI view
    - Interact with the controller and model as the GUI would

Test Driven Development:

- Tests are written BEFORE code
- Three Laws:
    - You are not allowed to write any production code until you have fisrt written a fialing test
    - you are not allowed to write more of a unit testt han is sufficient to fial - not compling is failing
    - you are not allowed to write more production code than is sufficient to pass the curerntl yfailing test

- Test and code grow together
- code is always verified
    - can see if fadding code has broken things that were working (regression testing)
- Refactoring happens often
    - clean up any messy implementation
    - can immediately test
- Built -in documentation

tests help define expectations of units/ classes



Topic 6: Software Testing

6.2)

Structural Testing (Path Analysis):

- A white-box or glass-box testing technique
    - we are looking at the code

- Is the program "hitting" every path of execution

- Levels of code coverage
  - C_0 → Statement/Line coverage
  - C_1 → Branch coverage
  - C_2 → Condition coverage
  - C_3 → Multiple condition coverage
  - C_4 → Path coverage

Statement Coverage:

```
public static String classify(int x) {
   boolean pos = true;
   boolean even = true;
   if (x > 0)
      pos = true;
   if (x % 2 == 0)
      even = true;
   return String.format("Number:%1$d,
            Positive:%2$b,Even:%3$b", x, pos, even);
}
```

String returned should contain:

- "Positive:true" if the number is positive, ":false" otherwise
- "Even:true" if the number is even, ":false" otherwise

Statement Coverage

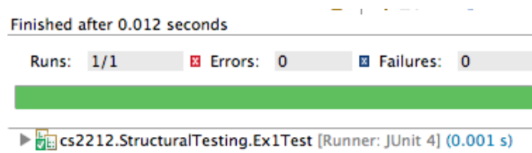
```
@Test
public void testClassify() {
   String result = Ex1.classify(2);
   assertThat(result,
               containsString("Positive:true"));
   assertThat(result, containsString("Even:true"));
}
```

Choose test case for value 2

Will this pass?

Statement Coverage

- Yes! It will pass



- We have full coverage of the method
  - (EclEmma plugin for Eclipse)

Element	Coverage
StructuralTesting	91.9 %
src/main/java	91.9 %
cs2212.StructuralTesting	91.9 %
Ex1.java	91.9 %
Ex1	91.9 %
classify(int)	100.0 %

Statement Coverage

$$C_0 = \frac{|\text{Statements exercised}|}{|\text{Statements}|}$$

Strategy: find paths that cover all statements; write test cases to exercise those paths

- Test suite should have $C_0 = 1$ (ie. 100%)
- Least restrictive of the coverage criteria
  - Some branches may be missed
- Helps measure correctness of code written
  - Better than nothing

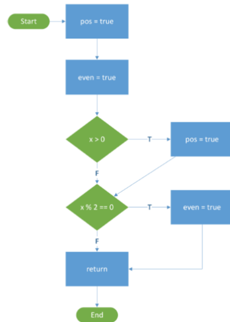
Structural Testing:

- Structural testing is not functional testing
- the method is not correct
  - still need to test correct functionality

Program Flowchats:

- Labelled, directed graph, where
  - Statements are represented by rectangle nodes
  - **Decisions** are **represented** by **diamond** nodes

Program Flowcharts



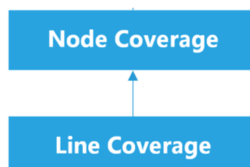
- Nodes
  - statements
- Edges
  - indicate parts of paths which may be followed through the code
- Labels on edges
  - indicate flow direction when a condition is true(T) vs false(F)

Structural Testing:

- If our test suite executes all statements, has visited every node in the flowchart
  - test suite achieves full/100% node coverage or covered every node

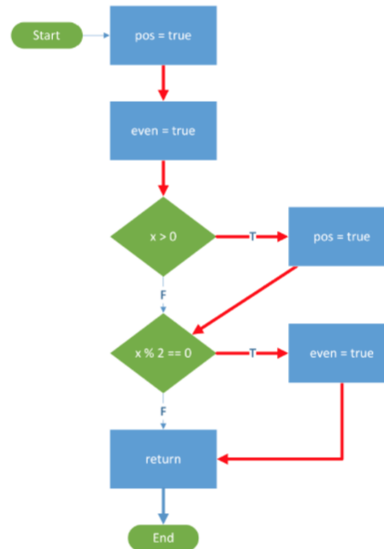
- Note : Statement != Line coverage
  - Any line containing code is measured
    - considered covered if any code no the line is executed
  - Not exactly the same thing as statement coverage
    - can have more than one statement per line
  - **Most people mean *statement coverage* when they say *line coverage***

- 100% node coverage implies 100% line coverage
  - The converse is not true (100% line coverage *DOES NOT* imply 100% node coverage)
  - Node(statement) coverage is stronger than line coverage



Example

- Consider the first example flowchart
  - Test case: x = 2
  - All nodes visited
    - 100% statement coverage
  - However, neither F branch has ever been taken



Example

Finished after 0.031 seconds

Runs: 1/1 Errors: 0 Failures: 0

cs2212.StructuralTesting.Ex1Test [Runner: JUnit 4] (0.002 s)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
StructuralTesting	20.0 %	2	8	10
src/main/java	20.0 %	2	8	10
cs2212.StructuralTesting	20.0 %	2	8	10
Ex2.java	0.0 %	0	6	6
Ex1.java	50.0 %	2	2	4
Ex1	50.0 %	2	2	4
classify(int)	50.0 %	2	2	4

Edge Coverage:

- if our test suite follows every edge in the flowchart:
  - we say that the test suite covers every edge
  - we can also say that it covers every decision
- Branch Coverage:

- has every branch of each control structure (if, switch) been executed?
- Equivalently, has every edge in the program flowchart been executed?
    - Caution: EclEmma branch coverage only considers edges from decision nodes - not every edge in the flow chart

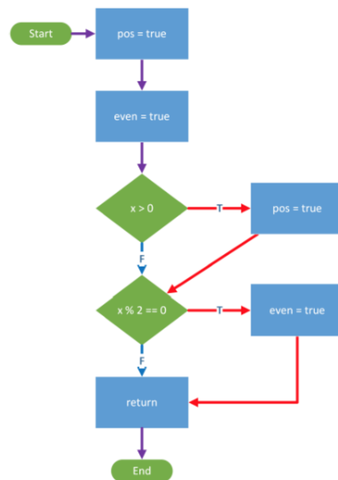
- More thorough than statement coverage
    - catches more problems
    - ex) we had 100% statement coverage for the first flowchart, but the bug was not detected
    - Would we have detected the bug if we followed every edge? see below tests

Edge Coverage

```
@Test
public void testClassifyPositiveEven() {
    String result = Ex1.classify(2);
    assertThat(result,
                containsString("Positive:true"));
    assertThat(result,
                containsString("Even:true"));
}

@Test
public void testClassifyNegativeOdd() {
    String result = Ex1.classify(-1);
    assertThat(result,
                containsString("Positive:false"));
    assertThat(result,
                containsString("Even:false"));
}
```

Edge Coverage



- We'll test with the following test cases:
    - x = 2
    - x = -1
    - A purple edge indicates an edge followed by both test cases
- This test suite will give us 100% edge coverage.

Edge Coverage

Finished after 0.045 seconds

Runs: 2/2 Errors: 0 Failures: 1

cs2212.StructuralTesting.Ex1EdgeTest [Runner: JUnit 4] (0.015 s)
testClassifyPositiveEven (0.001 s)
testClassifyNegativeOdd (0.014 s)

```
java.lang.AssertionError:
Expected: a string containing "Positive:false"
     but: was "Number:-1, Positive:true,Even:true"
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
at cs2212.StructuralTesting.Ex1EdgeTest.testClassifyNegativeOdd(Ex1EdgeTest.java:17)
```

In striving for 100% edge coverage, we caught a bug that our previous test – despite giving us 100% node coverage – could not.

Edge coverage can reveal failures where node coverage cannot

Coverage Terminology:

- If a test suite executes 100% of all statements , we say it achieves 100% node coverage (or statement coverage)
- If a test suit follows 90% of all edges, we say it achieves 90% edge coverage

Edge Coverage:

- 100% edge coverage implies 100% node coverage
    - If we followed every edge, we must have visited every node
    - the converse is not true (100% node coverage does not imply 100% edge coverage)
    - **Edge coverage is stronger than node coverage**

Minimal Test Suites:

- More tests != higher coverage
    - larger test suites take longer to run
    - often want to find some minimal test suite
        - Which still achieves 100% coverage

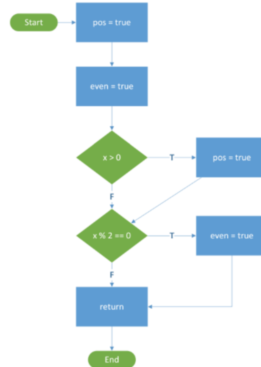
- important to define what minimal means
    - minimal test cases? minimal number of inputs used? minimal time to run?

Minimal Test Suites Example

Suppose we want 100% edge coverage

- Test suite: x = 2, x = 1, x = -2
  - Requires 3 test cases
- Test suite: x = 2, x = -1
  - Requires only 2 test cases

- Not possible to have smaller test suite, with only 1 test case
  - Must cover both decisions T and F edges
- x = 2, x = -1 is a minimal test suite for 100% edge coverage
  - in the sense of needing the fewest test cases



Minimal Test Suites Example

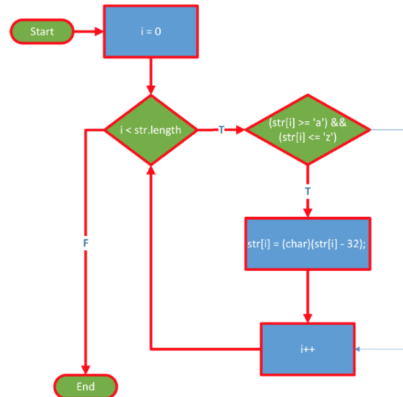
100% node coverage:

1) {'a'}

- but F branch of the inner condition is not taken

100% edge coverage:

1) {'a'} {'X'}
2) {'a', 'X'}



Minimal Test Suites:

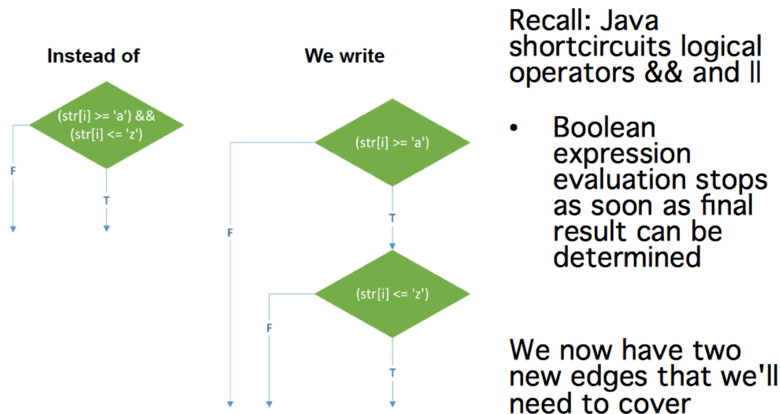
- Which of 1 and 2 is minimal for edge coverage?
- Depends on how we define minimality
  - 1 has 2 tests, but each sha just 1 character
  - 2 has just 1 test, but it has 2 characters

- Minimality for a test suite should always be defined
- For this problem, we could say that:
  - Test suite X is more minimal than test suite Y if either :
    - X has fewer test cases than Y; or
    - X has the same number of test cases as Y, but the total number of characters in X is less than in Y
      - saves us from accepting {'a','a','a','X','X','X'} as minimal

Condition Coverage:

- Decision: everything in parentheses after the if, while
  - e.g. `((str[i] >= 'a') && (str[i] <= 'z'))`
- Condition: the individual terms of the decision
  - e.g `(str[i] >= 'a'), (str[i] <= 'z')`
    - are the two conditions
- So far, we've written each decision in a single diamond.
- If we divide the *conditions* within each decision into separate diamonds, we can get a better reflection of what the program does.

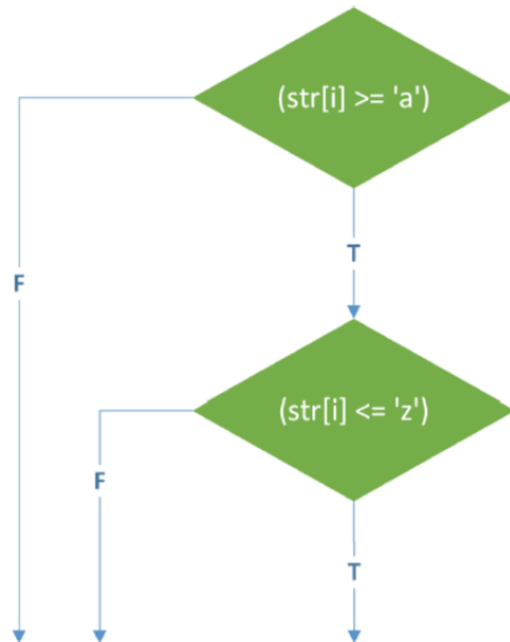
Flowcharts: Splitting up Decisions



Condition Coverage

We split up all diamonds and *then* cover all edges

How many characters will be needed in our test input to achieve 100% condition coverage?

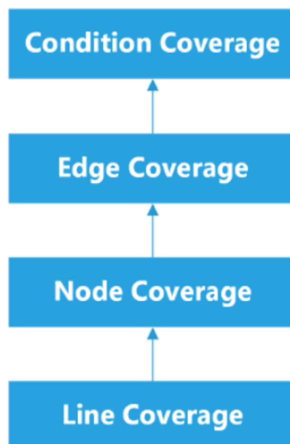


(Refer to slides for more test cases)

Condition Coverage:

- If we have 100% condition coverage,
    - we must have evaluated each condition of an if, while, etc. both ways
    - Therefore, we must have evaluated each decision both way

      Thus, condition average is stronger than edge(decision) coverage

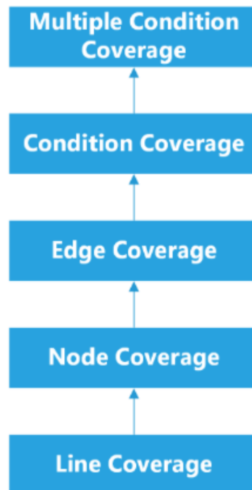


(For more examples refer to slides thanks….)

Multiple Condition Coverage:

- If we have achieved multiple condition coverage
    - We must have evaluated every possible combination of each condition at least once to true and once to false
    - Therefore, we must have evaluated each condition both ways

  Multiple condition coverage is stronger than condition coverage



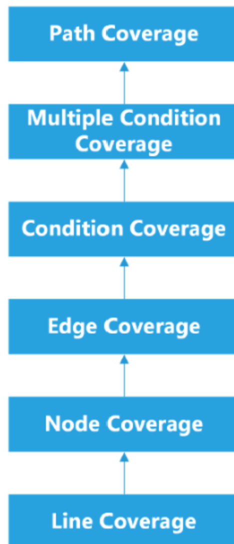
Path Coverage:

- Path: sequence of nodes visited during an execution
- Path coverage is the strongest possible coverage measure
    - 100% path coverage means that every possible path through the program flowchart has been followed
    - for code without loops, this is achievable
    - for code with non-deterministic loops:
        - Each iteration of the loop adds an additional path
        - for some code, we can iterate any number of times
        - ex) the number of iterations might depend on the size of an input

For some code, 100% path coverage is not possible
(examples again on slides)

- If we have achieved 100% path coverage
    - We mtus have evaluated every possible path through the program
    - therefore, we must have evaluated every possible combination of each condition at least once to true and once to false
    -

Path coverage is stronger than multiple condition coverage



Testing Loops:

- **Path coverage for non- deterministic loops is impossible**
- **for entire programs, we do not generally attempt 100% path coverage**
- **however path coverage is still useful for small sections of code**
- **We can't test paths in program with nondeterministic loops**
  - **but we want to do better than multiple condition coverage**
  - **or might smis certain kinds of errors**

Testing Loops

- Programmer may not consider what would happen if:
  - The loop decision is false right from the start
  - The loop decision is true once, and then false
- Sometimes, there is a maximum number of possible iterations for a loop (e.g. the loop might stop at the end of an array).
- Programmer may not consider what would happen if
  - Loop decision is true max times
  - Loop decision is true max-1 times

Testing Loops

It is therefore useful to write test cases which execute the loop:

- 0 times
- 1 time
- More than once
- max times (if applicable)
- max-1 times (if applicable)

Topic 6: Managing the Software Process

Part 3)

Functional Testing:

- Structural testing- white box
- can only test code that exists
- functional testing - black box
- Must consider unstated requirements
- ideally we should go through all requirements systematically and develop tests for them
    - standard methods of doing so exist

(See examples on slides)

Functional Vs Structural Testing:

- **Functional (black-box) testing:**
- Advantages
    - ensures program meets requirements
    - test boundaries ,etc) explicitly
- Disadvantages
    - cannot test undocumented features
    - may not test hidden implementation details thoroughly
- **Structural(white-box) testing:**
- Advantages
    - Tests all code and implementation details
    - coverage metrics can give us an idea of the extent to which our code is tested
- Disadvantage
    - Cannot Test whether all desired features are implemented
    - metrics are not a panacea
    - we saw 100% statement coverage essentially means nothing in relation to code equality/ correctness

2212 Notes
2015, B. Locke

Answer to study questions (click)

Practice Exams (click)