

# **CSE 470 : Software Engineering**



Afrina Khatun  
Lecturer  
CSE, BRACU

# Marks Distribution

- Attendance : 5
- Quiz: 10
- Mid Term Exam : 25
- Project and Assignments : 20
- Final Exam : 40

# Reference Books

2

1. Ian Sommerville, "Software Engineering", Addison Wesley, 8th edition, 2007.
2. Roger S Pressman, Roger Pressman,"Software Engineering: A Practitioner's Approach", McGraw-Hill, 7th edition, 2010.
3. \\tsr\\Fall-2019\\CSE\\AKH\\CSE470\\

# Ground Rules

- ❑ Makeup **MID** and **FINAL** will be conducted as BRACU policy.
- ❑ **No makeup quiz.**
- ❑ **Best (n-1) quiz** will be considered.
- ❑ **Assignment/ Report Submission** needs to be submitted by given **deadline**.
- ❑ Disciplinary action will be taken in case of **Cheating**.
- ❑ You need to maintain **class behavior**.
- ❑ You are **encouraged** to ask questions in class.
- ❑ You can **reach me** by phone, email etc. in case of important (**valid and logical**) query.
- ❑ Visit me in my **consultation hour** (available in tsr) for any query, discussion or even for gossiping.

# Goals of Software Engineering

- To produce software that is **absolutely correct**.
  - To produce software with **minimum effort**.
  - To produce software at the **lowest possible cost**.
  - To produce software in the **least possible time**.
  - To produce software that is **easily maintained** and **modified**.
  - To maximize the **profitability** of the software production effort.
- 
- In practice, none of these ideal goals is completely achievable. The challenge of software engineering is to see how close we can get to achieving these goals.
  - The art of software engineering is balancing these goals for a particular project.

# Real-life Software

Inevitably most software systems are **LARGE** Systems.  
This means:

- ☒ Many people involved in design, building and testing,
- ☒ Team effort, not individual effort
- ☒ Many millions of \$ spent on design and implementation
- ☒ Millions of lines of source code
- ☒ Lifetime measured in years or decades
- ☒ Continuing modification and maintenance

# Example — Eclipse

Eclipse is a popular software development environment for Java.

Some interesting characteristics of a recent release:

- ❑ Lines of source code > 1,350,000.
- ❑ Effort(person-years) > 400.
- ❑ Classes 17,456.
- ❑ Inheritance relations 15,187.
- ❑ Methods 124,359.
- ❑ Object instantiations 43,923.
- ❑ Fields 48,441.
- ❑ Call relations 1,066,838.
- ❑ Lifetime bugs > 40,000.
- ❑ Est. Development cost > \$ 54,000,000.

# Why Is Software Engineering Important

Cost of getting software wrong is often terrible.

- ☒ **Bankruptcy** of software producer.
- ☒ Injury or loss of human life **broken software can KILL people.**
- ☒ Software producer profitability depends on producing software efficiently and minimizing maintenance effort.  
**Software reuse is an economic necessity.**
- ☒ Immense body of old software (legacy code or dusty decks) that must be rebuilt or redesigned to be usable on modern computer systems.
- ☒ **Very, very few contemporary systems work correctly when first installed.** We need to do much better.
- ☒ **Over \$600,000,000,000 spent each year on producing software.**

# Software Horror Stories

- ? Bank of America spent \$23,000,000 on a 5-year project to develop a new accounting system. Spent over \$60,000,000 trying to make new system work, finally abandoned it. Loss of business estimated in excess of \$1,000,000,000.
- ? The B1 bomber required an additional \$1,000,000,000 to improve its air defense software, but the software still isn't working to specification.
- ? Ariane 5, flight 501.  
The loss of a \$500,000,000 spacecraft was ultimately attributed to errors in requirements, specifications and inadequate software reuse practices.



How the customer explained it



How the Project Leader understood it



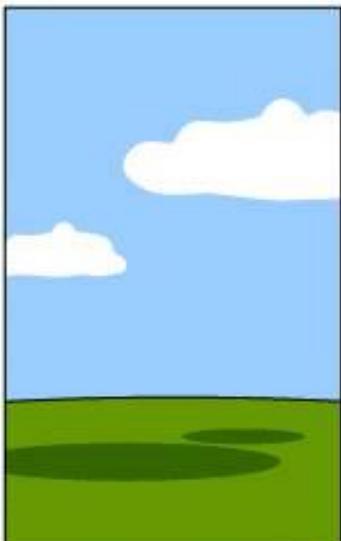
How the Analyst designed it



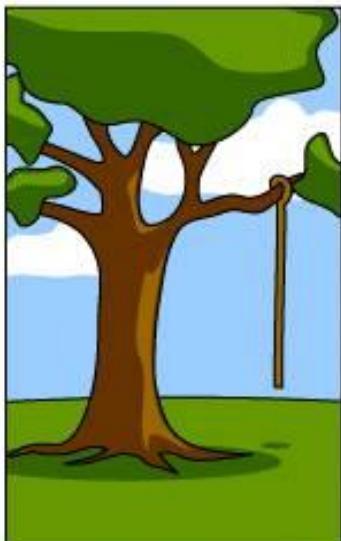
How the Programmer wrote it



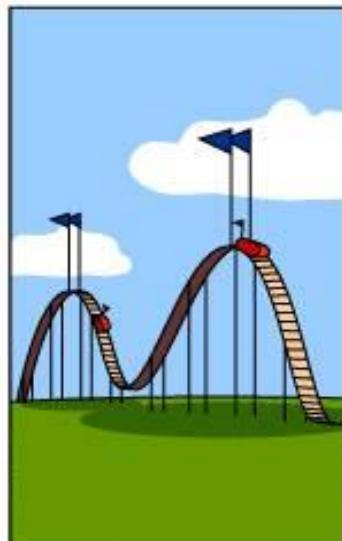
How the Business Consultant described it



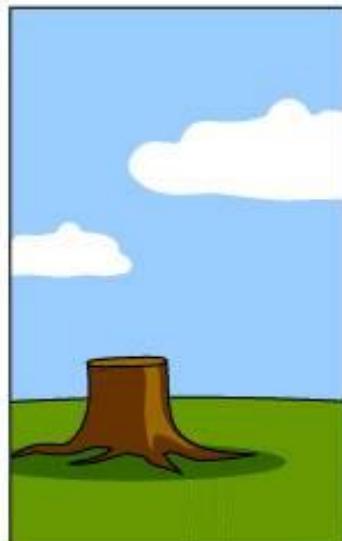
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

# What Is Software ?

- ❑ Programs

- ❑ Source code

- ❑ Data structures

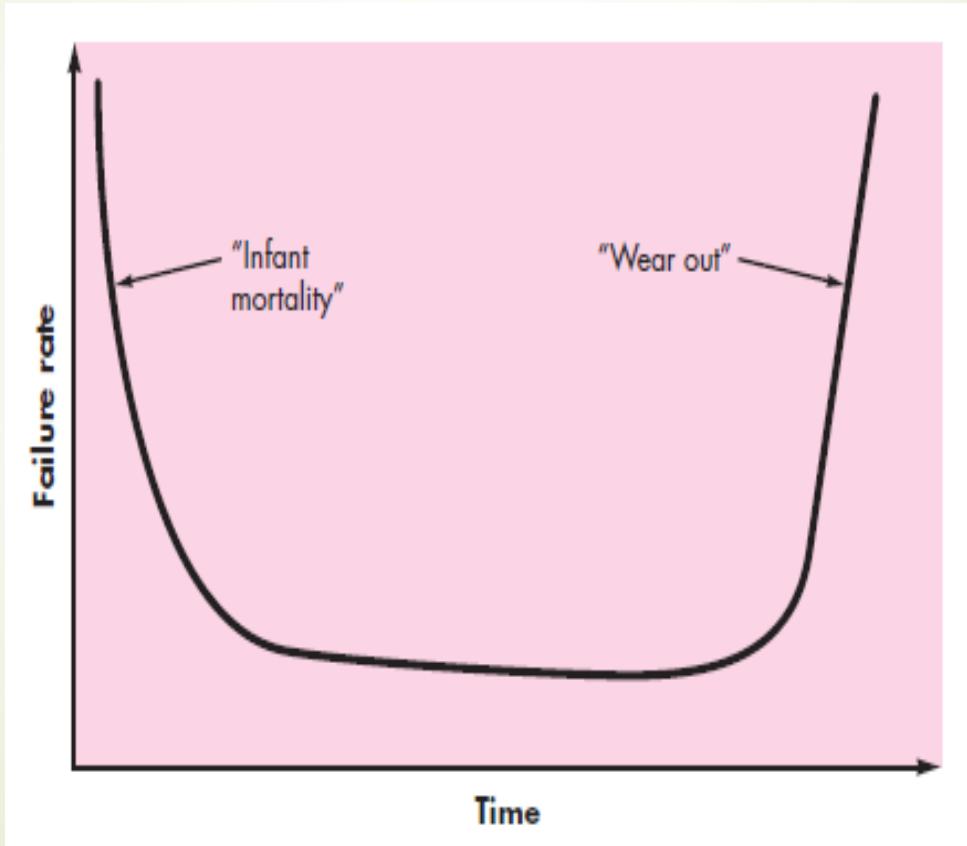
- ❑ Documents

- ❑ Requirements and specification documents

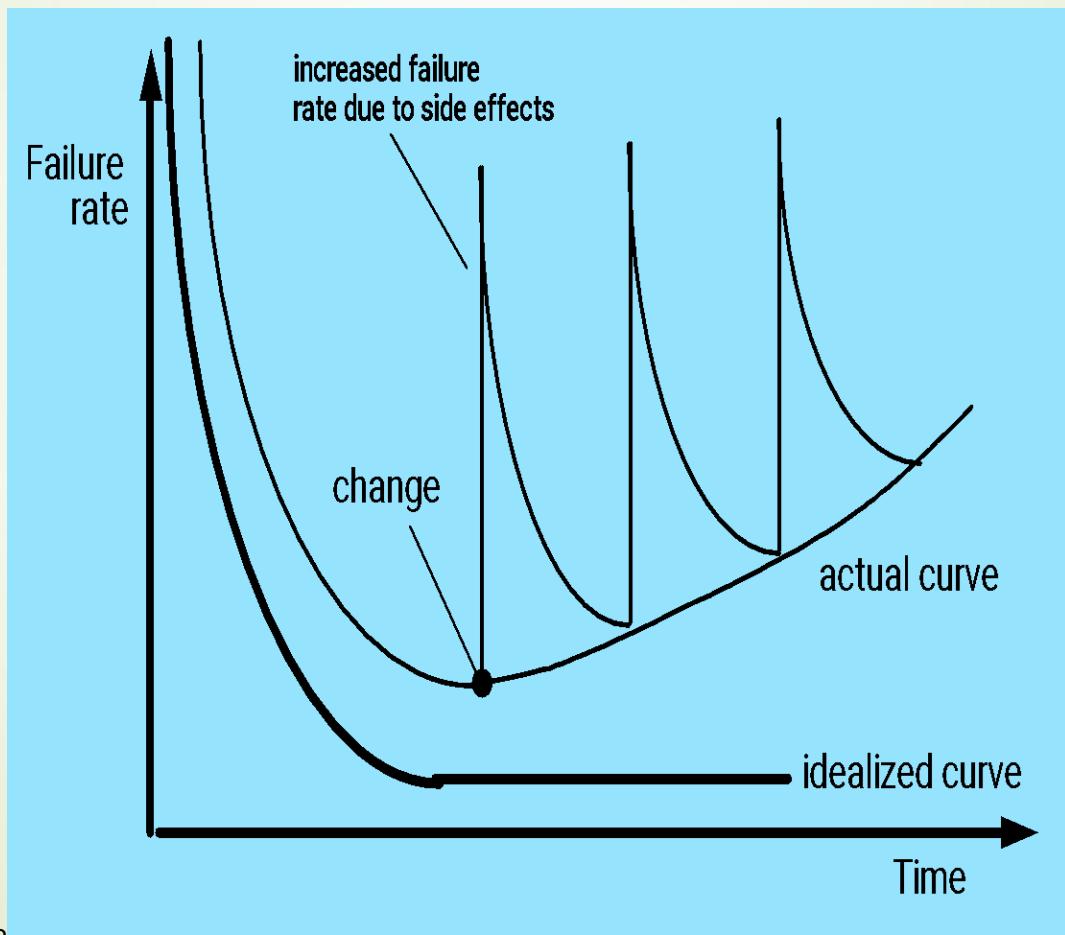
- ❑ Design documents

- ❑ Test suites and test plans

# Hardware Failure Curve



# Software failures



# What Is Good Software?

- ❑ **Correct, correct, correct**
- ❑ **Maintainable** and easy to modify
- ❑ Well modularized with well-designed interfaces
- ❑ **Reliable and robust**
- ❑ Has a **good user interface**
- ❑ **Well documented**
  - ❑ Internal documentation for maintenance and modification
  - ❑ External documentation for end users
- ❑ **Efficient**
  - ❑ **Not wasteful of system resources**, cpu & memory
  - ❑ **Optimized data structures and algorithms**

# Goodness Goals Conflict

- ❑ All goodness attributes cost \$s to achieve
- ❑ Interaction between attributes
  - ❑ High efficiency may degrade maintainability, reliability
  - ❑ More complex user interface may degrade efficiency, maintainability, and reliability
  - ❑ Better documentation may divert effort from efficiency and reliability
- ❑ Software engineering management has to trade-off satisfying goodness goals

# Legacy Software

*Why must it  
change?*

- ❑ software must be **adapted** to meet the needs of new computing environments or technology.
- ❑ software must be **enhanced** to implement new business requirements.
- ❑ software must be **extended to make it interoperable** with other more modern systems or databases.
- ❑ software must be **re-architected** to make it viable within a network environment.



# Software Myths

## ❓ Management myths

- ❓ We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know.
- ❓ My people do have state-of-the-art software development tools
- ❓ If we get behind schedule, we can add more programmers and catch up – Mongolian Horde Concept

# Software Myths...

## ❓ Customer myths

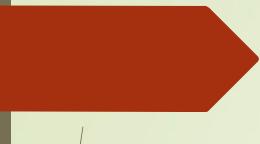
- ❓ A general statement of objectives is sufficient to begin writing programs...we can fill in the details later.
- ❓ Project requirement continually change, but change can be easily accommodated because software is flexible.



# Software Myths...

## ❓ Developer myths

- ❓ Once we write the program and get it to work, our job is done.
- ❓ Until I get the program “running” I really have no way of assessing its quality.
- ❓ The only deliverable for a successful project is a working program.



# Need Different Approaches for Developing Large Software

- ▣ Need formal **management** of software production process.
- ▣ Formal & detailed statement of requirements, specification and design.
- ▣ Much more attention to modularity and interfaces.
- ▣ Must be separable into manageable pieces.
- ▣ Need version control.
- ▣ More emphasis of rigorous and thorough testing.
- ▣ Need to plan for long term maintenance and modification.
- ▣ Need much more documentation, internal and external.



# Why Is Software Development Hard?

- ❑ **Changing requirements** and specifications
- ❑ **Inability to develop complete** and correct requirements
- ❑ **Programmer variability** and unpredictability
- ❑ **Communication and coordination**
- ❑ **Imprecise and incomplete requirements and specifications**
- ❑ **Inadequate software development tools**
- ❑ **Inability to accurately estimate** effort or time required
- ❑ Overwhelming complexity of large systems, more than linear growth in complexity with size of the system
- ❑ Poor software development processes
- ❑ **Lack of attention to issues of software architecture**

# What is Software Engineering?

The science ( & art) of building *high quality* software systems

- ❑ On **time**
- ❑ On **budget**
- ❑ With **correct** operation
- ❑ With **acceptable performance**

Software Engineering:

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of Software



# SE Framework Activities

- Communication
  - Requirement collection
- Planning
  - Specification
- Modeling
  - Requirement Analysis
  - Design
- Construction
  - Code generation
  - Testing
- Deployment
  - Release
  - Maintenance

# Major Software Production Tasks

## ② Requirements analysis:

- ② Analyze software system requirements in detail

## ② Specification:

- ② Develop a detailed specification for the software

## ② Design:

- ② Develop detailed design for the software data structures, software architecture procedural detail, interfaces

## ② Coding:

- ② Transform design into one or more programming language(s)

## ② Testing:

- ② Test internal operation of the system and externally visible operations & performance

## ② Release:

- ② Package and deliver software to users

## ② Maintenance:

- ② Error correction and enhancement after system

# The Essence of Practice

?

Polya suggests:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).



# Understand the Problem

- ❑ *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- ❑ *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- ❑ *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- ❑ *Can the problem be represented graphically?* Can an analysis model be created?

# Plan the Solution

- ② *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- ② *Has a similar problem been solved?* If so, are elements of the solution reusable?
- ② *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- ② *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

# Carry Out the Plan

- ❑ *Does the solution conform to the plan?* Is source code traceable to the design model?
- ❑ *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

# Examine the Result

- ② *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- ② *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

# SOFTWARE ENGINEERING

CSE 470 – Waterfall Model

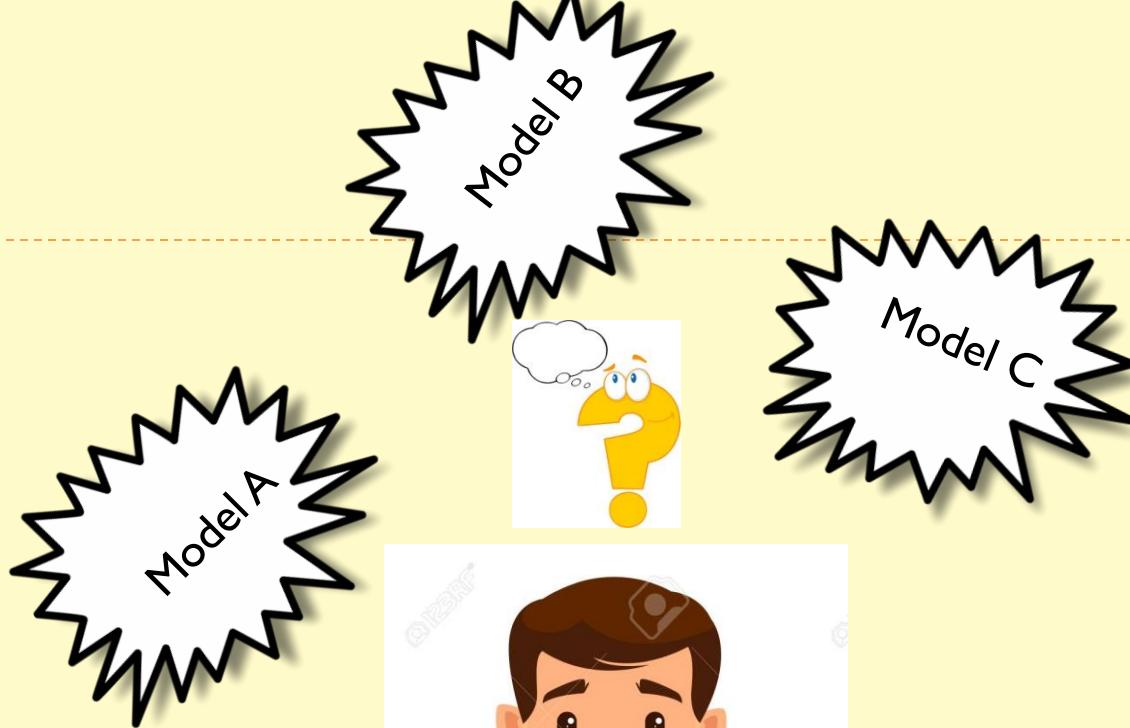
BRAC University





Team Lead







We know  
What we Want

Small Customer  
Base



A photograph of a waterfall cascading down a rocky cliff into a pool of water, surrounded by dense green foliage and misty mountains under a clear sky.

# Waterfall Model

- ▶ A sequential methodology for software project management.



**Requirement  
Analysis**

**Design**

**Coding**

**Testing**

**Deployment**

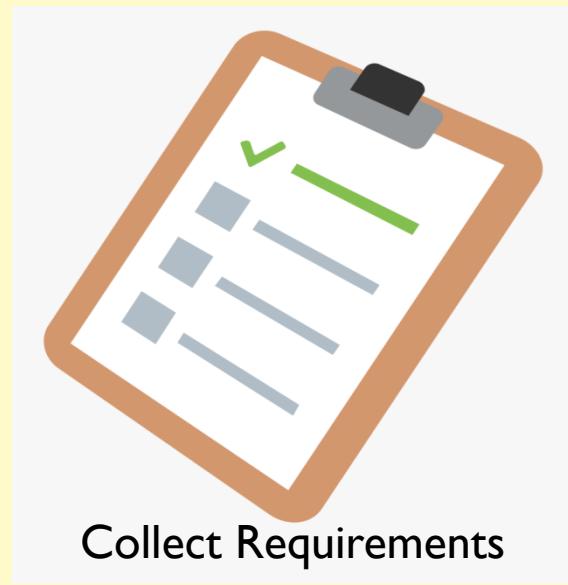
**Maintenance**

# Requirement Collection



Client Meeting

It starts with the concept about what the customer wants to do.

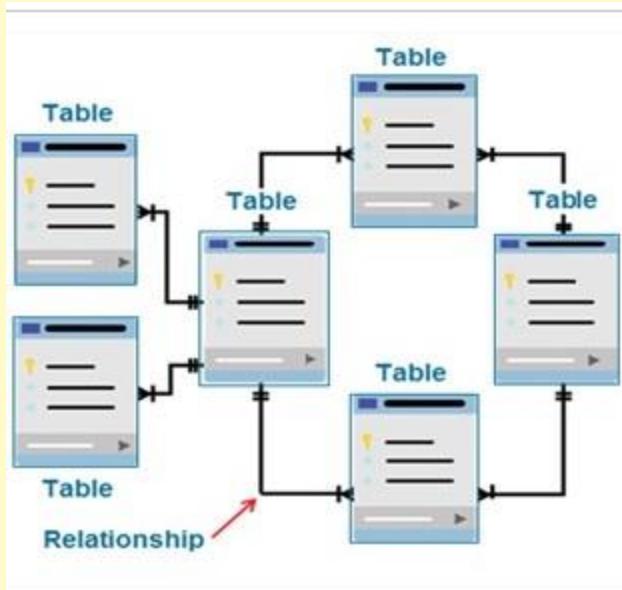




1. Address the problem
2. Identify the feasible and non-feasible requirements
3. Identify how the software will meet the customer requirements

APPROVED

# Design



Creates the logical and physical design of the software project



APPROVED

# Coding

1. We need to build it first
2. Coding can not start until design is fixed properly
3. Starts with converting the design in actual running software.
4. The design is split into blocks, and blocks are converted to code modules one after another.



APPROVED

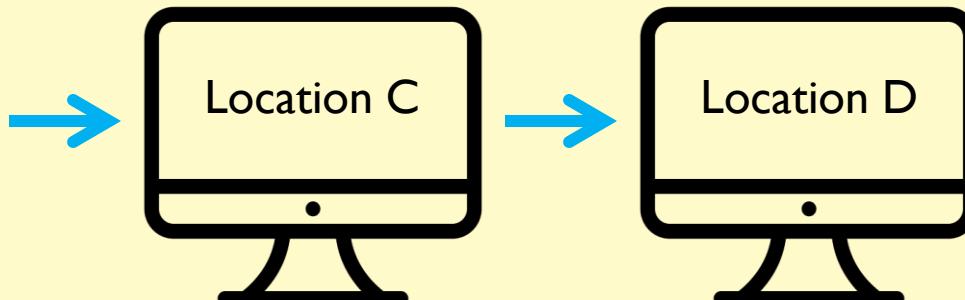
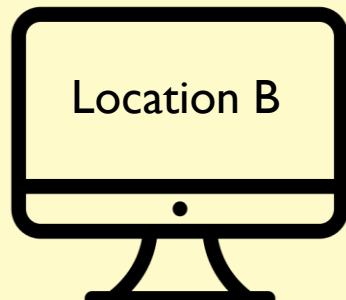
# Testing



1. Check the software against the requirements set at Requirement Analysis Phase.
2. In case of any problem, that problem is fixed in the code.

APPROVED

# Deployment and Maintenance



1. The software goes in production in actual information technology environment, specially goes to user environment.
2. Step by step deployment and maintenance performed
3. If anything goes wrong here will be maintained and resolved by the team.
4. Feedback may also be collected

# When to choose Waterfall Model



Requirements are well known



Small scale and short term project



Resources are available and trained



Technological tools required are not dynamic, instead are stable

# Advantages & Disadvantages

1. Simple to Use and Easy
2. Stages go one by one, so sudden changes can not create confusions
3. Any changes is done only in Development stage, so no need to get back and change everything.



1. While completing a stage, it freezes all the subsequent stages.
2. No way to verify the design
3. Once in testing phase, no more features can be added
4. Code Reuse not possible

# Example Case

- I. One of your uncle requested you to develop an accounting calculator for his local shop.
2. Your start-up company wants to develop an accounting calculator for super shops.

***Will you use Waterfall model for both Case 1 and 2 ?***



# SOFTWARE ENGINEERING

CSE 470 – V Model

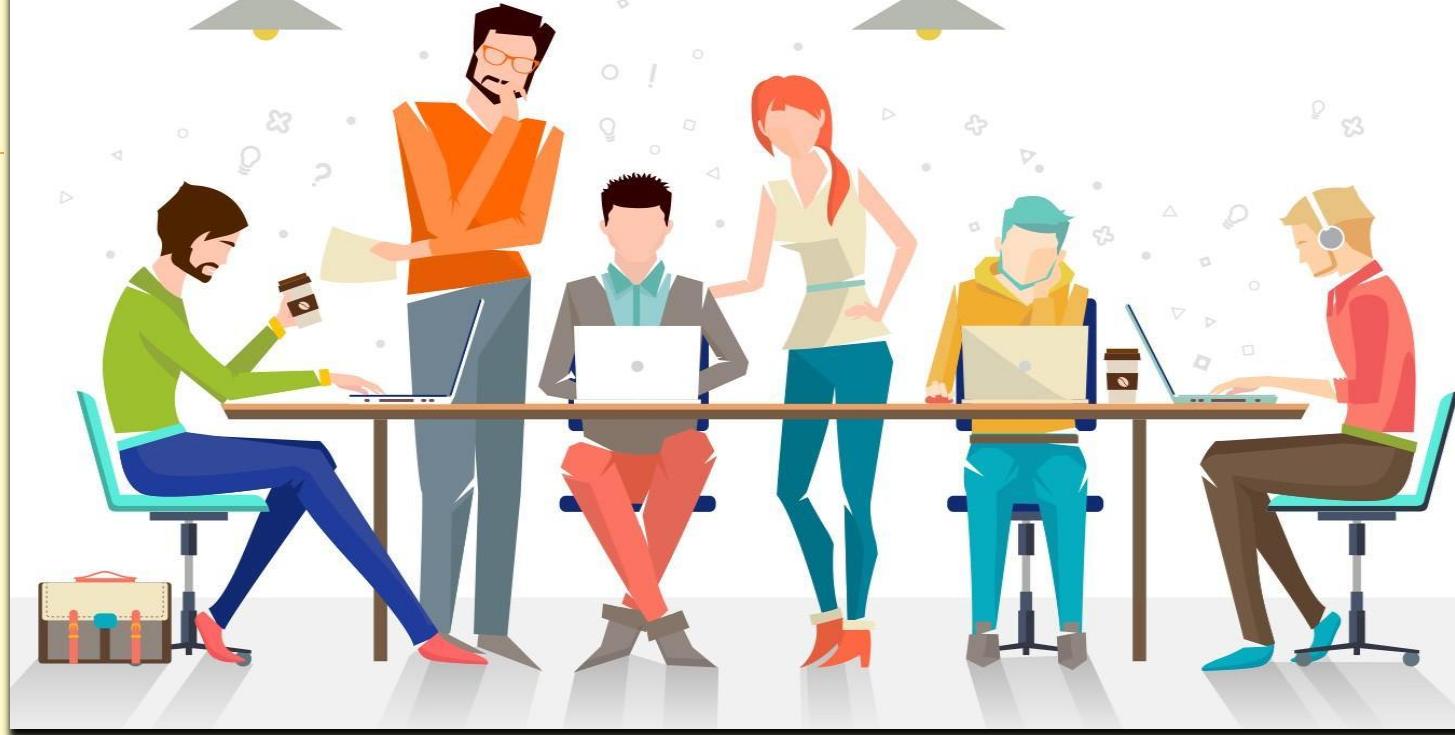
BRAC University



Inspiring Excellence



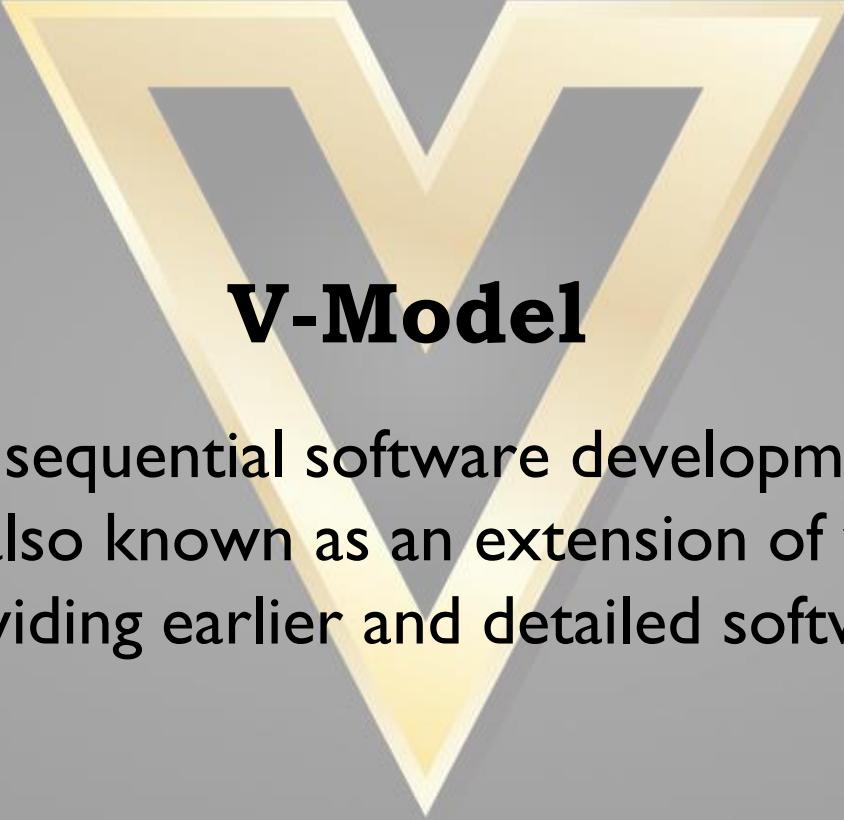
Team  
Lead



Customer asked to  
emphasize on proper  
testing of the accounting  
software.....!!!

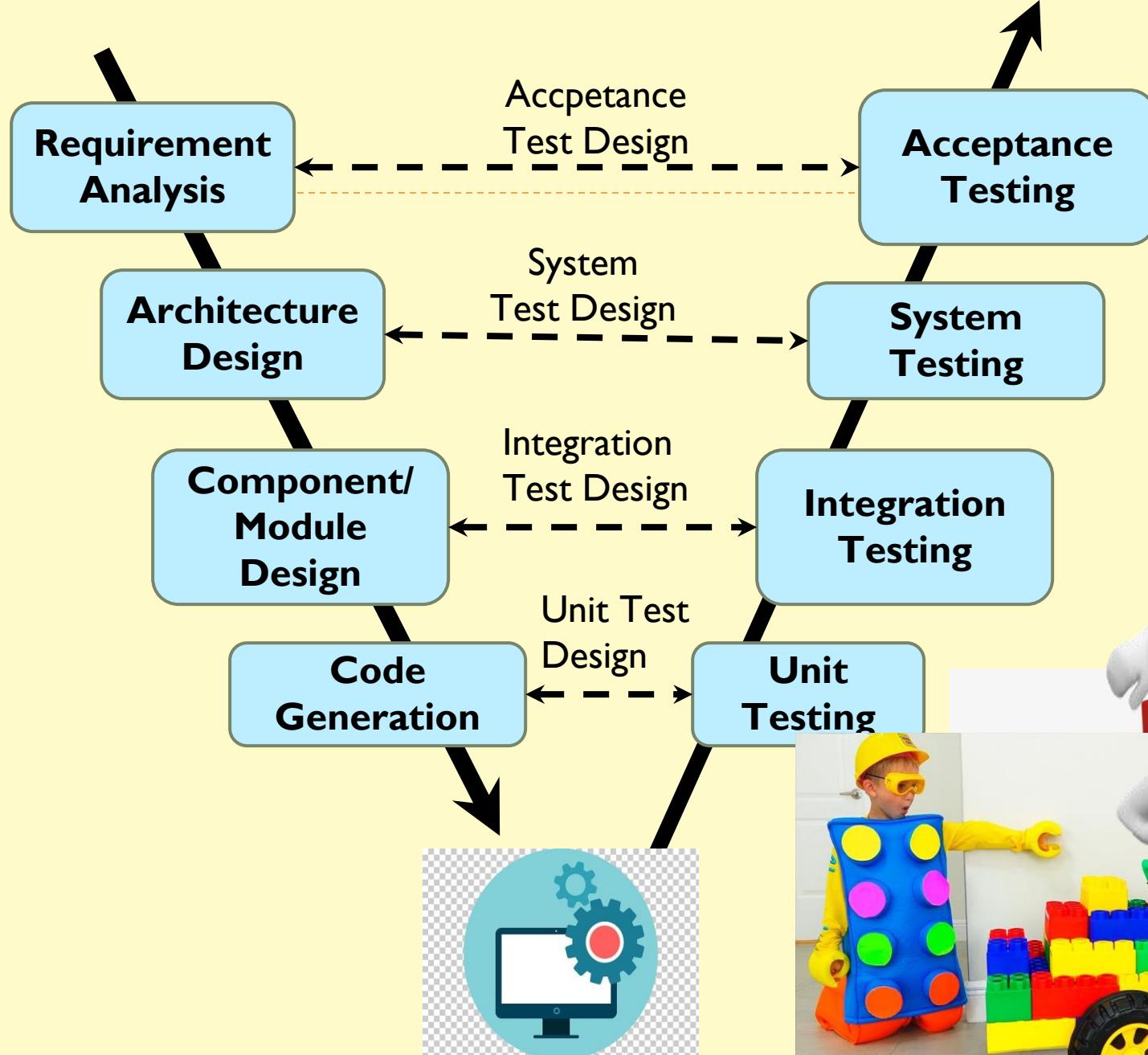
Requirement is  
Rigid.





## **V-Model**

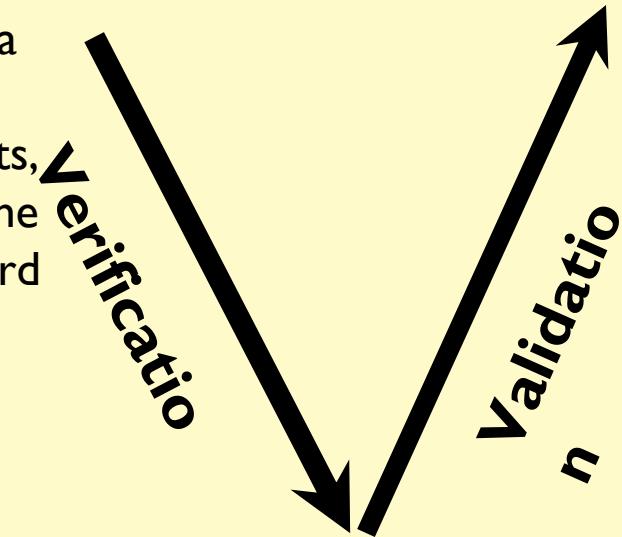
Its another sequential software development process model. Its also known as an extension of waterfall model, due to providing earlier and detailed software testing.



# Verification and Validation

Suppose, you have received a request for developing a android app. In this regard, you meet the customers, collected requirements, documented the requirements, created designs to built it and finally coded it as per the design. That means you are trying to follow all standard ways to develop the app rightly.

***Building the product in the right way is called “Verification”.***

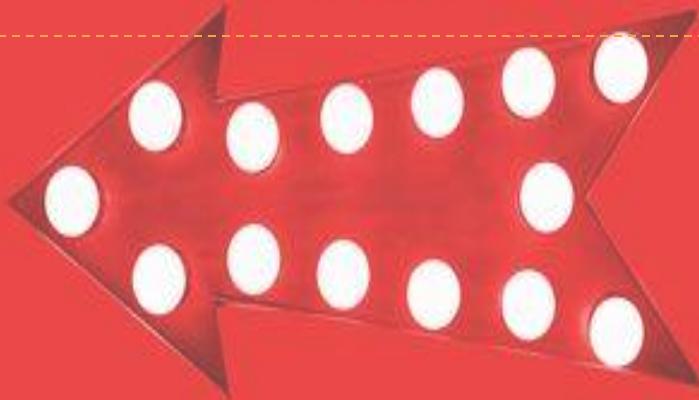


After completion of the app coding, we sent it to testing phase. Various tests are performed to ensure whether the app meets the customer requirements. That is you try to check whether the right app is developed.

***Building the right product is called “Validation”.***

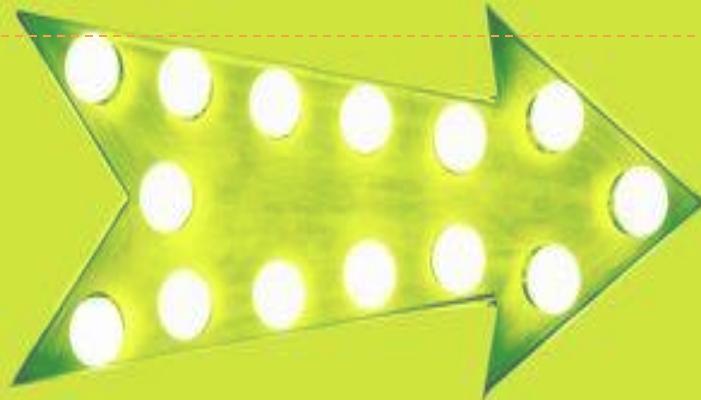


# CONS



1. Changes are not welcomed
2. Software developed at end of all phases, so no dummy prototypes can be found
3. If any test fails, then test document and code both needs to be updated

# PROS



1. Along with waterfall advantages, it emphasizes planning for verification and validation (V&V) of the product from the very beginning of requirement collection.
2. Test activities planned before testing
3. Saves time over waterfall, higher chance of success





# SOFTWARE ENGINEERING

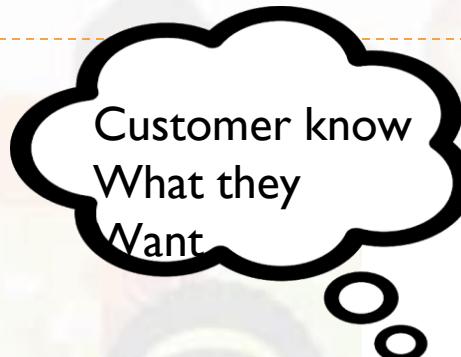
CSE 470 – Incremental and Iterative  
Model

BRAC University

# Sequential Process Model

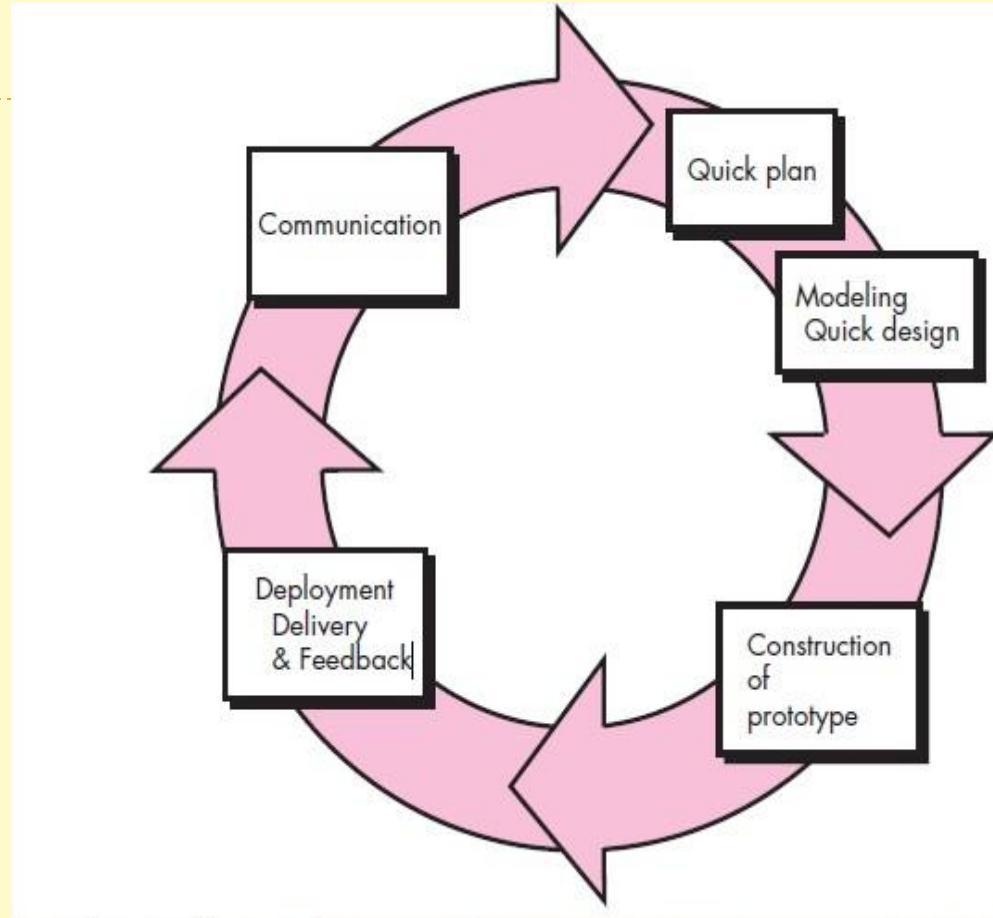


Team  
Lead



Customer Don't  
Know what they  
want !!!

# Evolutionary Process Models



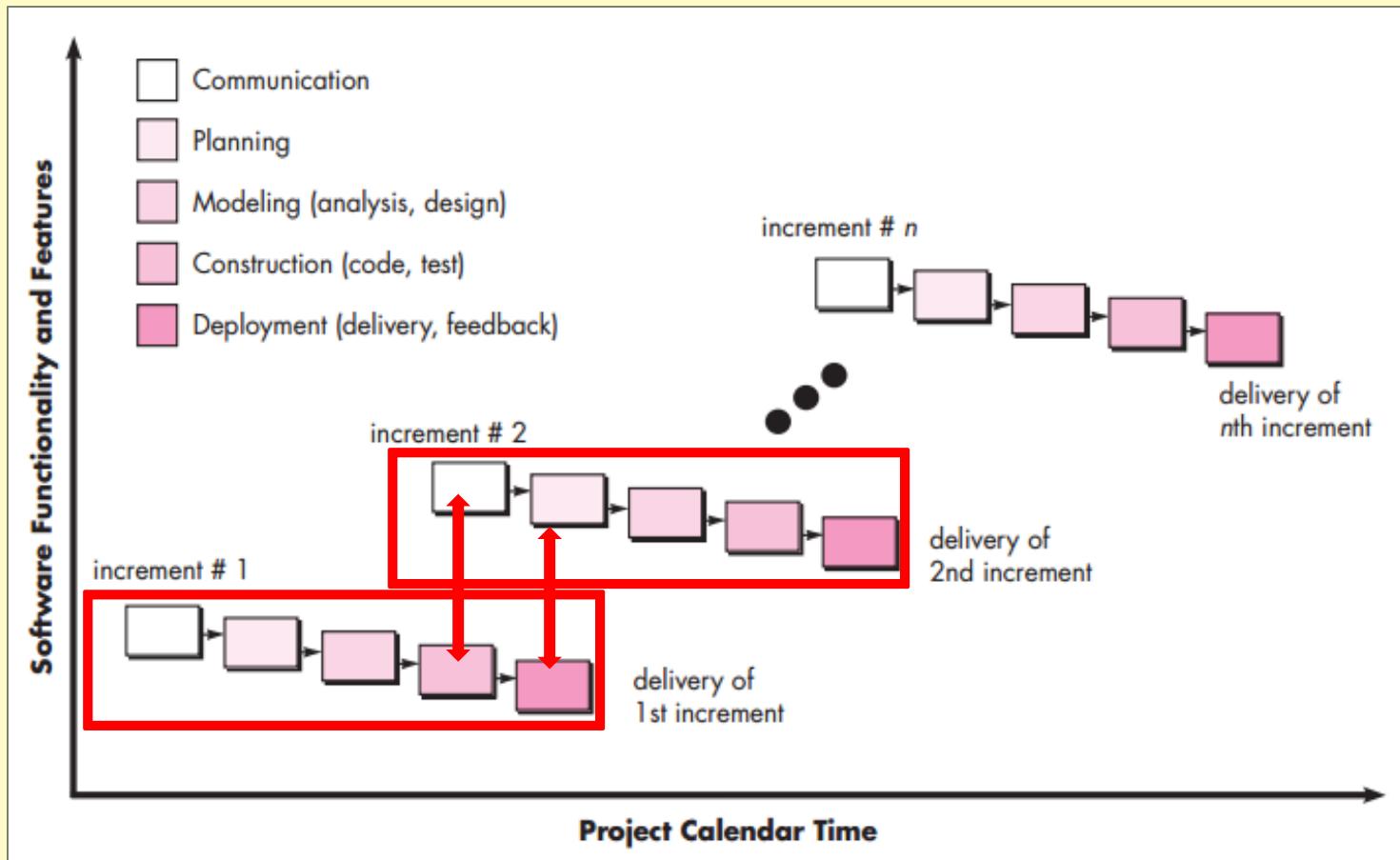
1. Can change requirements as you want.
2. Can go back to previous phases, such as after coding, we can go back to communication phase for requirement collection again.

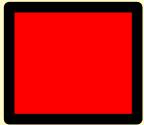
# **Incremental Process Model**

A software process model where the software will be delivered in increments

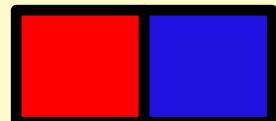
1. Customer wants to use the software from the very beginning of the project
2. Customer is quite well-known about the requirements
3. The software is divided into fixed number of increments to be delivered to customers

# Incremental Process Model

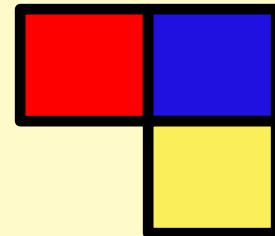
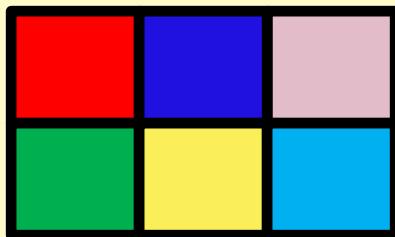




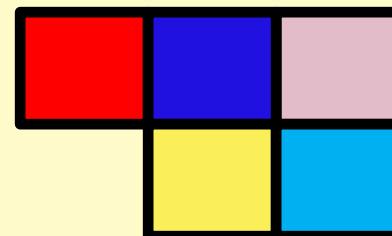
*Increment  
1*



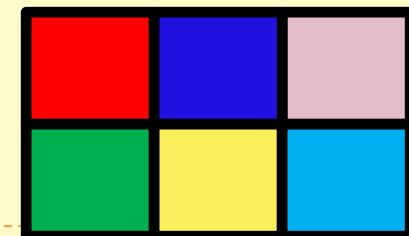
*Increment  
2*



*Increment  
3*

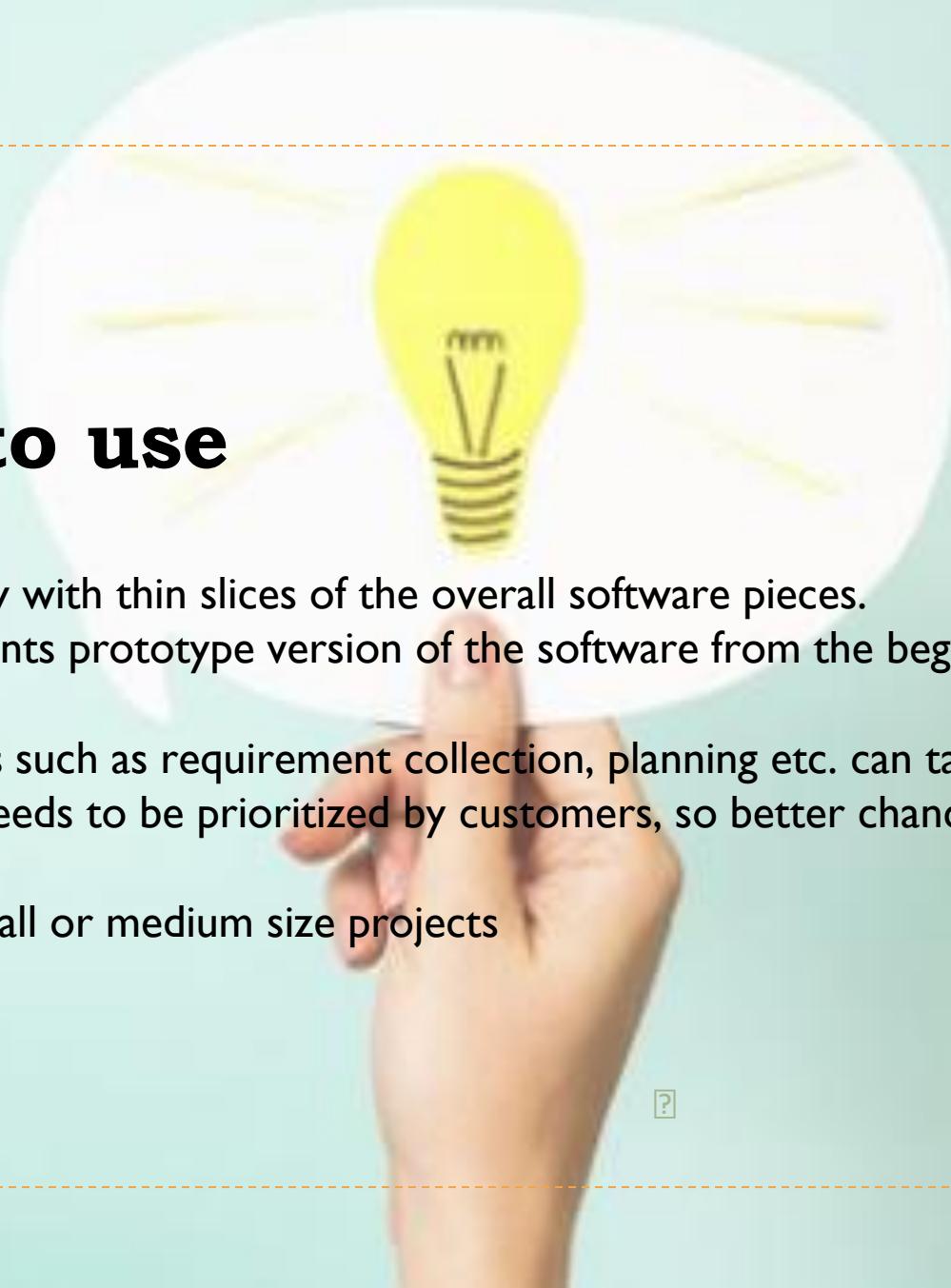


*Increment  
4*



*Increment*

1. For example, in first increment Login module is delivered. In second increment, another new module such as Navigation module is also added. In third, one more added.
2. That is, this model “adds onto” new section as increments.



## When to use

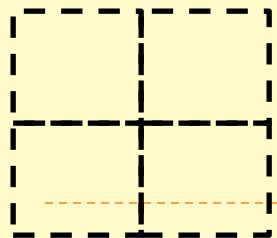
1. You are ready with thin slices of the overall software pieces.
2. Customer wants prototype version of the software from the beginning of the project
3. Parallel stages such as requirement collection, planning etc. can take place.
4. Increments needs to be prioritized by customers, so better chance of success
5. Better for small or medium size projects



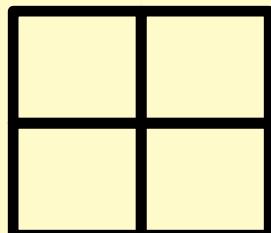
# **Iterative Process Model**

A software process model where the software will be delivered in iterations

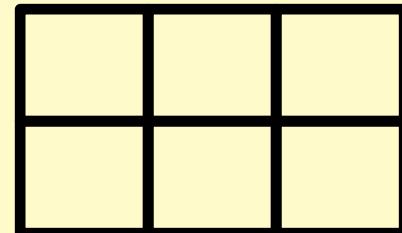
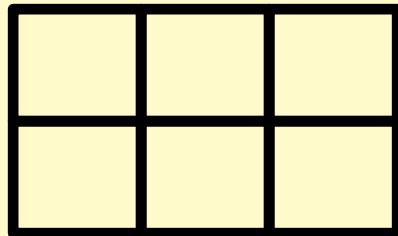
1. Customer is not sure about the requirements
2. Even development team may not be sure about which technology, algorithms may be used.
3. There is no fixed limit of iterations, that is time is set aside.



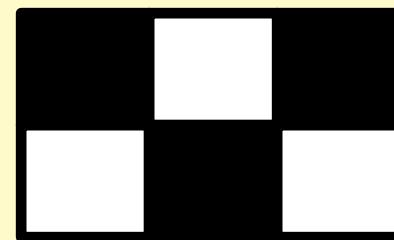
*Iteration  
1*



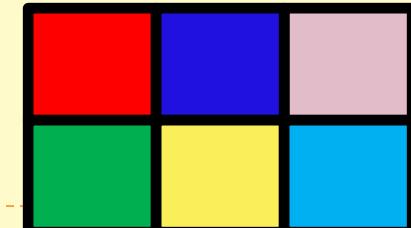
*Iteration  
2*



*Iteration  
3*



*Iteration  
4*



*Iteration  
4*

1. For example, in first iteration Login module is delivered. In second iteration, Login module is updated again. In third iteration, Navigation module is added . And in fourth iteration some refinement is done.

2. That is, this model “changes or reworks” on same section in iterations until customer accepts it.

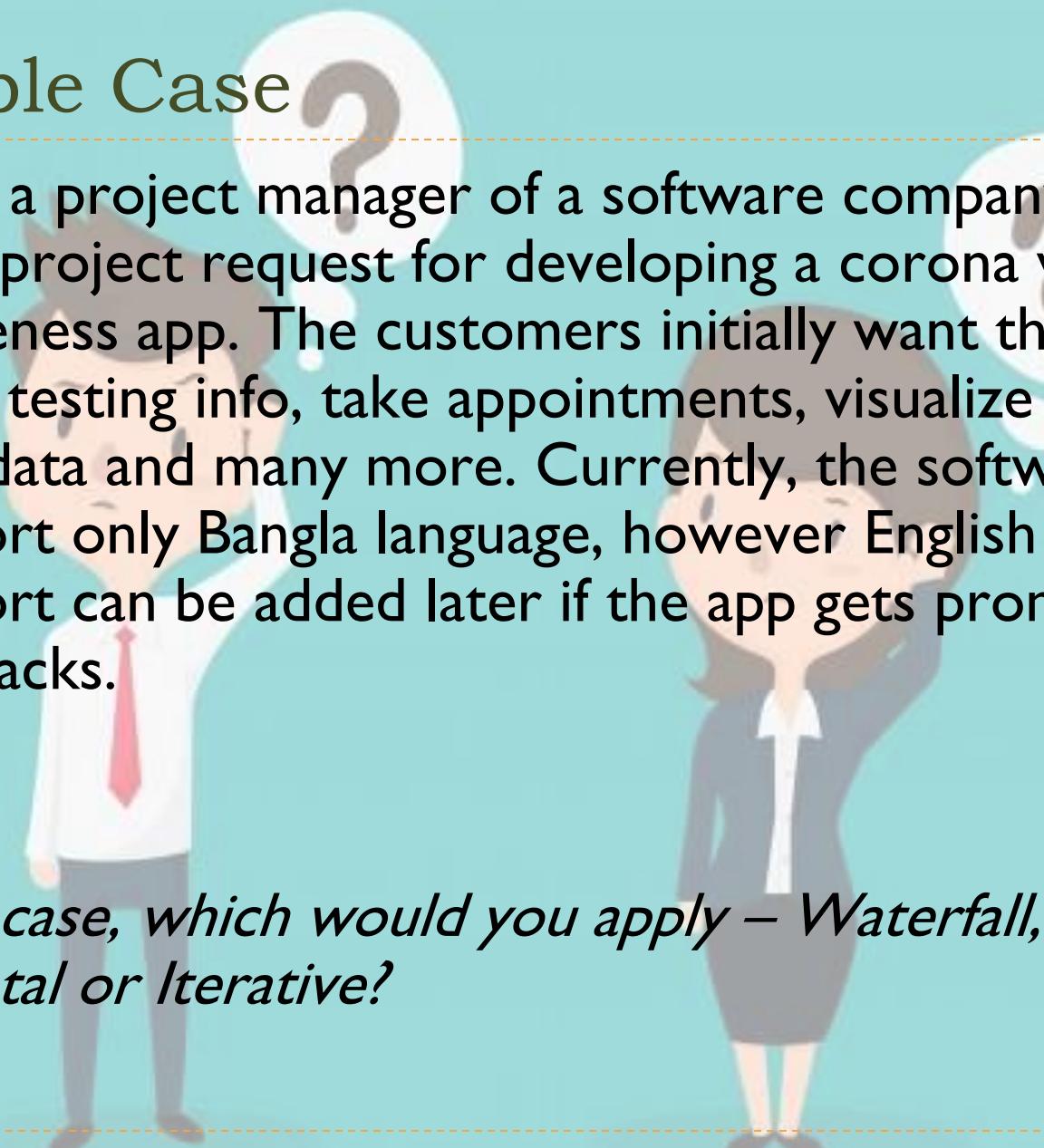


## When to use

1. Requirements are not fixed
2. Technological tools or requirements are not identified yet.
3. Instead of fixed time, quality of the features is refined with time
4. Customer feedbacks with repetitive iterations increase the product quality
5. Better for long-term and complex projects



# Example Case

- 
- A background illustration featuring two stylized human figures from the waist up. They are wearing white shirts, red ties, and dark trousers. A large, light gray question mark is positioned between them, partially obscuring their faces. The figure on the left has short brown hair and is looking towards the right. The figure on the right has long brown hair tied back and is looking towards the left.
- I. Being a project manager of a software company, you have got a project request for developing a corona virus awareness app. The customers initially want the app to show testing info, take appointments, visualize affected area data and many more. Currently, the software should support only Bangla language, however English language support can be added later if the app gets promising feedbacks.

*In such a case, which would you apply – Waterfall, Incremental or Iterative?*



# Software Engineering

## The Software Process(Spiral&CMMI)

Slides for cse470 video lecture series produced by:

A.M.Esfar-E-Alam

Afrina Khatun

Dr.Muhammad Zavid Parvez

# Spiral Model

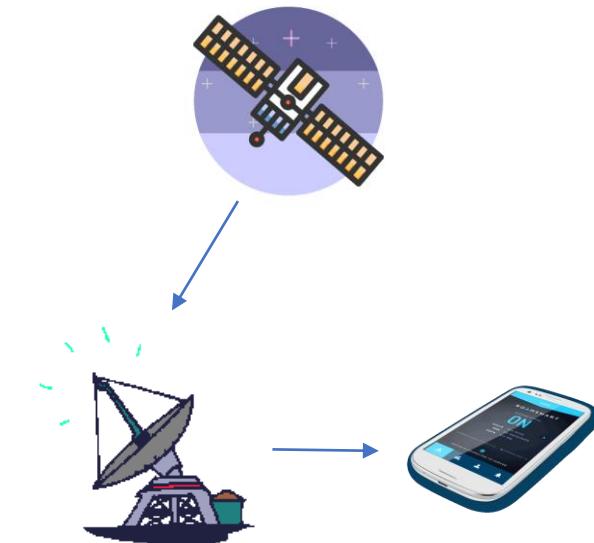
- Rarely Used but an important model
- Its a model that works for projects with unlimited budget, time and projects that has huge risk factors.
- Example, making a heavy lift system for space shuttle and international space station.
- Another example can be about a company name Galaxy inc.



- They wanted to send 6 dozens satellite in space and build a satellite based cellular system.
- So that remote places like even in Antarctica where you don't have any BTS(mobile tower) you can still be able to communicate using your cell.
- You are never out of network.

As you can see for this project:

- Risk were enormous
- Needs a huge budget
- No published materials or experienced worker
- Risks will be coming and identified once the project kicks off
- Several million codes had to be written and you don't even have Stackoverflow...

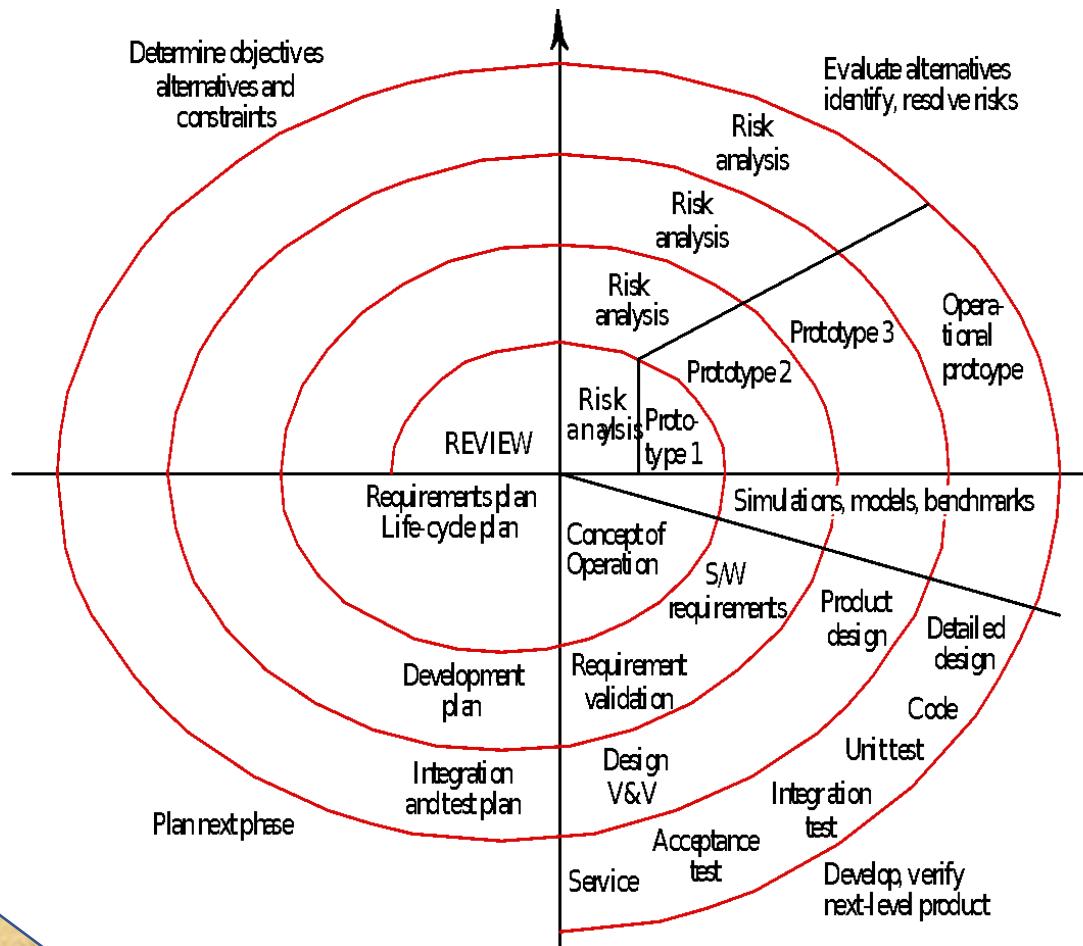


# Spiral Model Formal Definition

- The **spiral model** is a **risk-driven** process model generator for software projects. Based on the unique risk patterns of a given project, the spiral model guides a team to adopt elements of one or more process models, such as incremental, waterfall, or evolutionary prototyping.
- This model was first described by **Barry Boehm** in his 1986 paper "A Spiral Model of Software Development and Enhancement".



# Spiral model Figure



- Here is the image depicting spiral model.
- As you can see spiral loops showing phase by phase development.
- You can see we are doing risk analysis in every phase, planning and keep building prototype until we reach our goal.

# Spiral model sectors

- Objective setting
  - Specific objectives for the phase are identified
- Risk assessment and reduction
  - Risks are assessed and activities put in place to reduce key risks
- Development and validation
  - A development model for the system is chosen which can be any of the generic models
- Planning
  - The project is reviewed and next phase of the spiral is planned



# Spiral model usage

❑ Spiral model has been very influential in helping people to think about iteration in software processes and introducing the risk-driven approach to development. In practice, however as mentioned, the model is rarely used as published for practical software development.

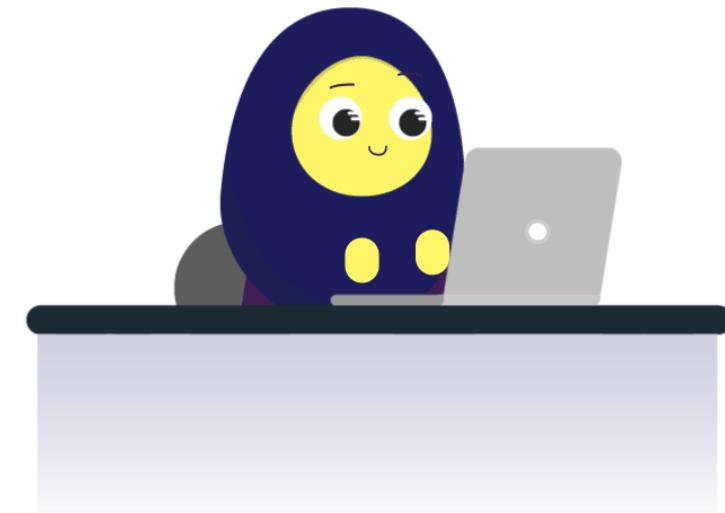
❑ So if you are a project manager or lead developer when would you suggest spiral model? It is if you have:

- Long term project commitment and budget
- Users and developers unsure of the needs
- Requirements are complex
- New product line
- Significant changes are expected(research and explanation)



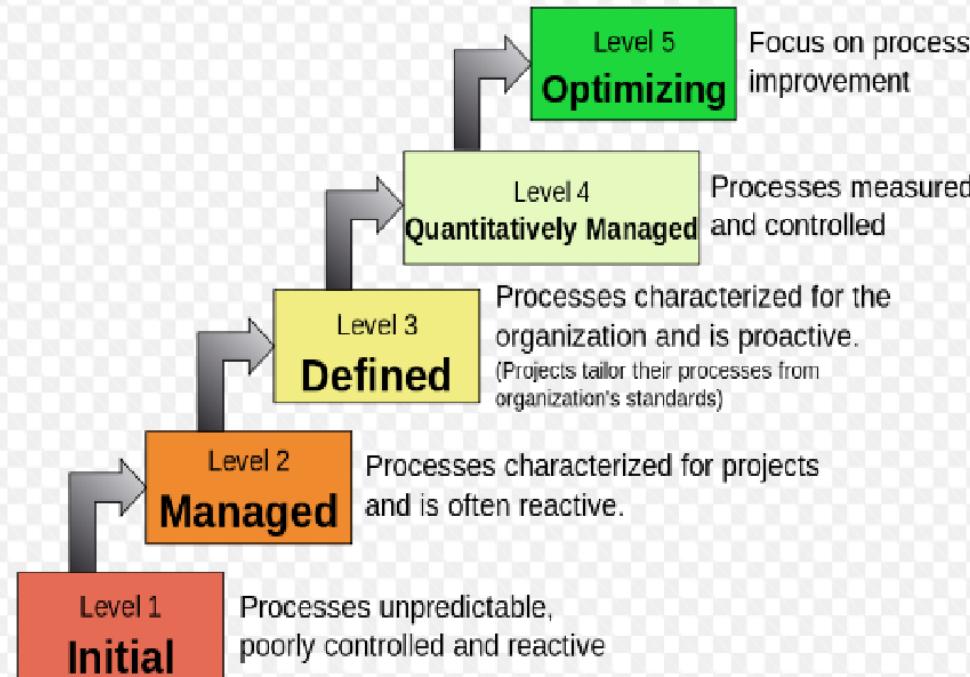
# CMMI

- ❑ The Capability Maturity Model Integration (CMMI) is a process and behavioral model that helps organizations streamline process improvement and encourage productive, efficient behaviors that decrease risks in software, product and service development.
- ❑ It is developed by CMU
- ❑ This process is mostly a requirement if you want to get a contract for software development in US govt organization.
- ❑ In this model work is divided in such a way so that you have different maturity level of a system you are building.
- ❑ It divided in 5 maturity level and you need to improve the system until you reach level 5



## CMMI: Capability Maturity Models Integrated

### Characteristics of the Maturity levels



34

- Once you reach level 5 that does not mean the end of your system.
- It means now the system is full proof, it just need regular maintenance nothing else.

<b>Level</b>	<b>Focus</b>	<b>Process Area</b>	
5 Optimizing	Continuous Process Improvement	<ul style="list-style-type: none"> <li>•Organizational Performance Management</li> </ul>	<ul style="list-style-type: none"> <li>•Causal Analysis &amp; Resolution</li> </ul>
4 Quantitatively Managed	Quantitative Management	<ul style="list-style-type: none"> <li>•Organizational Process Performance</li> </ul>	<ul style="list-style-type: none"> <li>•Quantitative Project Management</li> </ul>
3 Defined	Process Standardization	<ul style="list-style-type: none"> <li>•Requirements Development</li> <li>•Technical Solutions</li> <li>•Product Integration</li> <li>•Verification</li> <li>•Validation</li> <li>•Organizational Process Focus</li> </ul>	<ul style="list-style-type: none"> <li>•Organizational Process Definition</li> <li>•Organizational Training</li> <li>•Integrated Project Management</li> <li>•Risk Management</li> <li>•Decision Analysis &amp; Resolution</li> </ul>
2 Managed	Basic Project Management	<ul style="list-style-type: none"> <li>•Requirements Management</li> <li>•Project Planning</li> <li>•Project Monitoring &amp; Control</li> <li>•Supplier Agreement Management</li> </ul>	<ul style="list-style-type: none"> <li>•Measurement &amp; Analysis</li> <li>•Process &amp; Product Quality Assurance</li> <li>•Configuration Management</li> </ul>
1 Initial			

- Pause and go through this chart that will give you detailed Idea of what happens in each maturity levels

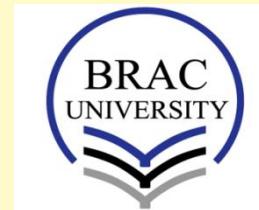




# SOFTWARE ENGINEERING

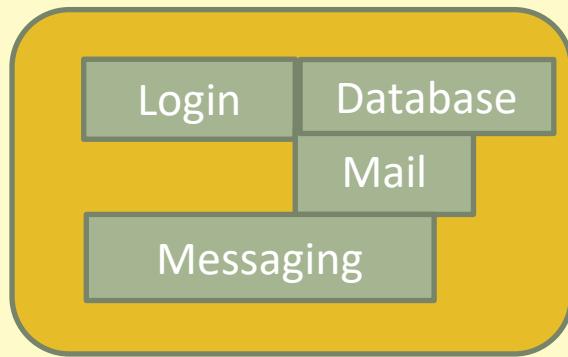
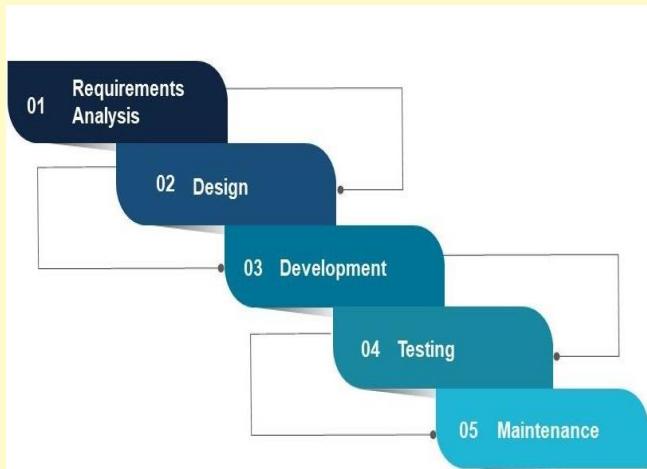
## CSE 470 – Agile Methodology

BRAC University



Inspiring Excellence

# Why we need Agile



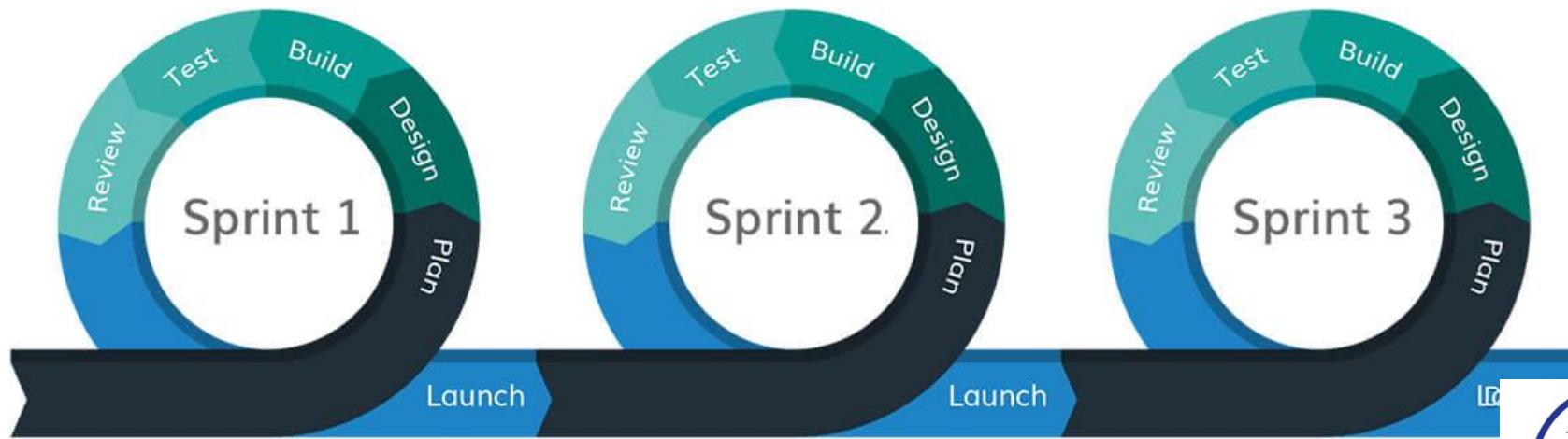
**1. Monolithic Software:** The ready product comes at the end of the process. now a days, the requirement of software changes frequently, even it can change a number of times in a day.

**2. For live systems, like Amazon, Facebook** a few seconds of downtime can cause a lot. Changes in such cases needs to be smooth enough so that customers interaction do not interrupt.

# What is ‘Agility?’

- Agile is a set of principles and values. It is a practice to be followed.
- Agile is a combination of Iterative and Incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software.

## Agile Methodology



# Agile Manifesto

*We are uncovering better ways of developing software by doing it and helping others do it.*



***Individuals and interactions***

over

***processes and tools***

***Working software***

over

***comprehensive documentation***

***Customer collaboration***

over

***contract negotiation***

***Responding to change***

over

***following a plan***

# Agile Methodologies

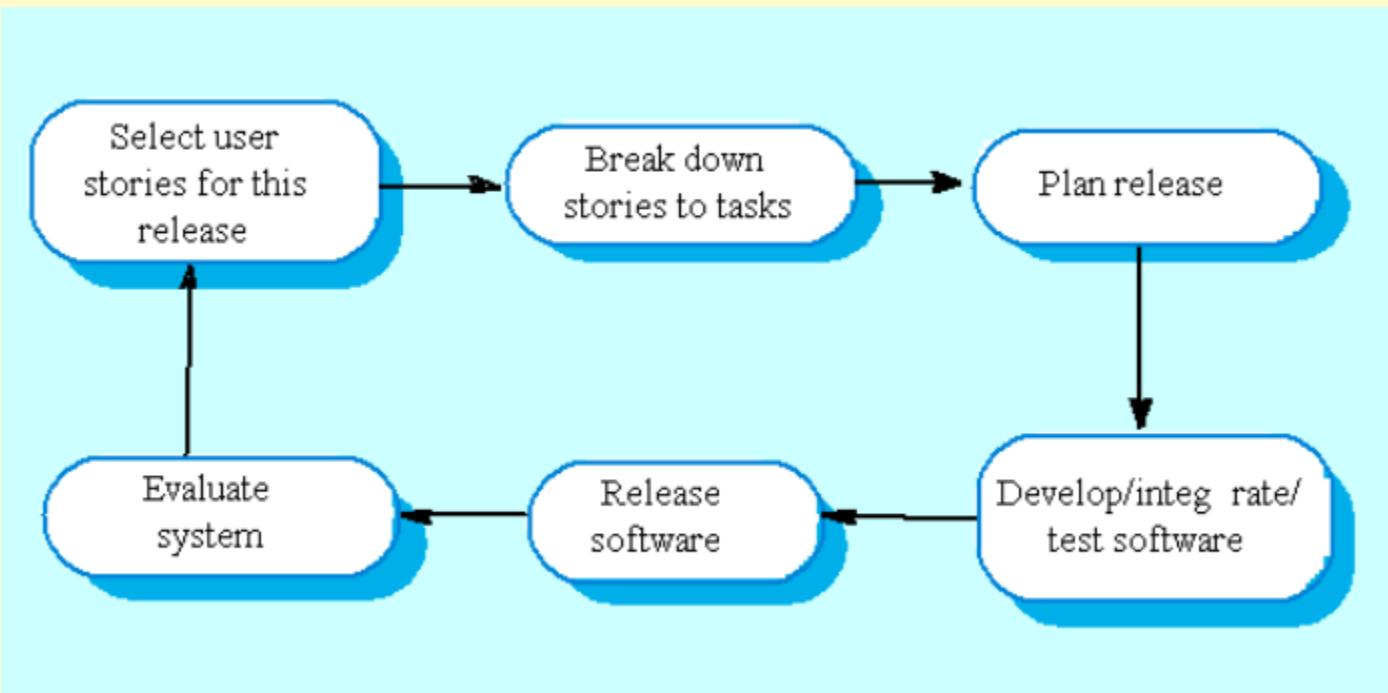
- Extreme Programming
- Agile Unified Process
- Scrum

# Extreme Programming

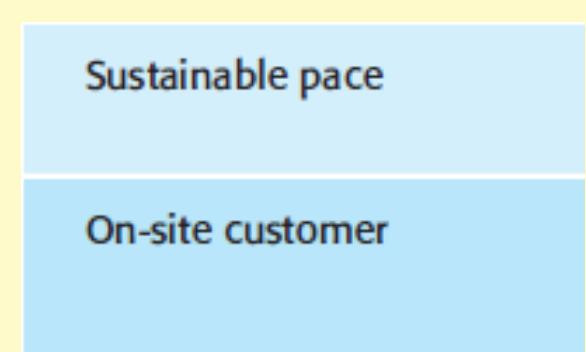
Agile Methodology

- In short also known as **XP**
- New versions may be built several times per day
- Increments are delivered to customers every 2 weeks
- All tests must be run for every build and the build is only accepted if tests run successfully.

# XP Workflow



# XP Principle or Practice



# Pair Programming

- ❖ Two people code in share, they switch roles from observer to -navigator frequently
- ❖ Observing code produces better code, so less business cost.
- ❖ Support each other, better chance for learning.



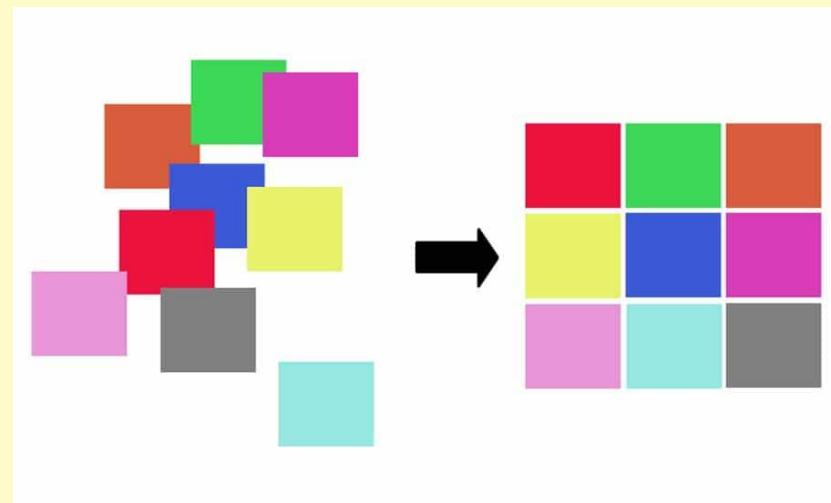
# Unit Testing

- ❖ Writes unit test for each method to identify problems and resolve it
- ❖ Test First Development (TDD) is used to ensure better code quality and less error. It refers to write test code first and after that write the development code.



# Refactoring

- It's a technique for restructuring existing code by changing its internal structure without changing its external behaviour.
- Refactoring includes reducing duplicate codes, breaking long classes/methods into small ones, appropriate variable naming and many more.
- It is done for requirement change, improving design and easy extendibility of software.



# **Agile Unified Process**

- Another Agile Software Development Methodology
- Agile Unified Process (AUP) is a simplified version of the Rational Unified Process (RUP).

# AUP Phases

## Inception

- Try to identify the business scope of the project, initial requirements and potential solution of the problem.

## Elaboration

- Design and prove the solution architecture, more requirements can be extracted

## Construction

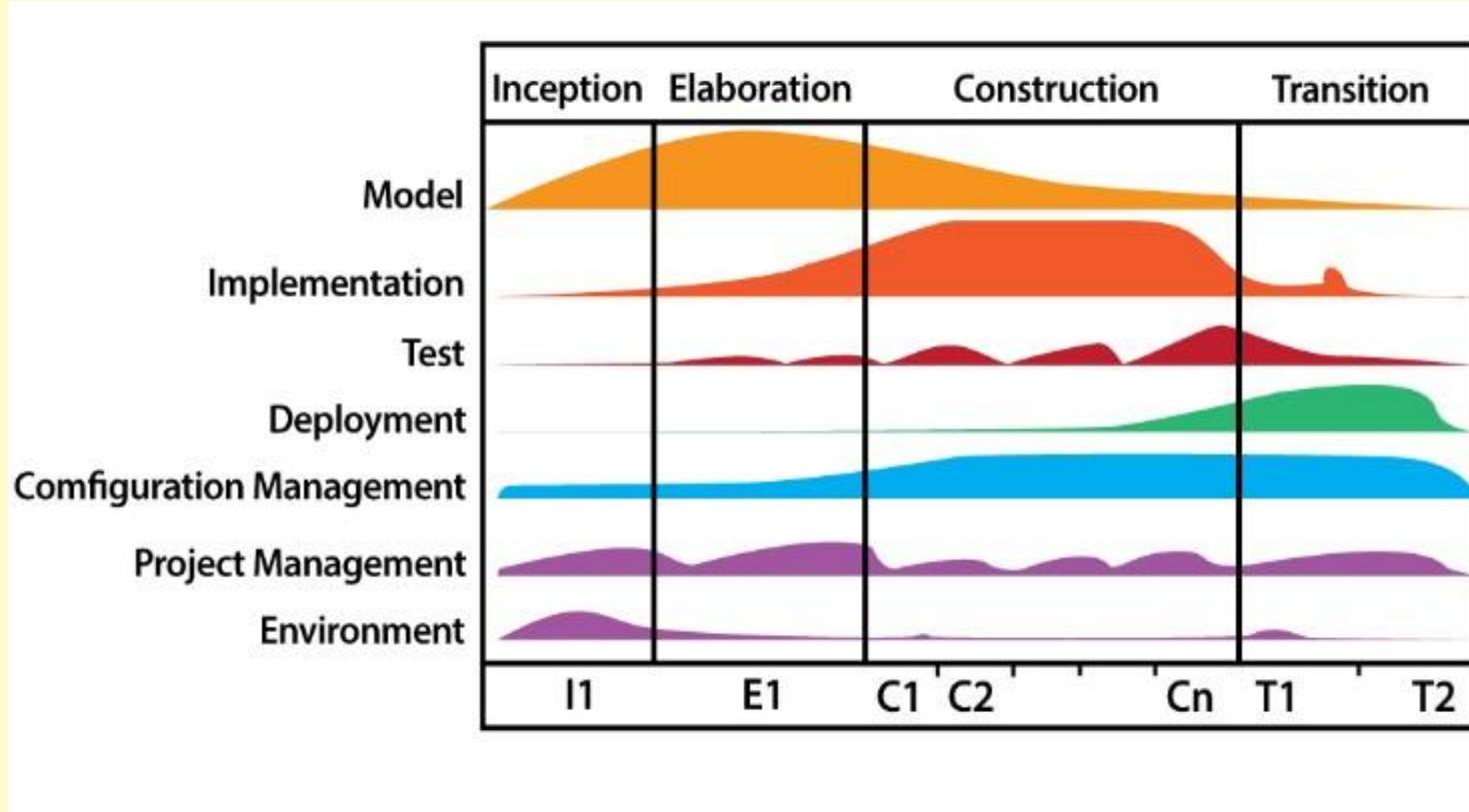
- Continuous implementation and testing ( specially unit testing) in form of iterations

## Transition

- Lastly, deploy the system in production environment and adjust feedbacks

# AUP Disciplines

- ❖ **Model**
- ❖ **Implementation**
- ❖ **Test**
- ❖ **Deployment**
- ❖ **Configuration Management:** Manage access to project artifacts/versions. This includes not only tracking artifact versions over time but also controlling and managing changes to them.
- ❖ **Project Management:** Direct the activities that take place within the project. This includes managing risks, directing people (assigning tasks, tracking progress, etc.), and coordinating with people and systems outside the scope of the project to be sure that it is delivered on time and within budget.
- ❖ **Environment:** Support the rest of the effort by ensuring that the proper process, guidance (standards and guidelines), and tools (hardware, software, etc.) are available for the team as needed.



# Scrum

- It is the most widely used Agile S/W development method for project management
- The full software is delivered in 14-30 days iterations

# The Agile - Scrum Framework

Inputs from Executives,  
Team, Stakeholders,  
Customers, Users



Product Owner



The Team



Product Backlog



Sprint Planning Meeting



Sprint Backlog



# Scrum Framework

## Roles

- Product owner
- Scrum Master
- Team

## Ceremonies

- Sprint planning
- Sprint review and Sprint retrospective
- Daily scrum meeting

## Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

# Scrum Roles



# Scrum Roles

## – Product Owner

- Possibly a Product Manager or Project Sponsor
- Decides features, release date, prioritization, \$\$\$



## – Scrum Master

- Typically a Project Manager or Team Leader
- Responsible for enacting Scrum values and practices
- Remove impediments / politics, keeps everyone productive

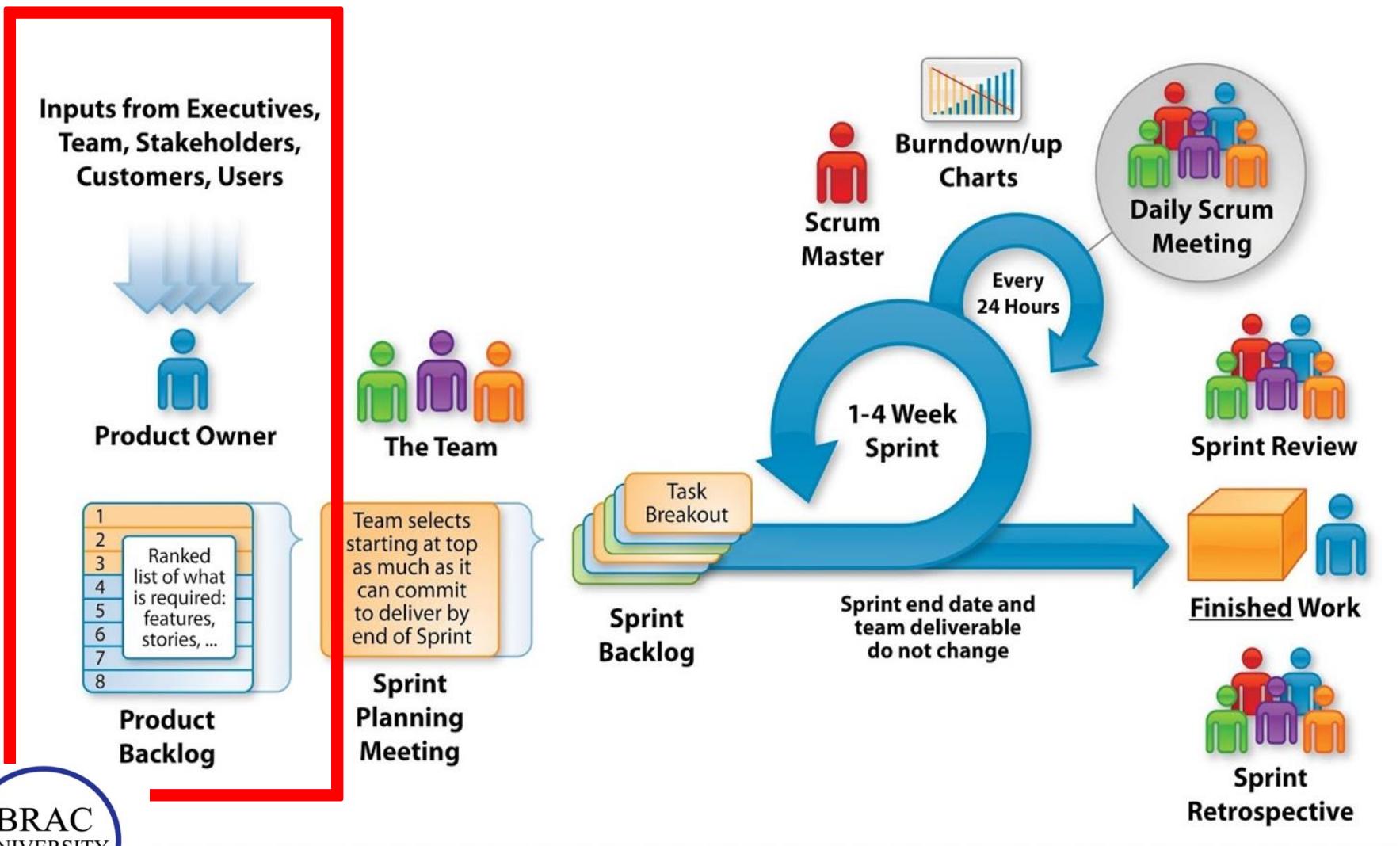


## – Project Team

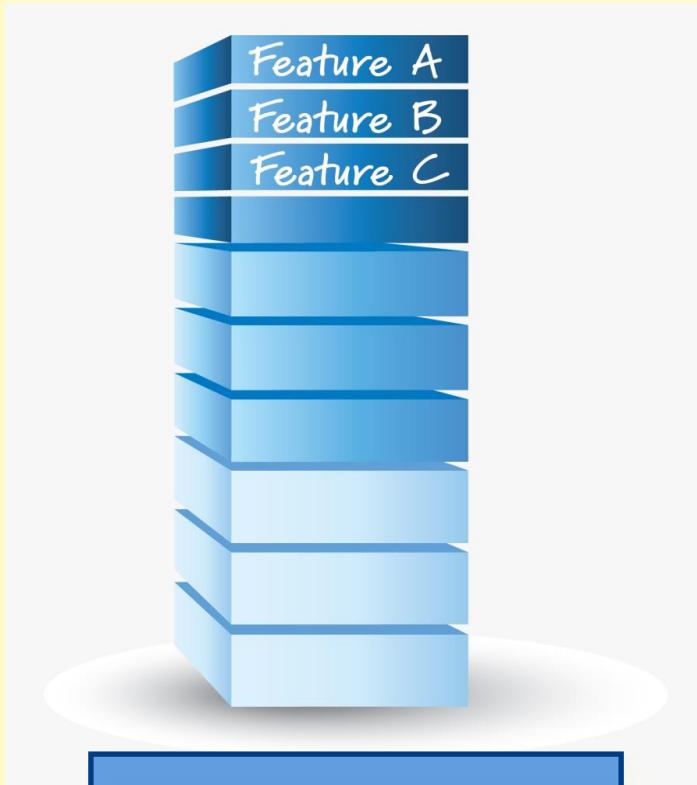
- 5-10 members; Teams are self-organizing
- Cross-functional: QA, Programmers, UI Designers, etc.
- Membership should change only between sprints



# The Agile - Scrum Framework



# Artifact - Product Backlog



This is the  
product backlog

- The requirements
- A list of all desired work on project
- Ideally expressed as a list of user stories along with "story points", such that each item has value to users or customers of the product
- Prioritized by the product owner
- Reprioritized at start of each sprint

# Artifact - Product Backlog

Feature	Backlog item	Estimate
A	Allow a guest to make a reservation	3 (story points)
B	As a guest, I want to cancel a reservation.	5
C	As a guest, I want to change the dates of a reservation.	3
D	As a hotel employee, I can run RevPAR reports (revenue-per-available-room)	8
E	Improve exception handling	8
...	...	30
...	...	50

# The Agile - Scrum Framework

Inputs from Executives,  
Team, Stakeholders,  
Customers, Users



Product Owner



The Team



Product Backlog

Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint Planning Meeting



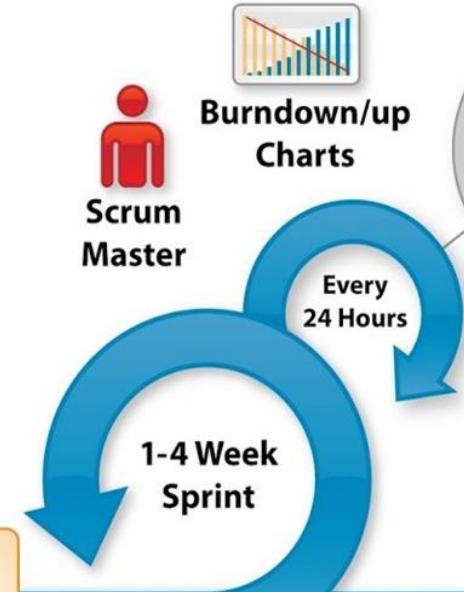
Sprint Backlog



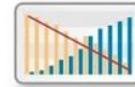
Burndown/up Charts  
Scrum Master

Every 24 Hours

1-4 Week Sprint



Sprint end date and team deliverable do not change



Daily Scrum Meeting



Sprint Review



Finished Work



S|  
Retro

# Artifact - Sprint Backlog



This is the sprint backlog

- High priority features are selected based on points from product backlog.
- These features will be covered within that 14 days time slot called as **sprint**.
- Estimated work remaining is updated daily
- Any team member can add, delete change sprint backlog

# The Agile - Scrum Framework

Inputs from Executives,  
Team, Stakeholders,  
Customers, Users



Product Owner



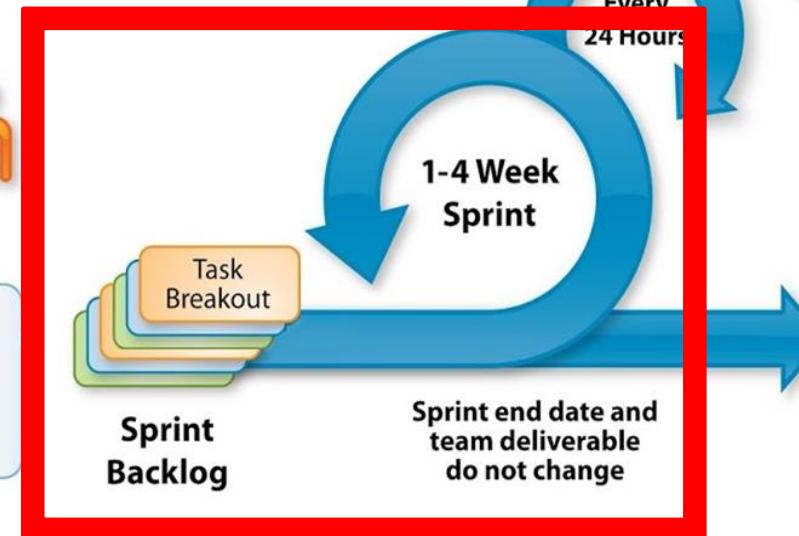
Product  
Backlog



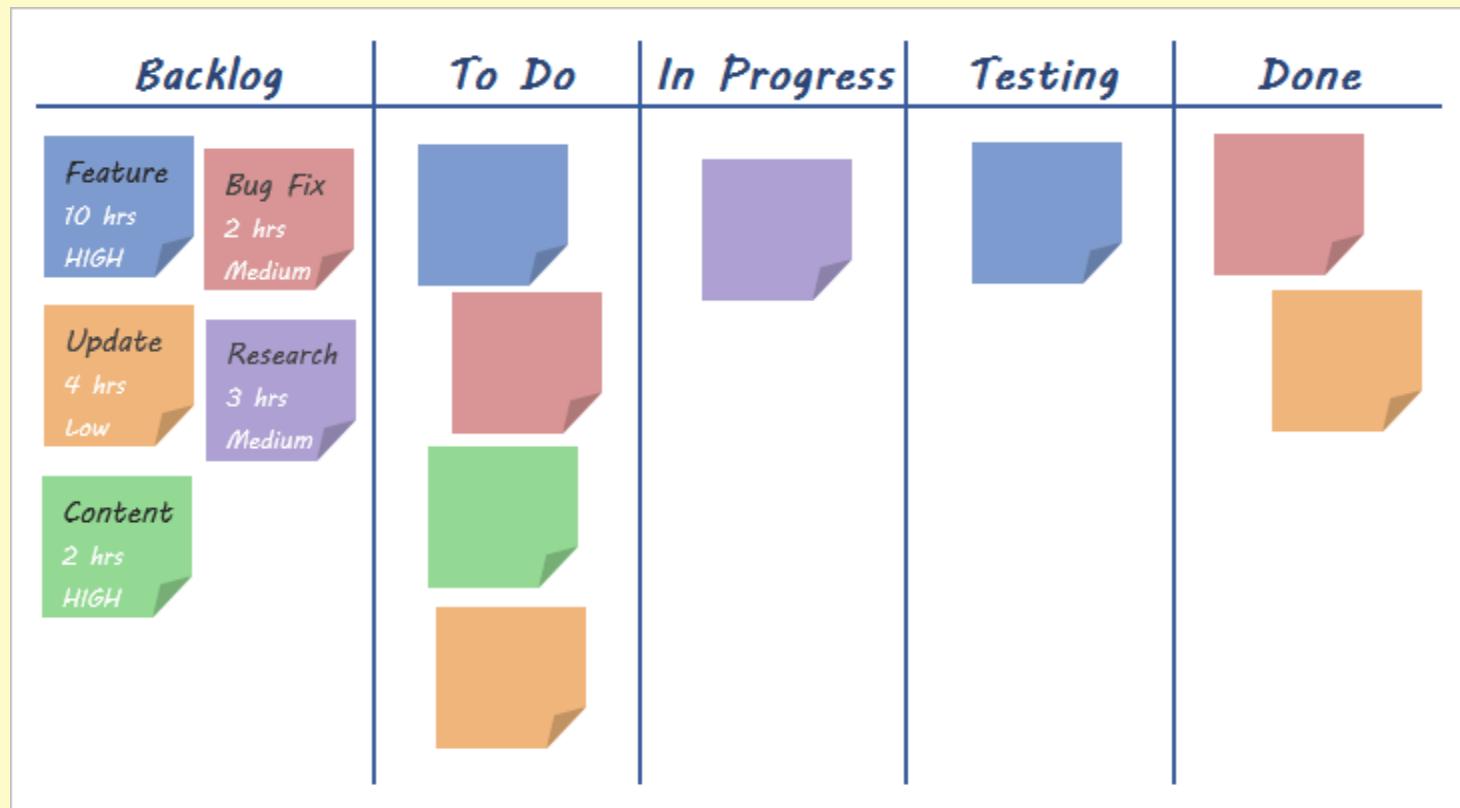
The Team

Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint  
Planning  
Meeting



# Artifact - Sprint Board



# The Agile - Scrum Framework

Inputs from Executives,  
Team, Stakeholders,  
Customers, Users



Product Owner

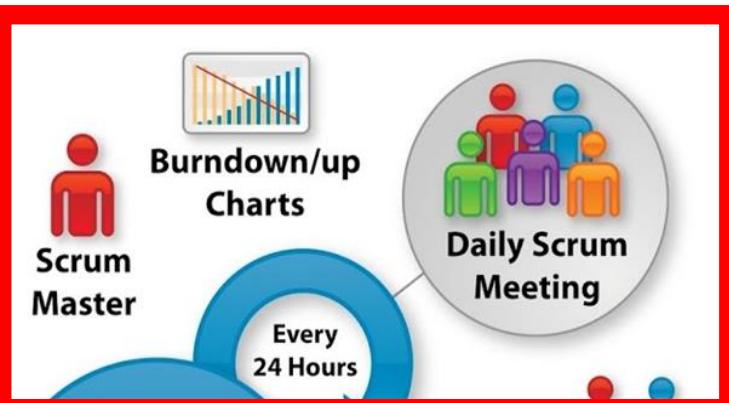
The Team



Product Backlog

Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint Planning Meeting



1-4 Week Sprint

Sprint end date and team deliverable do not change



Finished Work



Sprint Retrospective

# Ceremony - Daily Scrum Meeting

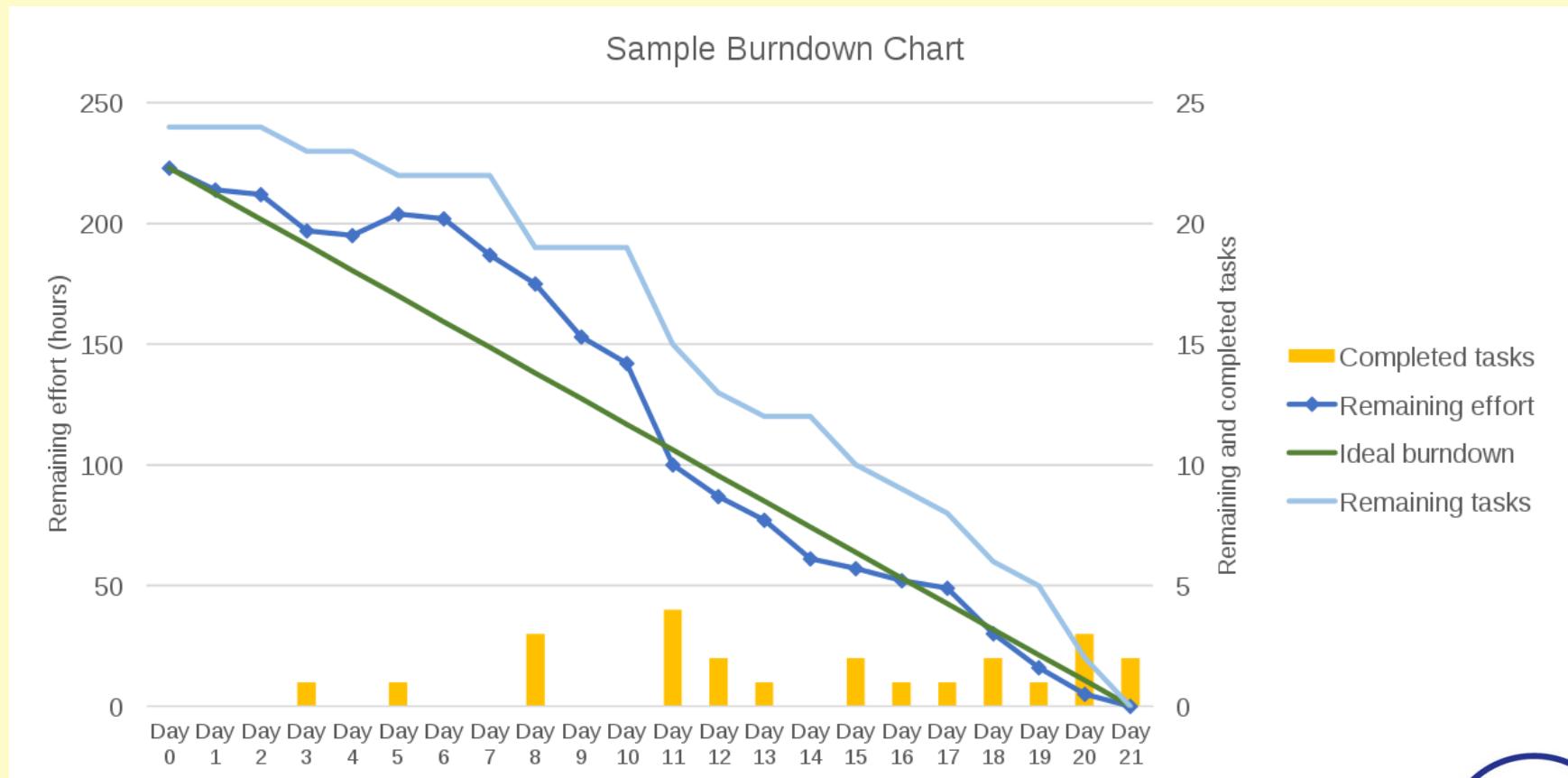
- Parameters
  - Daily, ~15 minutes, Stand-up
  - Anyone late pays a \$1 fee
- Not for problem solving
  - Whole world is invited
  - Only team members, Scrum Master, product owner, can talk
  - Helps avoid other unnecessary meetings
- Three questions answered by each team member:
  1. What did you do yesterday?
  2. What will you do today?
  3. What obstacles are in your way?



# Artifact – Burndown Chart

- A display of what work has been completed and what is left to complete
  - one for each developer or work item
  - updated every day
  - (make best guess about hours/points completed each day)
- *variation:* Release burndown chart
  - shows overall progress
  - updated at end of each sprint

# Artifact – Burndown Chart



# The Agile - Scrum Framework

Inputs from Executives,  
Team, Stakeholders,  
Customers, Users



Product Backlog



Sprint Planning Meeting



Sprint end date and team deliverable do not change



# Ceremony - Sprint Review

- Team presents what it accomplished during the sprint
- Typically takes the form of a demo of new features or underlying architecture
- Informal
  - 2-hour prep time rule
  - No slides
- Whole team participates
- Invite the world



# Ceremony – Sprint Retrospective

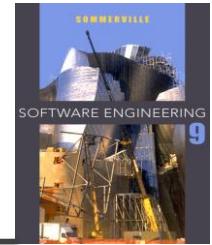
- Identify the scopes of improvement for better result in next sprints
- What worked well, what went wrong are also discussed



# Summary

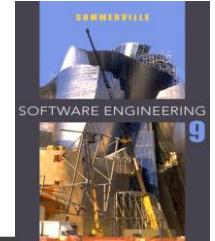
- We have learnt three types of agile principles used in the industry.





---

# Chapter 4 – Requirements Engineering



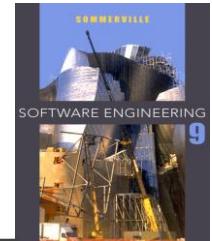
# Topics covered

---

1. What is requirements
2. Types of requirements
3. Requirements engineering processes

The overall requirements engineering process includes four high-level requirements engineering sub-process

- Requirements elicitation
- Requirements analysis
- Requirements validation
- Requirements management

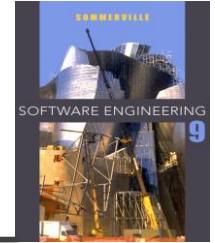


## User Requirement Definition

- 1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.**

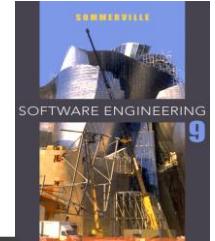
## System Requirements Specification

- 1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.**
- 1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.**
- 1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.**
- 1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.**
- 1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.**



# What is a requirements?

- ✧ It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- ✧ This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract – therefore must be open to interpretation;
  - May be the basis for the contract itself – therefore must be defined in detail;
  - Both these statements may be called requirements.



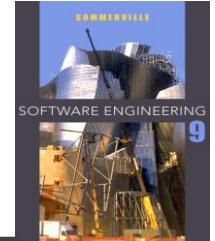
# Functional and non-functional requirements

## ✧ Functional requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should not do.

## ✧ Non-functional requirements

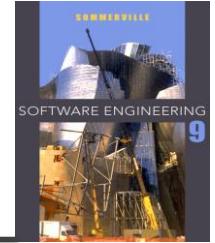
- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.



# Functional requirements

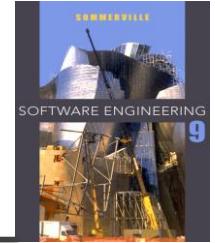
---

- ✧ Describe **functionality or system services**.
- ✧ Depend on the type of software, expected users and the type of system where the software is used.
- ✧ Functional user requirements may be high-level statements of what the system should do.
- ✧ Functional system requirements should describe the system services in detail.



# Non-functional requirements

- ✧ These define system properties and constraints e.g. reliability, response time and storage requirements.  
Constraints are I/O device capability, system representations, etc.
- ✧ Process requirements may also be specified mandating a particular IDE, programming language or development method.
- ✧ Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.



# Non-functional classifications

## ✧ Product requirements

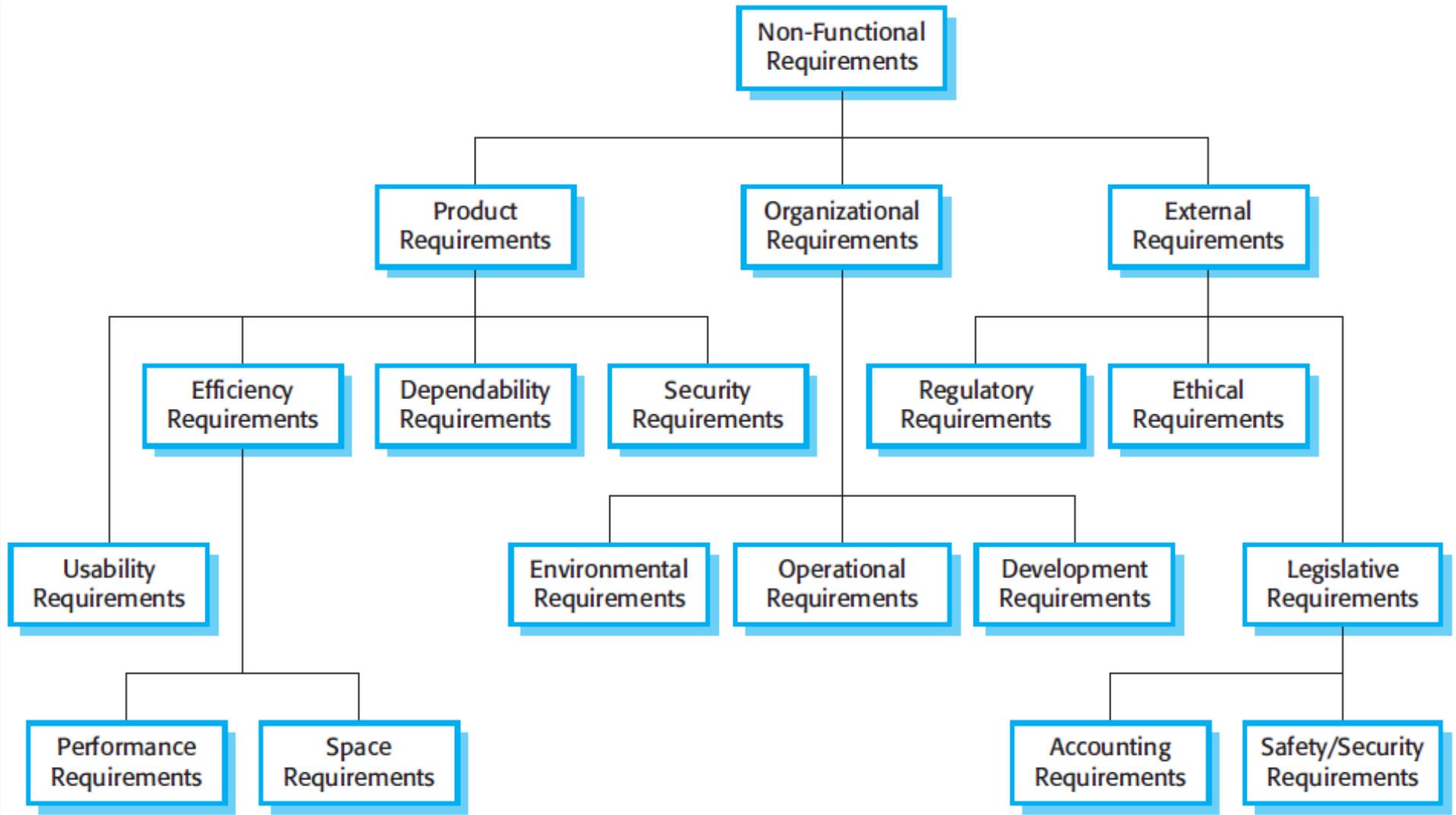
- Requirements which specify that the delivered **product must behave in a particular way** e.g. execution speed, reliability, etc.

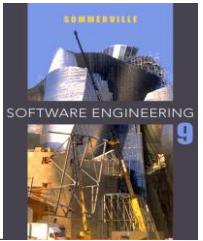
## ✧ Organisational requirements

- Requirements which are a consequence of **organisational policies and procedures** e.g. process standards used, implementation requirements, etc.

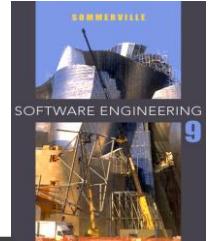
## ✧ External requirements

- Requirements which arise from factors which **are external to the system** and its development process e.g. interoperability requirements, legislative requirements, etc.





Sl.No	Functional Requirement	Non-functional Requirement
.1	Defines all the services or functions required by the customer that must be provided by the system	.Defines system properties and constraints e.g .reliability, response time and storage requirements Constraints are I/O device capability, system representations, etc
.2	.It describes what the software should do	It does not describe what the software will do, but .how the software will do it
.3	Related to business. For example: Calculation of order value by Sales Department or gross pay by the Payroll Department	.Related to improving the performance of the business For example: checking the level of security. An operator should be allowed to view only my name and .personal identification code
.4	.Functional requirement are easy to test	Nonfunctional requirements are difficult to test
.5	Related to the individual system features	Related to the system as a whole
.6	Failure to meet the individual functional requirement may degrade the system	Failure to meet a non-functional requirement may .make the whole system unusable



# Examples of nonfunctional requirements in the MHC-PMS

## Functional requirement

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

## Product requirement

The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

## Organizational requirement

Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

## External requirement

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

### **3. Functional requirements**

ARENA supports five types of users:

- The *operator* should be able to define new games, define new tournament styles (e.g., knock-out tournaments, championships, best of series), define new expert rating formulas, and manage users.
- *League owners* should be able to define a new league, organize and announce new tournaments within a league, conduct a tournament, and declare a winner.
- *Players* should be able to register in an arena, apply for a league, play the matches that are assigned to the player, or drop out of the tournament.
- *Spectators* should be able to monitor any match in progress and check scores and statistics of past matches and players. Spectators do not need to register in an arena.
- The *advertiser* should be able to upload new advertisements, select an advertisement scheme (e.g., tournament sponsor, league sponsor), check balance due, and cancel advertisements.

### **4. Nonfunctional requirements**

- *Low operating cost.* The operator must be able to install and administer an arena without purchasing additional software components and without the help of a full-time system administrator.
- *Extensibility.* The operator must be able to add new games, new tournament styles, and new expert rating formulas. Such additions may require the system to be temporarily shut down and new modules (e.g., Java classes) to be added to the system. However, no modifications of the existing system should be required.
- *Scalability.* The system must support the kick-off of many parallel tournaments (e.g., 10), each involving up to 64 players and several hundreds of simultaneous spectators.
- *Low-bandwidth network.* Players should be able to play matches via a 56K analog modem or faster.

## **VOIP (conference call Project)**

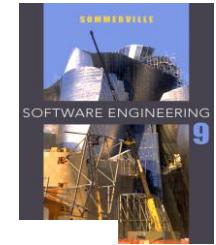
### **Functional Requirements**

- Add Participants
- Drop Participants
- Count Participants
- Mute Microphone
- Invite/prompt Operator

### **Non-Functional Requirements**

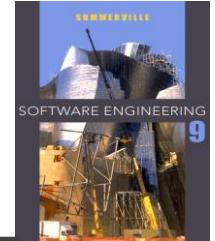
- Voice/ Video quality
- Reliability
- Availability
- Ease of use
- Cost
- Localization

# Metrics for specifying non-functional requirements



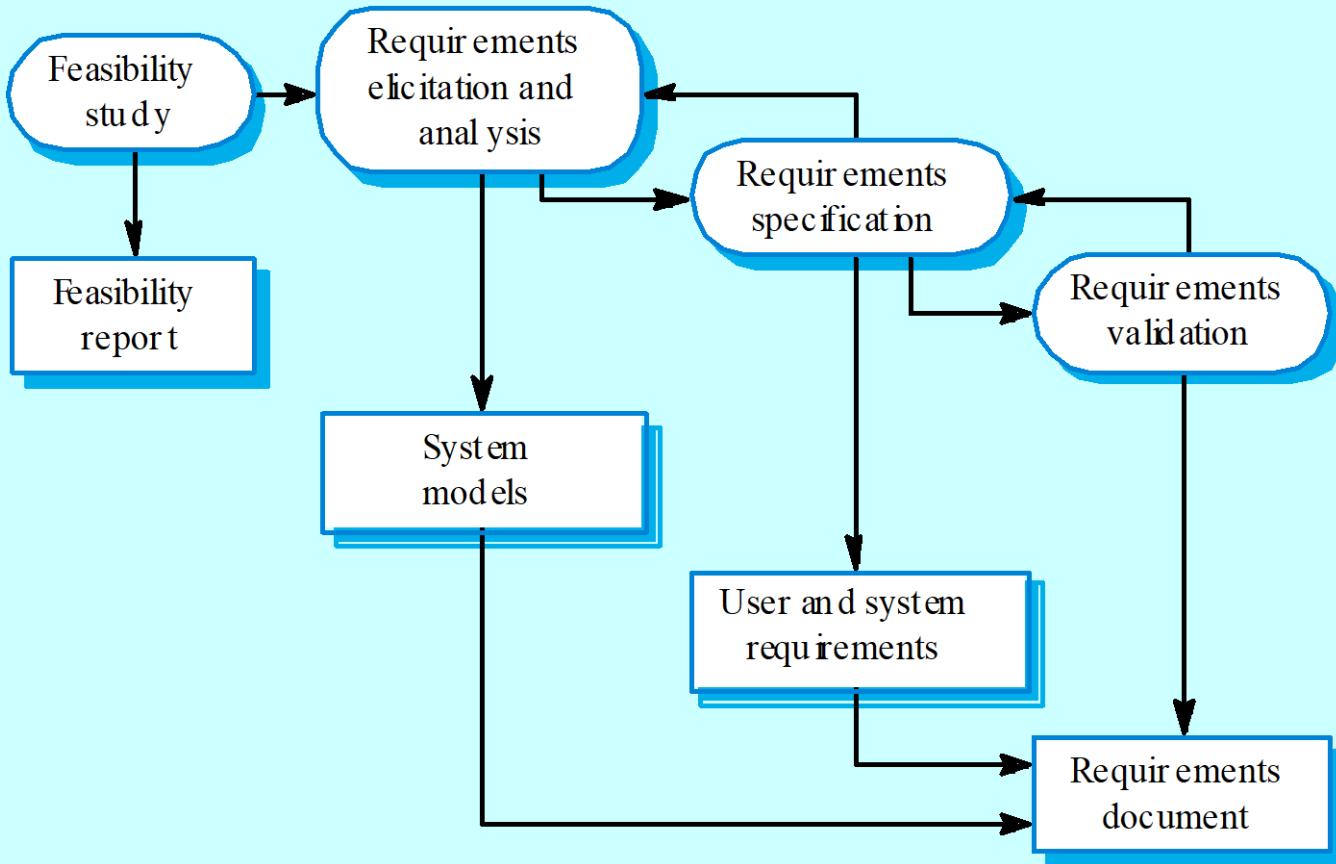
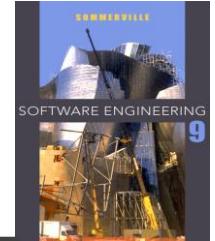
Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

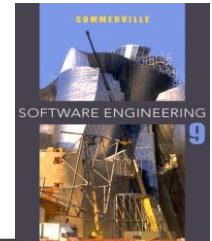
# Requirements engineering processes



- ✧ The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- ✧ However, there are a number of generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.

# The requirements engineering process

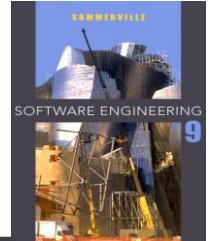




# Feasibility studies

---

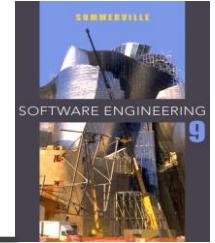
- ✧ A feasibility study decides whether or not the proposed system is worthwhile.
- ✧ A short focused study that checks
  - If the system contributes to organisational objectives;
  - If the system can be engineered using current technology and within budget;
  - If the system can be integrated with other systems that are already used.



# Feasibility study implementation

---

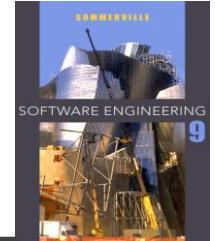
- ✧ Based on information assessment (what is required), information collection and report writing.
- ✧ Questions for people in the organisation
  - What if the system wasn't implemented?
  - What are current process problems?
  - How will the proposed system help?
  - What will be the integration problems?
  - Is new technology needed? What skills?
  - What facilities must be supported by the proposed system?



# Elicitation and analysis

---

- ✧ Sometimes called requirements elicitation or requirements discovery.
- ✧ Involves ***technical staff working with customers*** to find out about the ***application domain***, the ***services*** that the system should provide and the system's ***operational constraints***.
- ✧ May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called ***stakeholders***.

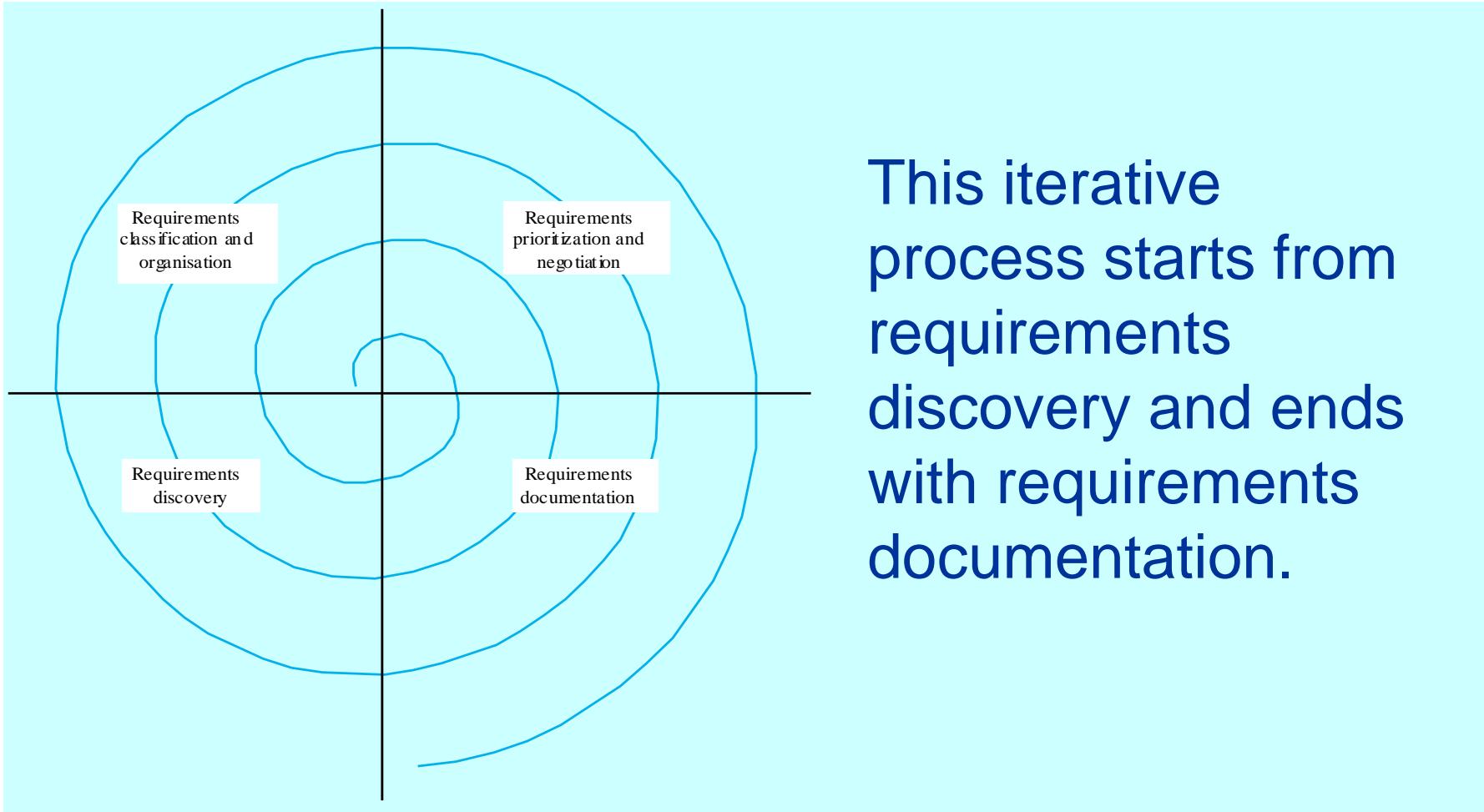
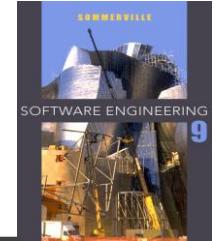


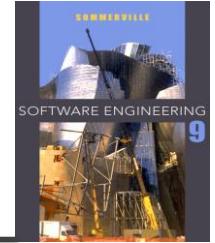
# Problems of requirements analysis

---

- ✧ Stakeholders don't know what they really want.
- ✧ Stakeholders express requirements in their own terms.
- ✧ Different stakeholders may have conflicting requirements.
- ✧ Organisational and political factors may influence the system requirements.
- ✧ The requirements change during the analysis process.  
New stakeholders may emerge and the business environment change.

# The requirements spiral for elicitation and analysis process





# Process activities

---

## ✧ **Requirements discovery**

- Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

## ✧ **Requirements classification and organisation**

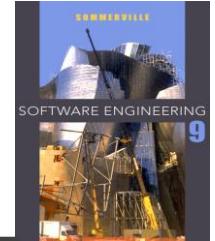
- Groups related requirements and organises them into coherent clusters.
  - *For example, it is possible to use model of the system architecture to identify subsystem and grouping related requirements.*

## ✧ **Prioritisation and negotiation**

- Prioritising requirements and resolving requirements conflicts.
  - *For example it is possible to organize stakeholders negotiations so that compromises can be reached.*

## ✧ **Requirements documentation**

- Requirements are documented and input into the next round of the spiral. The documentation can be formal or informal.
  - *For example, extreme programming uses formal documents and cards and it is very easy for stakeholders to handle, change and organise requirements.*



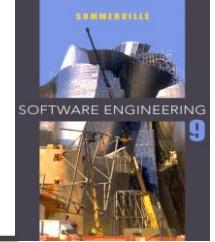
# Requirements discovery

---

- ✧ The process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- ✧ Sources of information include documentation, system stakeholders and the specifications of similar systems.

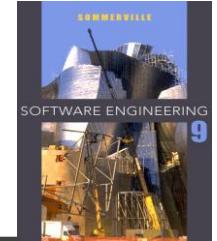
[http://www.navodayaengg.in/wp-content/uploads/2015/10/Lect6\\_Requirements-Discovery.pdf](http://www.navodayaengg.in/wp-content/uploads/2015/10/Lect6_Requirements-Discovery.pdf)

# Requirements discovery and viewpoints

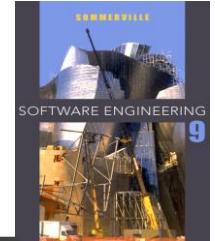


- ✧ Viewpoints are a way of structuring the requirements **to represent the perspectives of different stakeholders**. Stakeholders and other sources may be classified under different viewpoints.
- ✧ This multi-perspective analysis is important as there is no single correct way to analyse system requirements.
- ✧ Types of viewpoint
  - Interactor viewpoints (customers and account database)
  - Indirect viewpoints (management and security staff.)
  - Domain viewpoints (standards that have been developed for interbank communications)

# Requirements discovery and interviewing



- ❖ In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- ❖ There are two types of interview
  - Closed interviews where a pre-defined set of questions are answered.
  - Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.

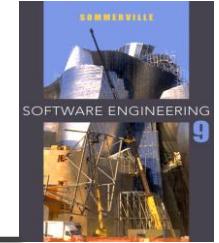


# Interviews in practice

---

- ❖ Normally a mix of closed and open-ended interviewing.
- ❖ Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- ❖ Interviews are **not** good for understanding domain requirements for two reasons:
  - Requirements engineers cannot understand specific domain terminology;
  - Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

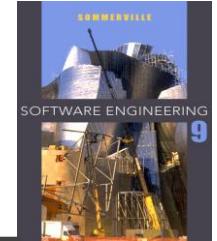
# Requirements discovery and scenarios (1/2)



- ✧ People usually find it **easier** to relate to real-life examples than to abstract description.
- ✧ They can understand and critique a scenario of how they can interact with the system.
- ✧ That is, scenario can be **particularly useful** for adding detail to an outline requirements description:
  - they are description of example interaction sessions;
  - each scenario covers one or more possible interaction;

Several forms of scenarios have been developed, each of which provides different types of information at different level of detail about the system.

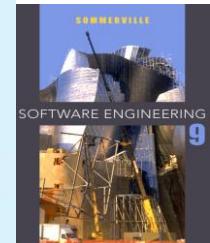
# Requirements discovery and scenarios (2/2)



- ✧ Scenarios are real-life examples of how a system can be used.
- ✧ They should include
  - A description of the starting situation;
  - A description of the normal flow of events;
  - A description of what can go wrong;
  - Information about other concurrent activities;
  - A description of the state when the scenario finishes.
- ✧ Scenarios described as text **might be supplemented** by some kind of diagrams, screen-shot and so on.
- ✧ Alternatively, a more structured approach such as **event scenarios and/or use-case** may be adopted.

**Initial assumption:** The user has logged on to the LIBSYS system and has located the journal containing the copy of the article.

**Normal:** The user selects the article to be copied. The system prompts the user to provide subscriber information for the journal or to indicate a method of payment for the article. Payment can be made by credit card or by quoting an organisational account number.



The user is then asked to fill in a copyright form that maintains details of the transaction and submit it to the LIBSYS system.

The copyright form is checked and, if it is approved, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed. If the article has been flagged as 'print-only' it is deleted from the user's system once the user has confirmed that printing is complete.

**What can go wrong:** The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If the resubmitted form is still incorrect, then the user's request for the article is rejected.

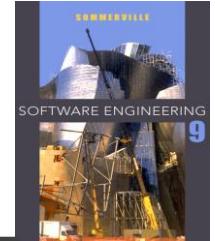
The payment may be rejected by the system, in which case the user's request for the article is rejected.

The article download may fail, causing the system to retry until successful or the user terminates the session.

It may not be possible to print the article. If the article is not flagged as 'print-only' it is held in the LIBSYS workspace. Otherwise, the article is deleted and the user's account credited with the cost of the article.

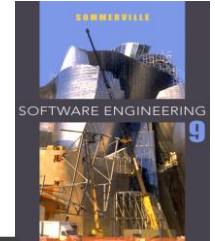
**Other activities:** Simultaneous downloads of other articles.

**System state on completion:** User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.



## Use-cases and scenarios (1/2)

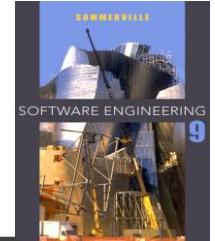
- ✧ Scenarios and use-cases are **effective technique for eliciting** requirements for **interactor** viewpoints. In fact each interaction can be represented as a use-case supplemented by scenarios.
- ✧ They may also be used for **indirect viewpoint** when these viewpoint receive some result. For example consider the Library Manager of our example.



## Use-cases and scenarios (2/2)

---

- ❖ Due to their “*low-level nature*” in this context (*requirements engineering process*), scenarios and use-cases ***are not so effective***
  - for discovering constraints and/or *high-level non-functional* requirements;
  - for discovering *domain* requirements.

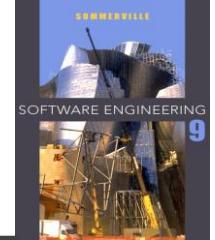


# Requirements validation

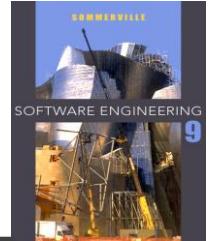
---

- ✧ Concerned with demonstrating that the requirements define the system that the customer really wants.
- ✧ Requirements validation covers a part of analysis in that it is concerned with finding problems with requirements.
- ✧ Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.
  - In fact, a change to the requirements usually means that the system design and the implementation must also be changed and the testing has to be performed again.

# Checks required during the requirements validation process



- ❖ **Validity checks.** Does the system provide the functions which best support the customer's needs? ( Other functions maybe identified by a further analysis )
- ❖ **Consistency checks.** Are there any requirements conflicts?
- ❖ **Completeness checks.** Are all the requirements needed to define all functions required by the customer sufficiently specified?
- ❖ **Realism checks.** Can the requirements be implemented given available budget, technology and schedule?
- ❖ **Verifiability.** Can the requirements be checked?



# Requirements validation techniques

The following techniques can be used *individually* or in *conjunction*.

✧ **Requirements reviews**

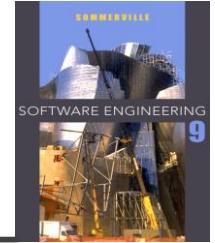
- Systematic manual analysis of the requirements performed by a team of reviewers.

✧ **Prototyping**

- Using an executable model of the system to check requirements.

✧ **Test-case generation**

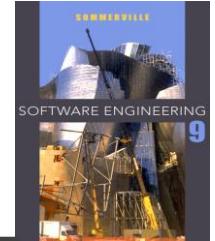
- Developing tests for requirements to check testability.
- If the test is difficult to design, usually the related requirements are difficult to implement.



# Requirements management

---

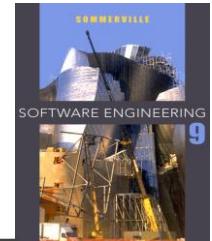
- ✧ The requirements for large systems are frequently changing.
- ✧ In fact, during the software process, the stakeholders' understanding of the problem is constantly changing.
- ✧ Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- ✧ Requirements are inevitably incomplete and inconsistent
  - New requirements emerge during the process as business needs change and a better understanding of the system is developed;
  - Different viewpoints have different requirements and these are often contradictory.



# Requirements management

---

- ✧ **It is hard** for the users and customers **to anticipate** what effects the new system will have on the organization.
  
- ✧ Often, **only when the system has been deployed**, new requirements inevitably emerge.
  
- ✧ This is mainly due to the fact that, **when the end-users have experience of the new system**, they discover new needs and priority.

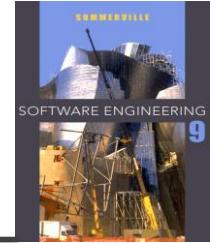


# Requirements change

1. The priority of requirements from different viewpoints changes during the development process. **Conflicts** have to inevitably **converge** in a **compromise**.
2. System customers may specify requirements from a **business perspective** **that conflict with end-user** requirements.
3. The **business and technical** environment of the system **changes** during its development.
4. New hardware, new interface, business priority, new regulations, etc.

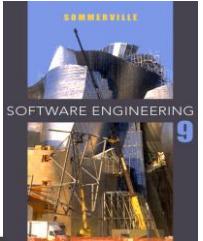
# Requirement changes and the requirements management

---

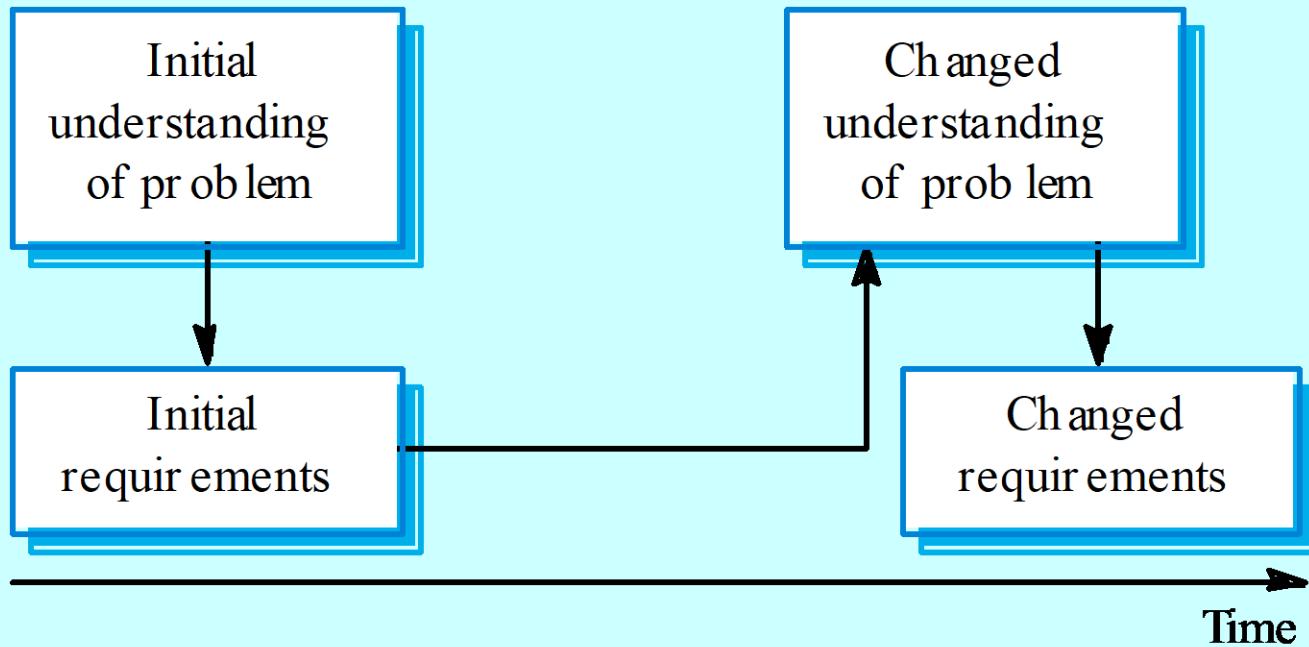


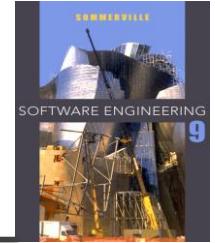
The requirements management is the process of identifying, understanding and controlling changes to system requirements.

- ✧ It might be useful to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.
- ✧ The process of requirements management should start as soon as a draft a version of the requirement document is available.



# Requirements evolution

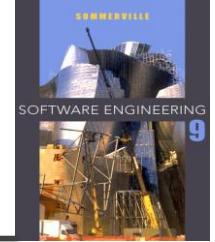




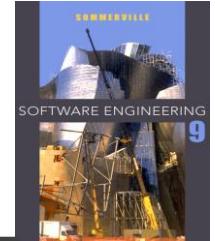
# Enduring and volatile requirements

- ✧ From an **evolution perspective**, requirements fall into two classes:
- ✧ **Enduring requirements.** Stable requirements derived from the **core activity** of the customer organisation and **relate directly to the domain** of the system.
  - E.g., In a hospital, requirements will always relate to doctors, nurses, etc.
  - These requirements may be derived from a **domain conceptual models** that show **entities and relations** between them.
- ✧ **Volatile requirements.** Requirements which **change during development or when the system is in use**.
  - E.g., In a hospital, requirements derived from healthcare policy;

# Requirements management planning

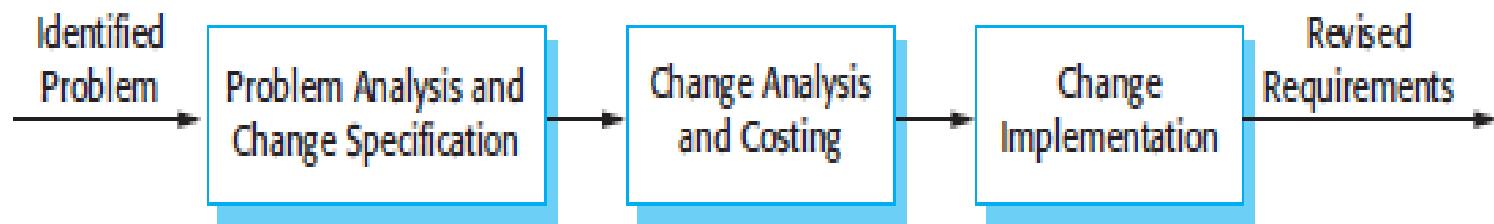


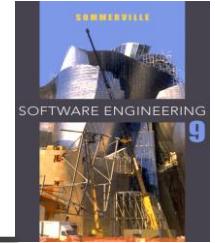
- ✧ Since the RE process is very expensive, it might be useful to establish a planning.
- ✧ In fact, during the requirements engineering process, you have to plan:
  - **Requirements identification**
    - How requirements are individually identified; they should be uniquely identified in order to keep a better traceability.
  - **A change management process**
    - The process followed when requirements change: the ***set of activities*** that estimate the ***impact and cost of changes***.
  - **Traceability policies**
    - ***The policy for managing*** the amount of information about relationships between requirements and between system design and requirements that should be maintained (e.g., in a Data Base)
  - **CASE tool support**
    - The tool support required to help manage requirements change; tools can range from specialist requirements management systems to simple data base systems.



# Change Management Phases

- ✧ *Problem analysis and change specification*
- ✧ *Change analysis and costing*
- ✧ *Change implementation*





# Key points

---

- ✧ The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification and requirements management.
- ✧ Requirements elicitation and analysis is iterative involving domain understanding, requirements collection, classification, structuring, prioritisation and validation.
- ✧ Systems have multiple stakeholders with different requirements.
- ✧ Business changes inevitably lead to changing requirements.
- ✧ Requirements management includes planning and change management.
- ✧ Requirements validation is concerned with checks for validity, consistency, completeness, realism and verifiability.

# SOFTWARE ENGINEERING

CSE 470 – Class Diagram in UML

BRAC University



Inspiring Excellence

# What is a Class?

---

- ▶ A general template that we use to create specific instances or objects in the application domain
  - ▶ Represents a kind of person, place, or thing about which the system will need to capture and store information
  - ▶ Abstractions that specify the attributes and behaviors of a set of objects
-

# What is an Object?

---

- ▶ Entities that encapsulate state and behavior
- ▶ Each object has an identity
  - ▶ It can be referred individually
  - ▶ It is distinguishable from other objects



# Types of Classes

---

- ▶ **Ones found during analysis:**
  - ▶ people, places, events, and things about which the system will capture information
  - ▶ ones found in application domain
- ▶ **Ones found during design**
  - ▶ specific objects like windows and forms that are used to build the system



# Potential Classes

- ▶ *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- ▶ *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- ▶ *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- ▶ *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- ▶ *Organizational units* (e.g., division, group, team) that are relevant to an application.
- ▶ *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- ▶ *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

# 2 Kinds of Classes during Analysis

---

## ▶ Concrete

- ▶ Class from application domain
- ▶ Example: Customer class and Employee class

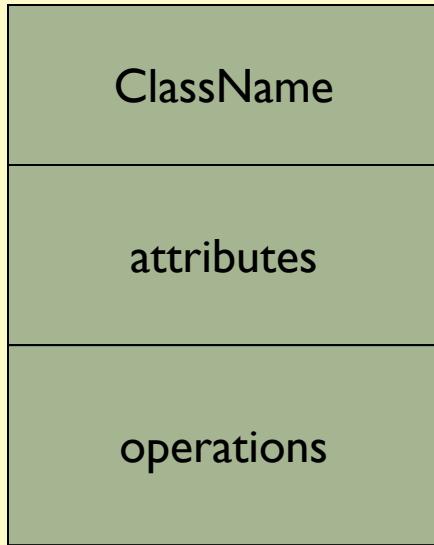
## ▶ Abstract

- ▶ Useful abstractions
- ▶ Example: Person class



# Classes

---



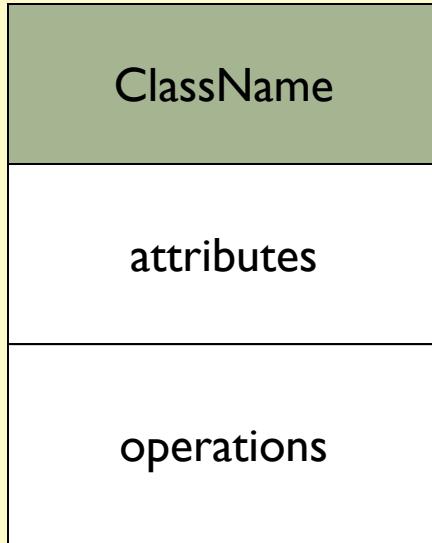
A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.



# Class Names

---



The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.



# Attributes in a Class

---

- ▶ Properties of the class about which we want to capture information
- ▶ Represents a piece of information that is relevant to the description of the class within the application domain



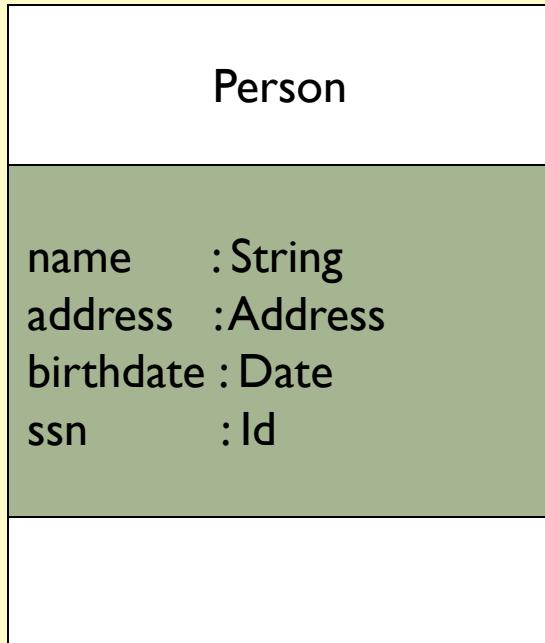
# Attributes in a Class

---

- ▶ Only add attributes that are primitive or atomic types
- ▶ Derived attribute
  - ▶ attributes that are calculated or derived from other attributes
  - ▶ denoted by placing slash (/) before name



# Class Attributes

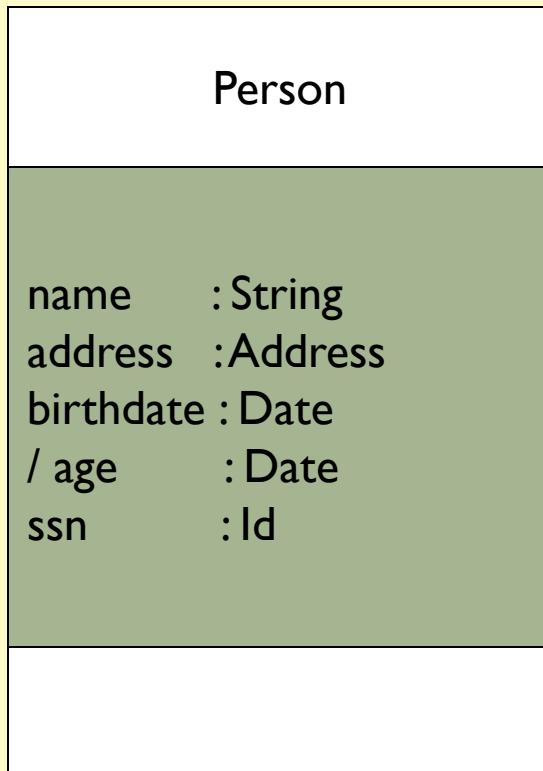


An *attribute* is a named property of a class that describes the object being modeled.

In the class diagram, attributes appear in the second compartment just below the name-compartment.



# Class Attributes (Cont'd)



Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date



# Operations in a Class

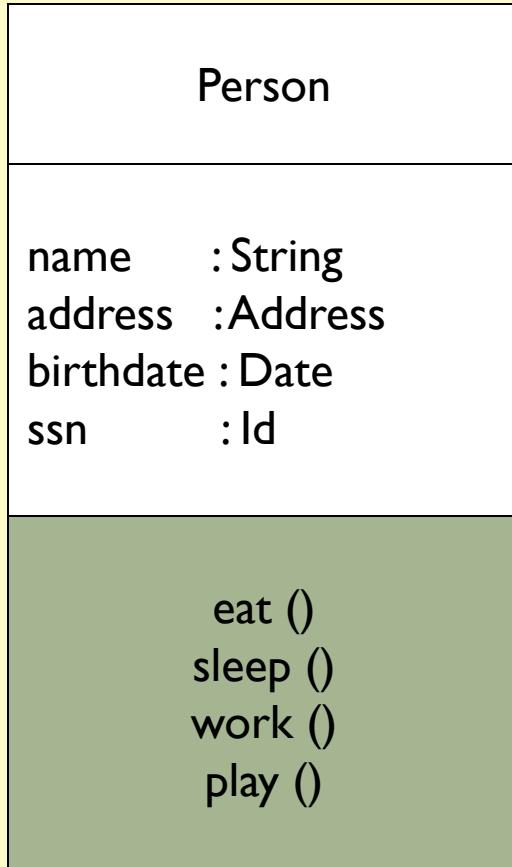
---

- ▶ Represents the actions or functions that a class can perform
- ▶ Describes the actions to which the instances of the class will be capable of responding
- ▶ Can be classified as a constructor, query, or update operation



# Class Operations

---



*Operations describe the class behavior and appear in the third compartment.*



# Class Operations (Cont'd)

---

## PhoneBook

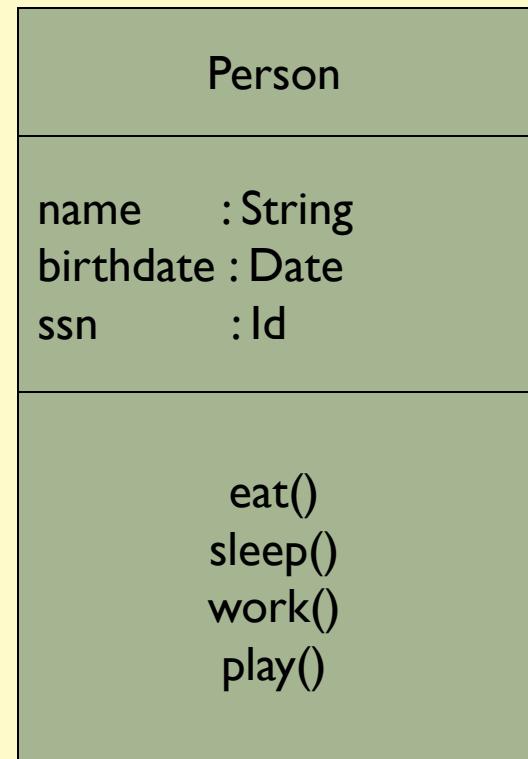
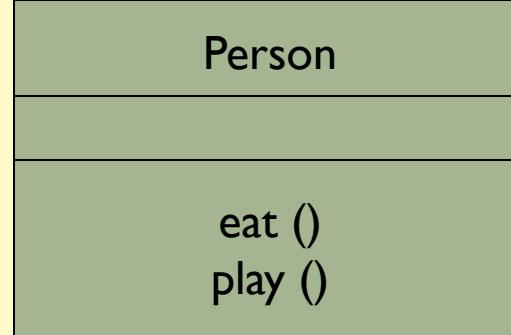
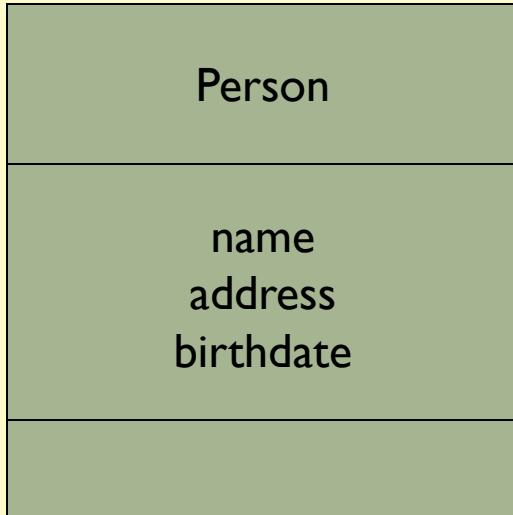
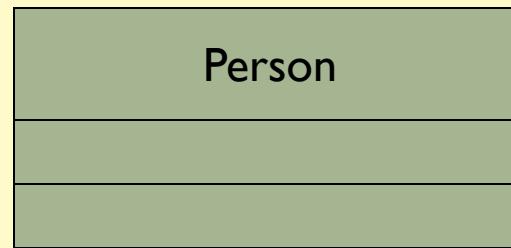
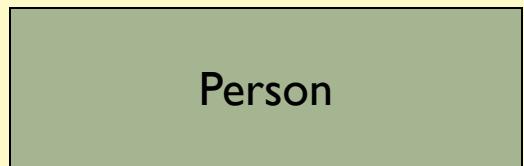
```
newEntry (n : Name, a : Address, p : PhoneNumber, d : Description)  
getPhone ( n : Name, a : Address) : PhoneNumber
```

You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.



# Depicting Classes

When drawing a class, you need not show attributes and operation in every diagram.



# UML Representation of Class

---

Class Name

Attributes of Class

Operations/methods  
of Class



# Visibility of Attributes and Operations

---

- ▶ Relates to the level of information hiding to be enforced

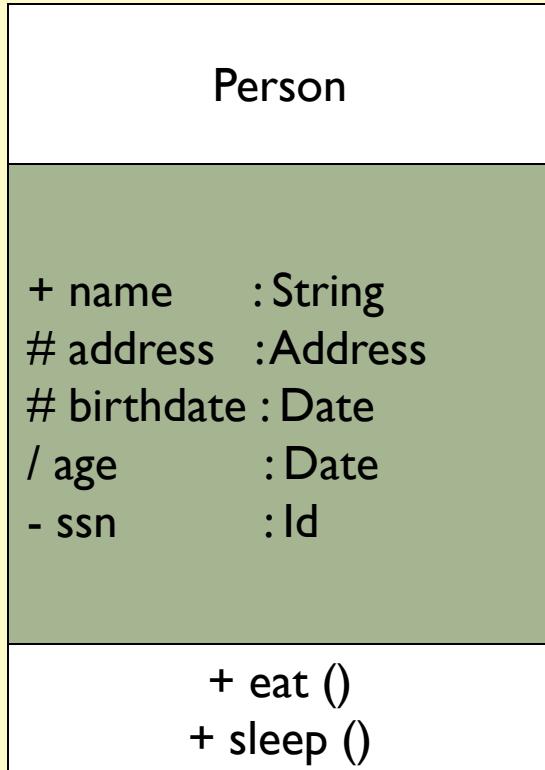


# Visibility of Attributes and Operations

<b>Visibility</b>	<b>Symbol</b>	<b>Accessible To</b>
Public	+	All objects within your system.
Protected	#	Instances of the implementing class and its subclasses.
Private	-	Instances of the implementing class.



# Visibility (Cont'd)



Attributes can be:

- + public
- # protected
- private
- / derived



# Relationships among Classes

---

- ▶ Represents a connection between multiple classes or a class and itself
- ▶ 2 basic categories:
  - ▶ association relationships
    - ▶ Aggregation
    - ▶ Composition
  - ▶ generalization relationships



# Association Relationship

---

- ▶ A bidirectional semantic connection between classes
- ▶ Type:
  - ▶ name of relationship
  - ▶ role that classes play in the relationship



# Association Relationship

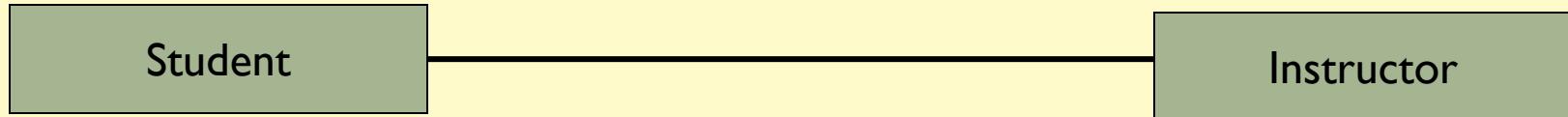
- ▶ Name of relationship type shown by:
  - ▶ drawing line between classes
  - ▶ labeling with the name of the relationship
  - ▶ indicating with a small solid triangle beside the name of the relationship the direction of the association



# Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

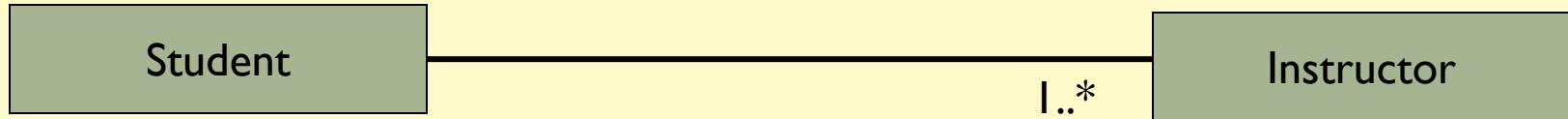
An *association* denotes that link.



# Association Relationships (Cont'd)

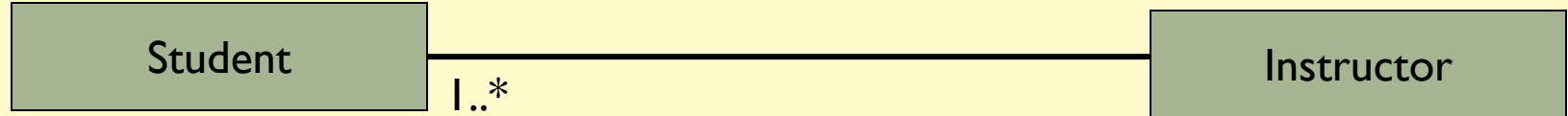
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *Student* has one or more *Instructors*:



# Association Relationships (Cont'd)

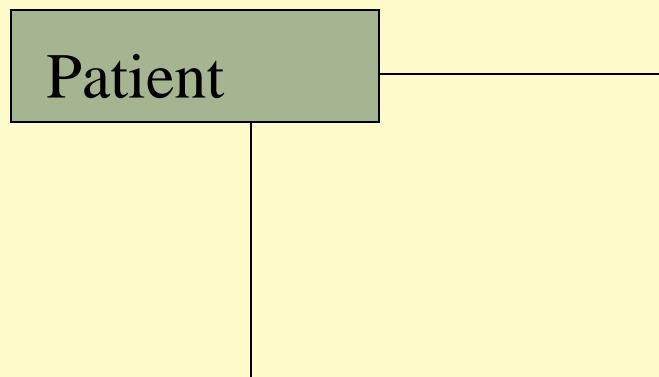
The example indicates that every *Instructor* has one or more *Students*:



# Association Relationship

---

- ▶ Role type shown by:
  - ▶ drawing line between classes
  - ▶ indicating with a plus sign before the role name



+ primary insurance  
carrier



# Aggregation Relationship

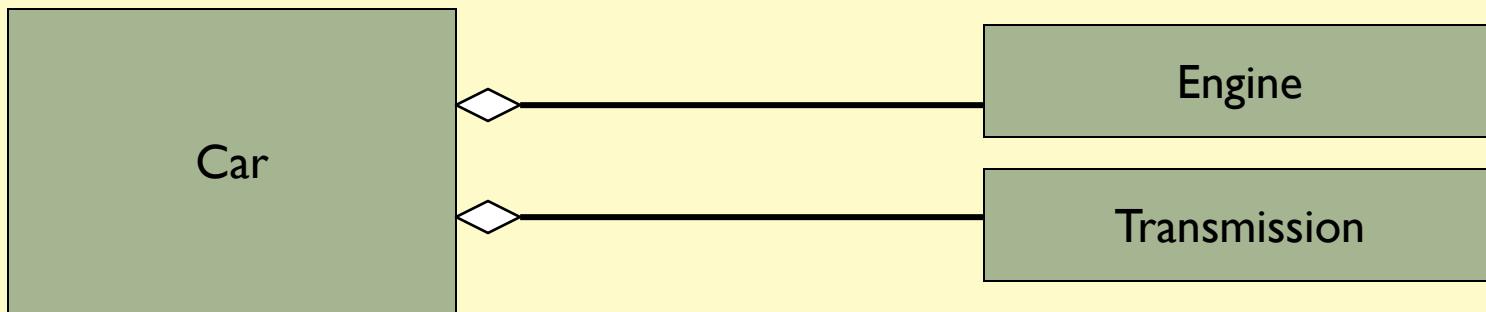
---

- ▶ Specialized form of association in which a whole is related to its part(s)
- ▶ Represented by *a-part-of* relationship
- ▶ Specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate.



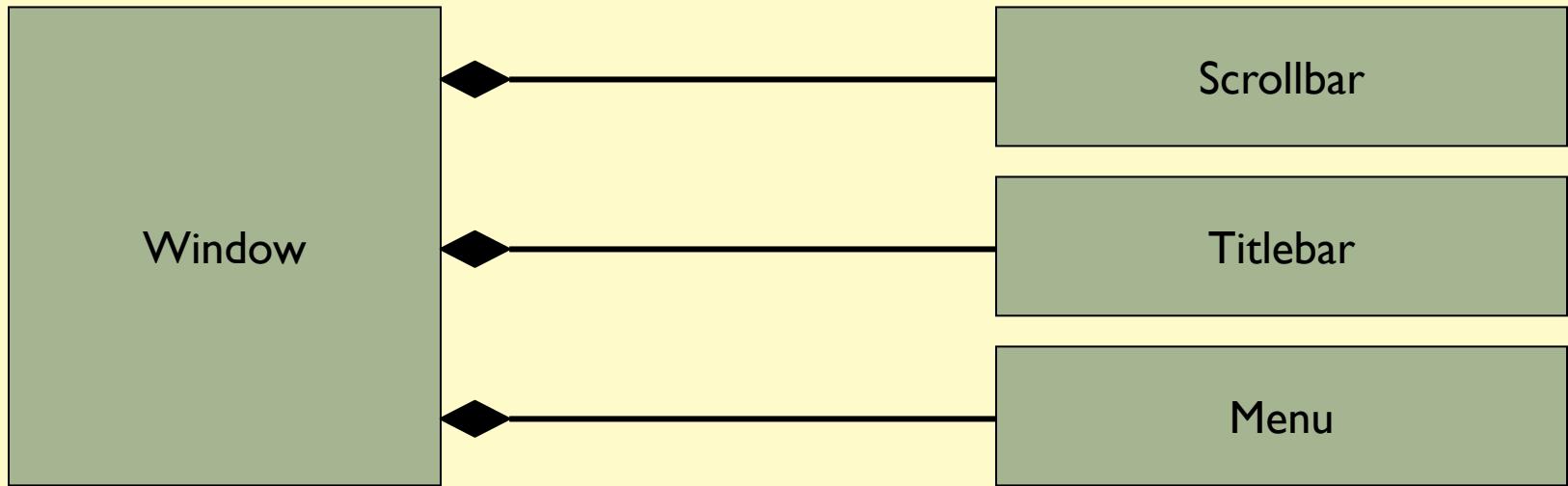
# Aggregation Relationship

- ▶ Aggregations are denoted by a hollow-diamond adornment on the association.



# Composition Relationship

- ▶ A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (i.e., they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association



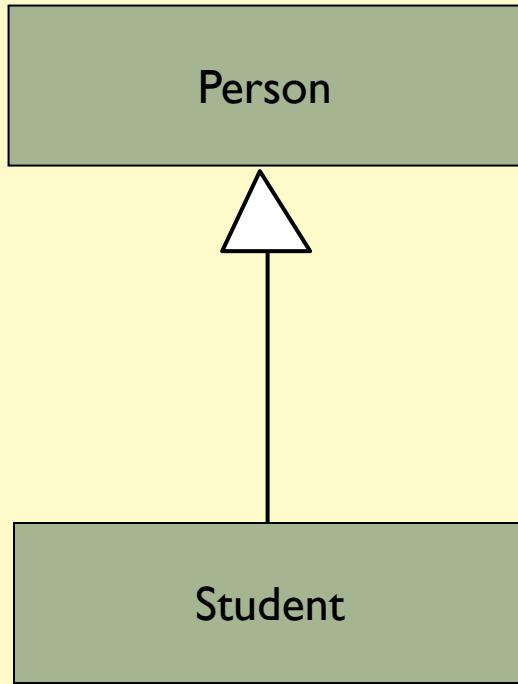
# Generalization Relationship

---

- ▶ Enables the analyst to create classes that inherit attributes and operations of other classes
- ▶ Represented by *a-kind-of* relationship



# Generalization Relationships

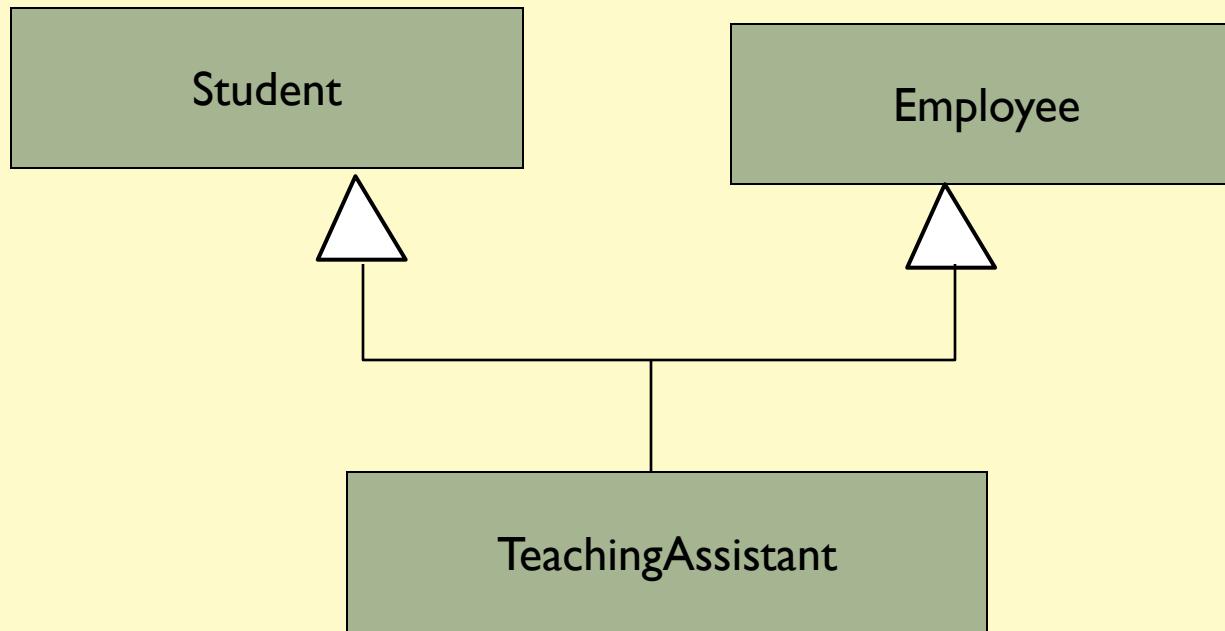


A **generalization** connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

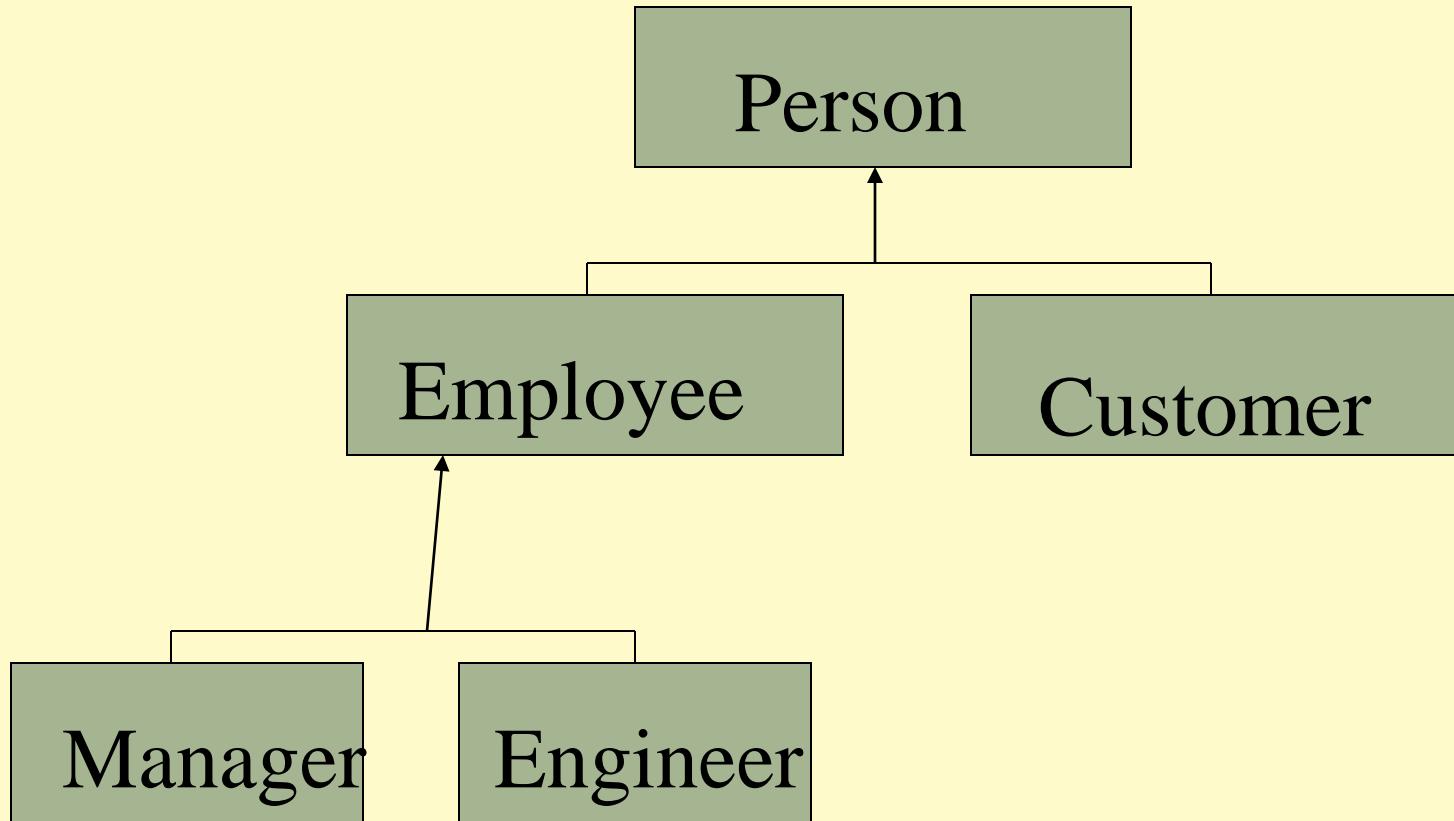


# Generalization Relationships (Cont'd)

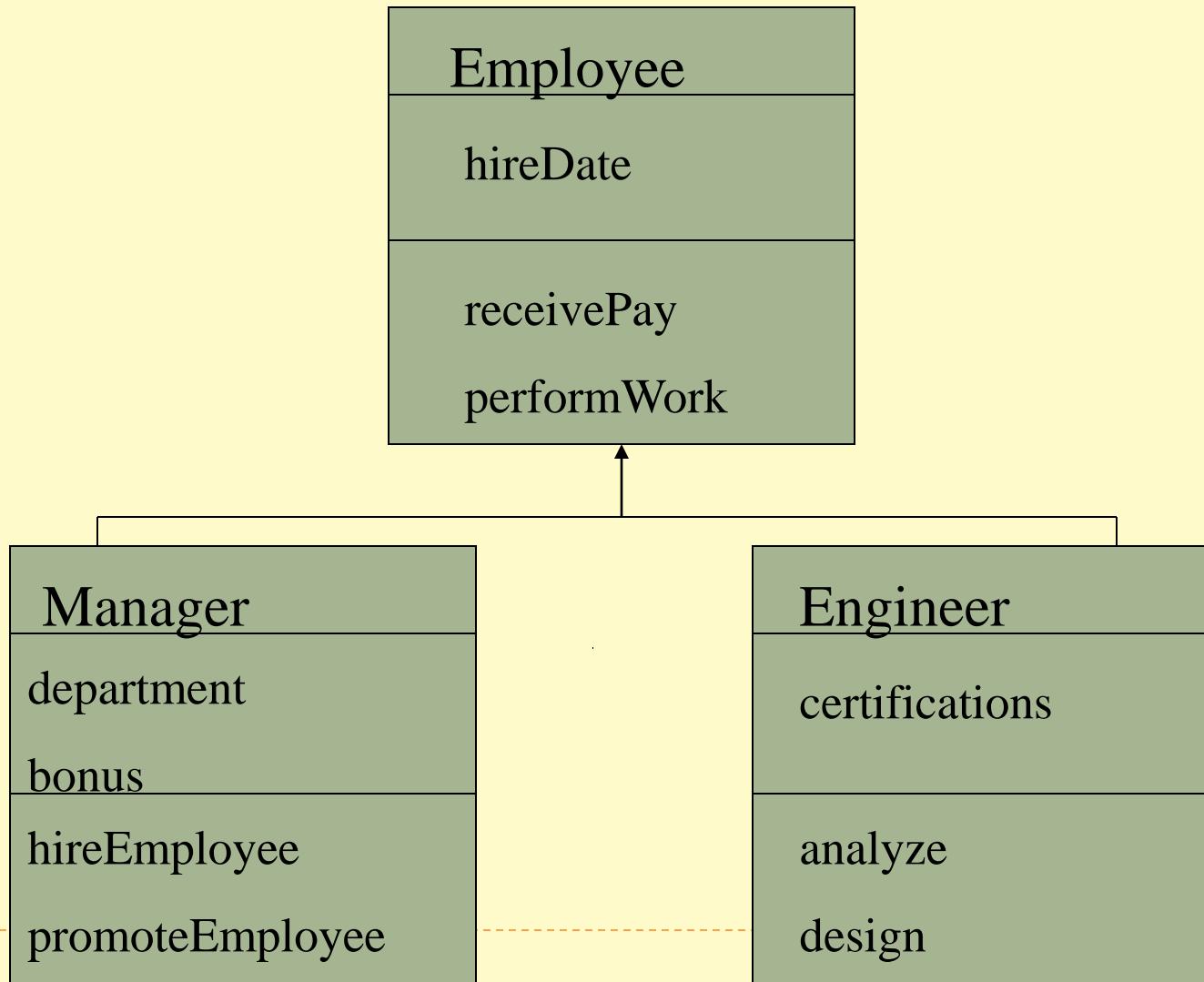
UML permits a class to inherit from multiple superclasses, although some programming languages (e.g., Java) do not permit multiple inheritance.



# Generalization Relationship

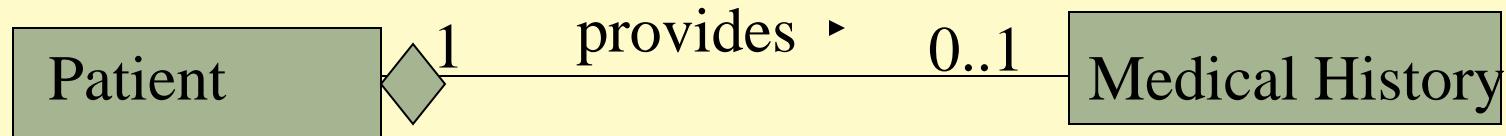


# Generalization Relationship



# Multiplicity

- ▶ Documents how many instances of a class can be associated with one instance of another class



# Multiplicity

---

- ▶ Denotes the **minimum number.. maximum number** of instances

Exactly one                    |

Zero or more                0..\*      or      0..m

One or more                |..\*      or      |..m

Zero or one                0..|

Specified range            2..4

Multiple, disjoint ranges    |..3, 5



# SOFTWARE ENGINEERING

CSE 470 – Class Diagram Design

BRAC University



Inspiring Excellence

# Guidelines for Analyzing Requirements / Use Cases

- ▶ A common or improper noun implies a class of objects
- ▶ A proper noun implies an instance of a class
- ▶ A collective noun implies a class of objects made up of groups of instances of another class



# Guidelines for Analyzing Requirements / Use Cases (2)

---

- ▶ An adjective implies an attribute of an object
- ▶ A doing verb implies an operation
- ▶ A being verb implies a relationship between an object and its class
- ▶ A having verb implies an aggregation or association relationship



# Guidelines for Analyzing Requirements / Use Cases (3)

---

- ▶ A transitive verb implies an operation
- ▶ An intransitive verb implies an exception
- ▶ A predicate or descriptive verb phrase implies an operation
- ▶ An adverb implies an attribute of a relationship or an operation

# Class Diagram

---

- ▶ Ensure that the classes are both necessary and sufficient to solve the underlying problem
  - ▶ no missing attributes or methods in each class
  - ▶ no extra or unused attributes or methods in each class
  - ▶ no missing or extra classes



# Discarding Unnecessary and Incorrect Classes

---

- ▶ Redundant Classes: Some potential classes differ only in name.
  - ▶ Irrelevant Classes: Classes that have nothing to do with the system. Example: *computer connection*
  - ▶ Vague Classes: Classes whose meaning is not clear at all. Examples: *system* and *software*
-

# Discarding Unnecessary and Incorrect Classes

---

- ▶ Attributes: Some nouns in the list above are likely to be modeled as attributes rather than classes. Examples: *author, title*
- ▶ Operations: Some nouns are likely to be operations rather than classes. Example: *book search.*
- ▶ Roles: Some nouns are roles of objects involved in associations rather than classes.
- ▶ Implementation Constructs: Anything that is not part of the real-world problem.



# Types of Classes

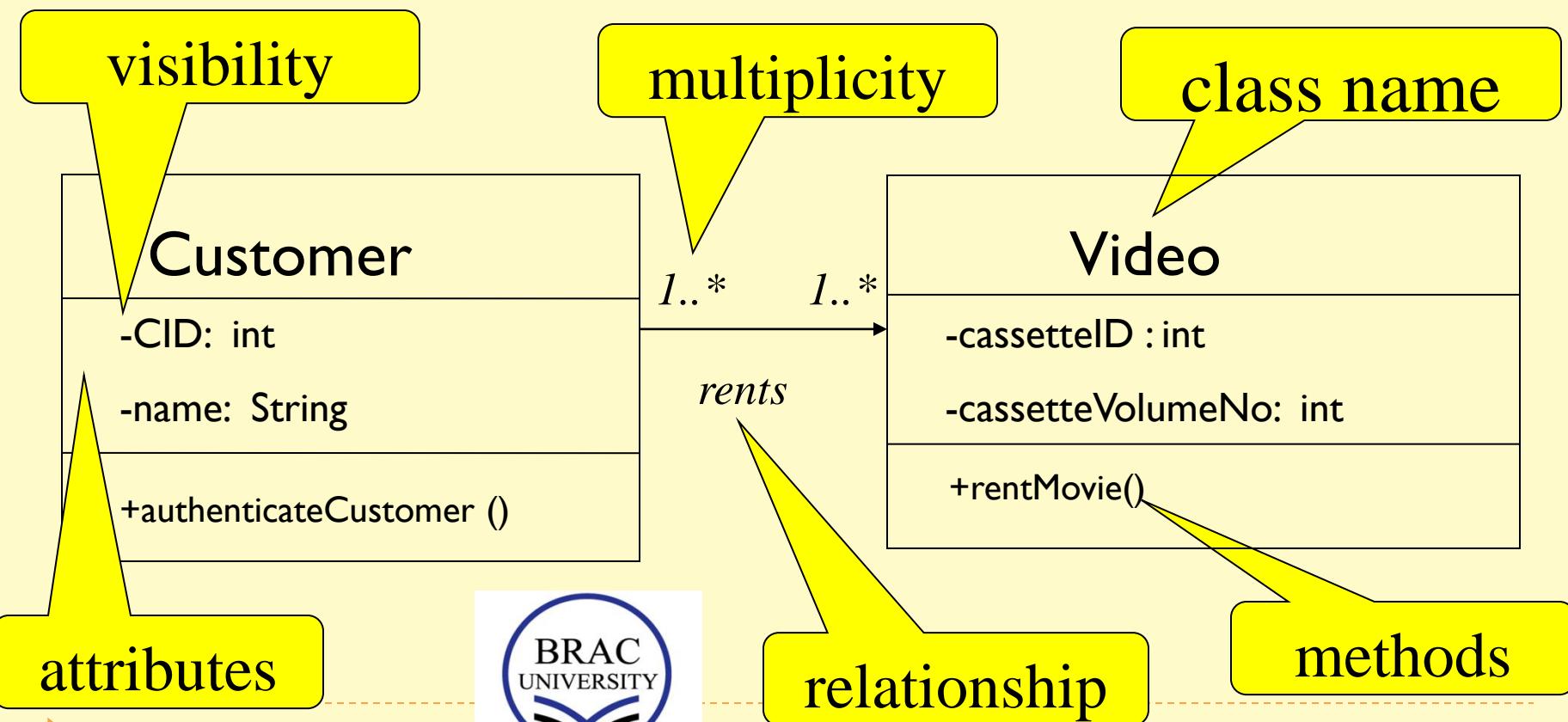
---

- ▶ **Ones found during analysis:**
  - ▶ people, places, events, and things about which the system will capture information
  - ▶ ones found in application domain
- ▶ **Ones found during design**
  - ▶ specific objects like windows and forms that are used to build the system

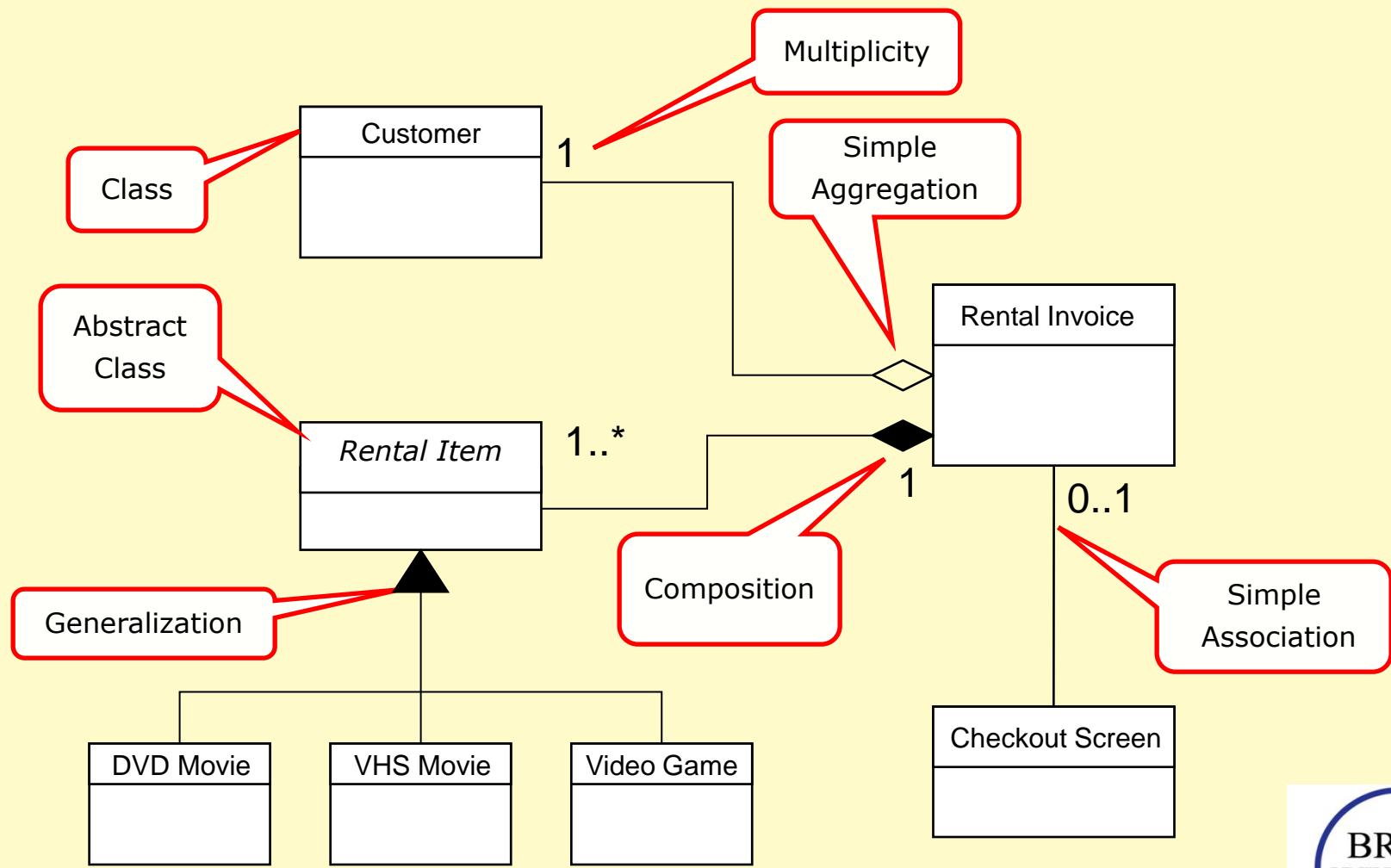


# Example of a Class Diagram

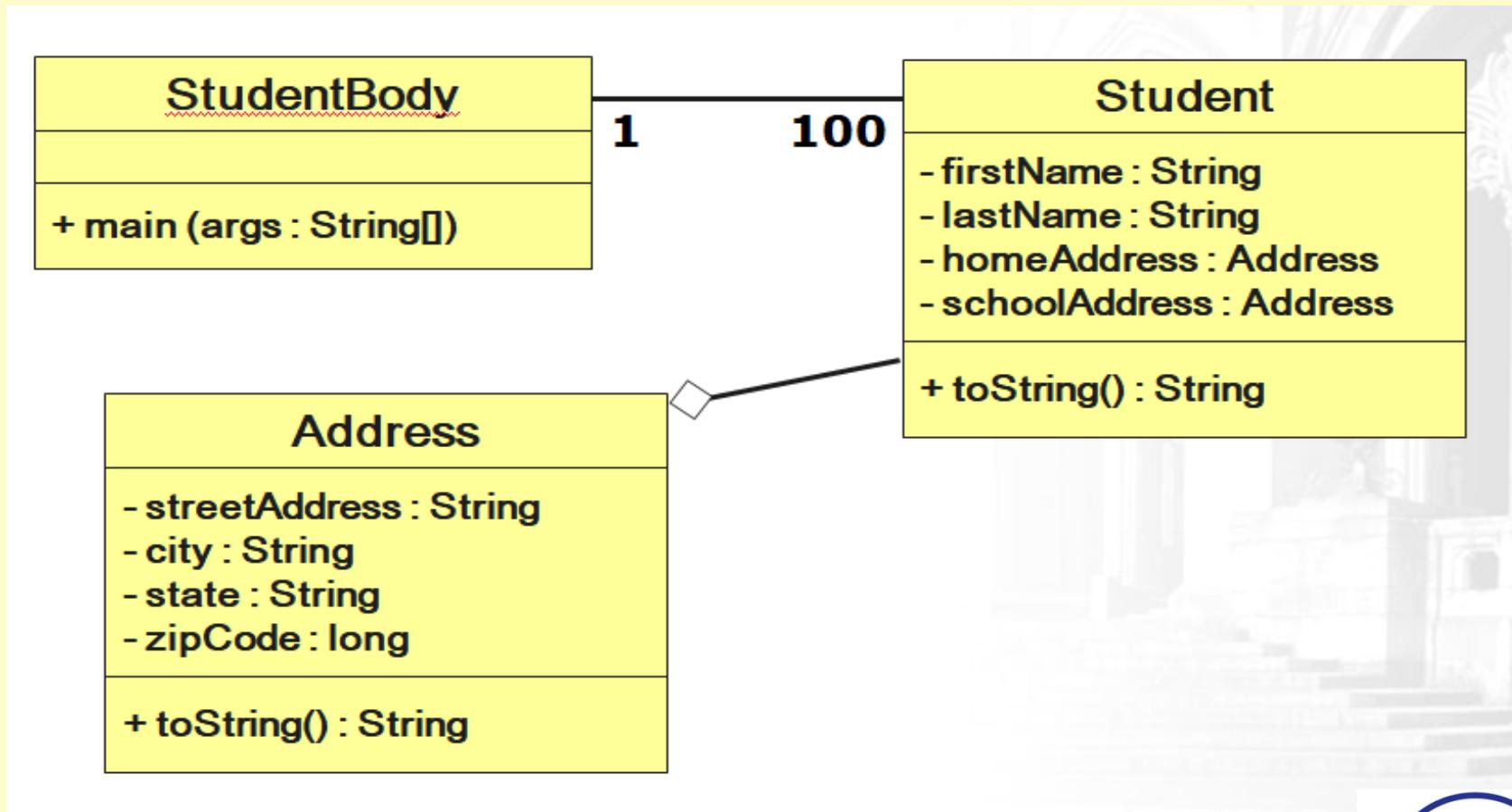
## Video Rental System



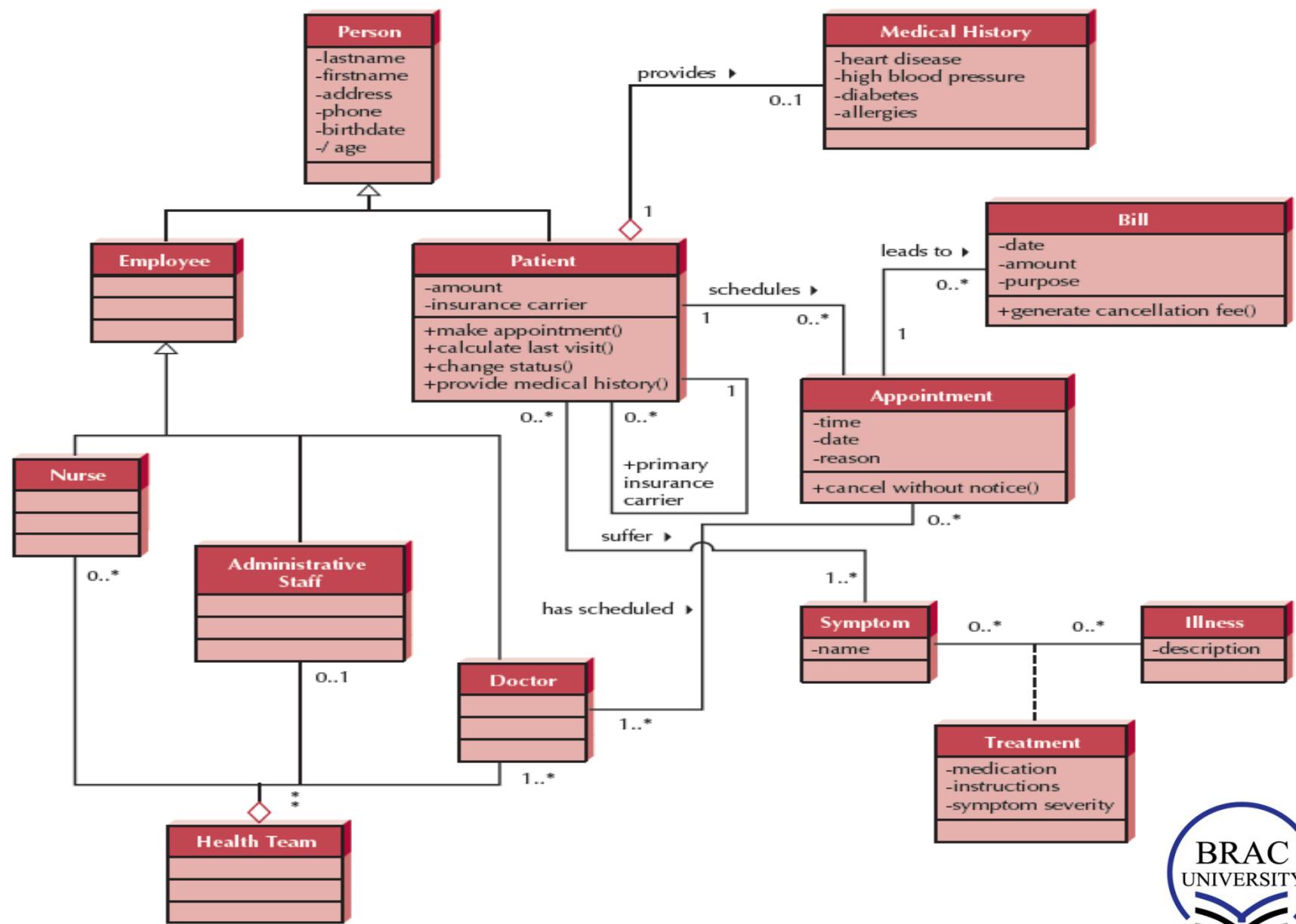
# Class diagram example 1



# Class diagram example 2



# Class Diagram Example 3



# SOFTWARE ENGINEERING

CSE 470 - Software Architecture

BRAC University

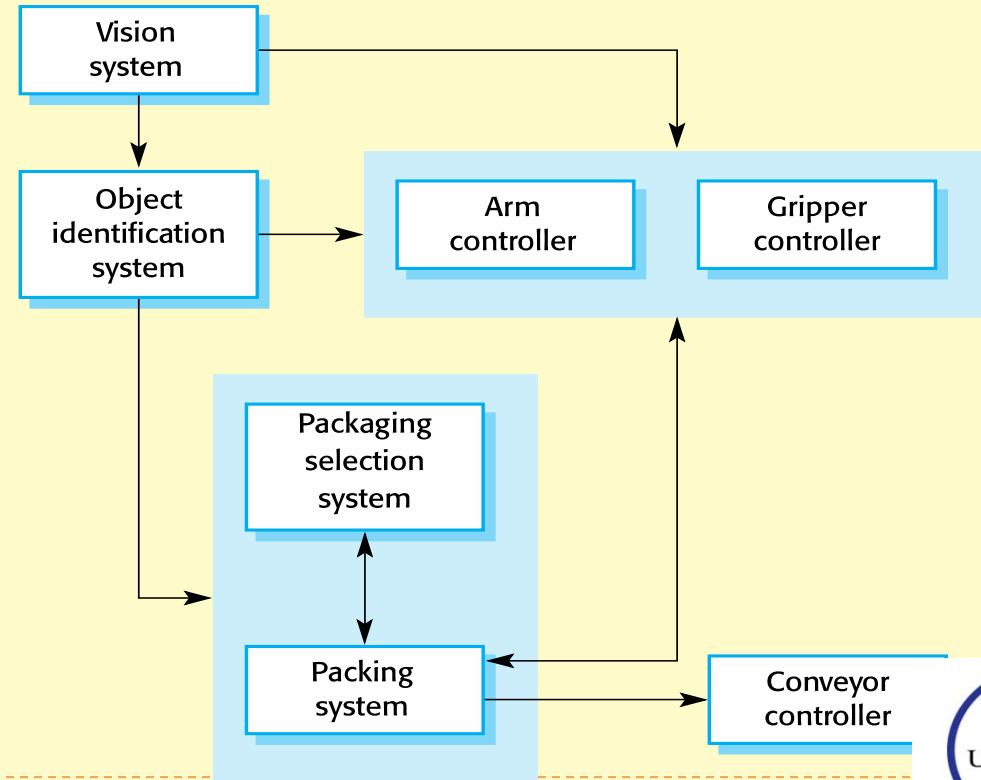
# Software Architecture

- ▶ Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- ▶ Software Architecture is how the defining components of a software system are **organized and assembled**. How they **communicate** each other. And how **the constraints of the whole system is ruled by**.
- ▶ **The architectural model defines the development phases**



# Architectural Representation

- ▶ Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.

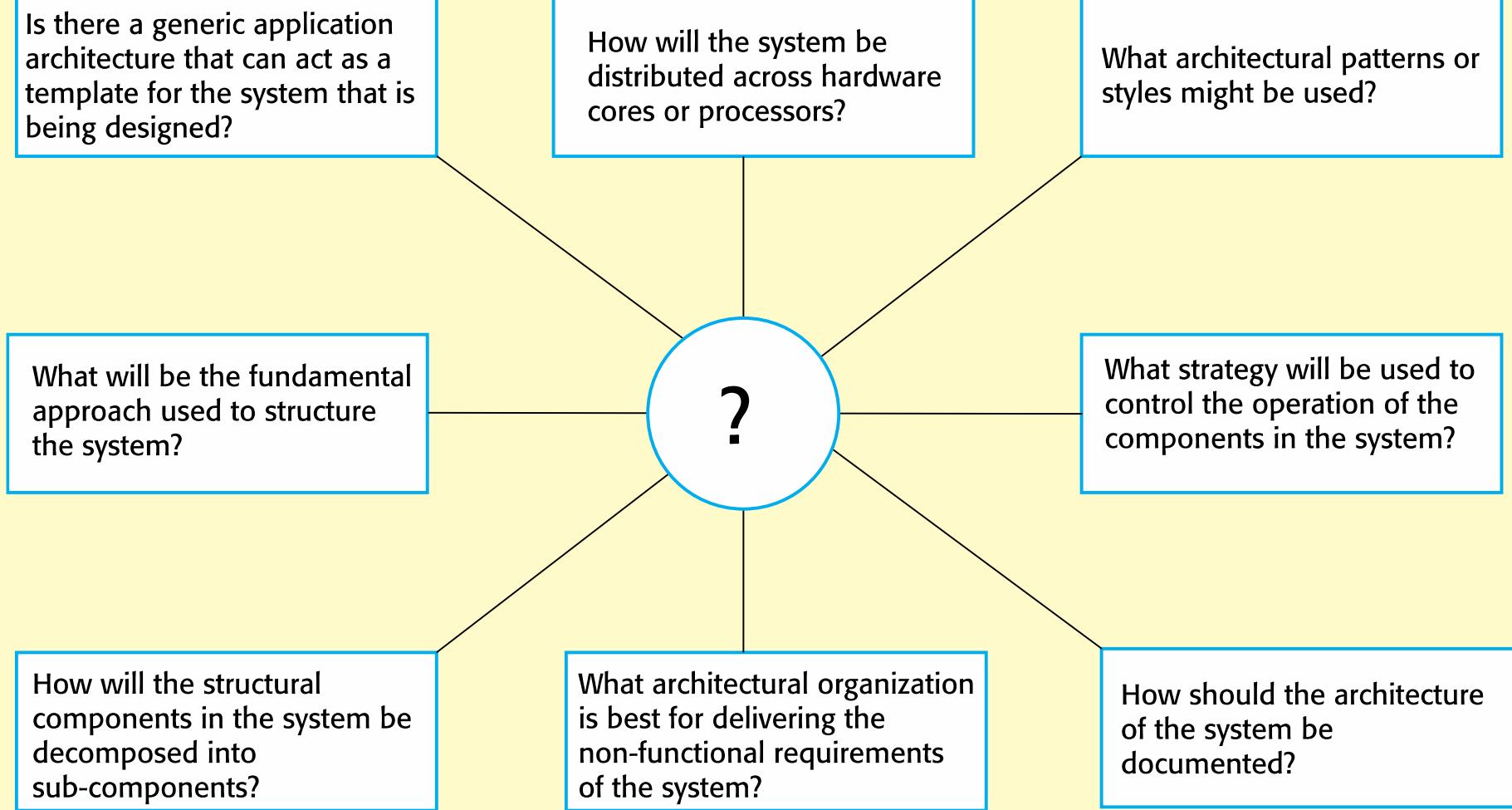


# Advantages of explicit architecture

---

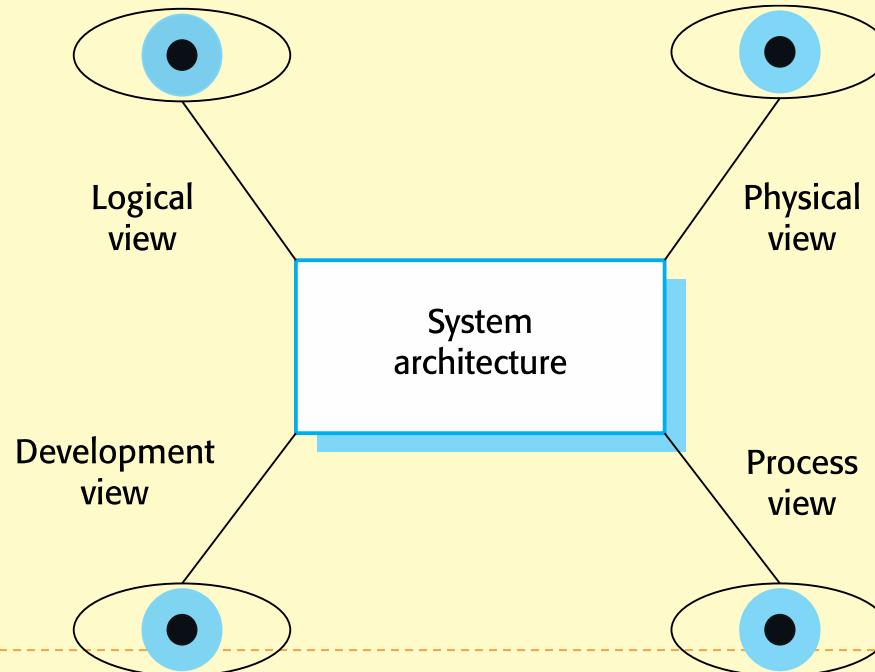
- ▶ Stakeholder communication
  - ▶ Architecture may be used as a focus of discussion by system stakeholders.
- ▶ System analysis
  - ▶ Means that analysis of whether the system can meet its non-functional requirements is possible.
- ▶ Large-scale reuse
  - ▶ The architecture may be reusable across a range of systems

# Architecture Design Decisions



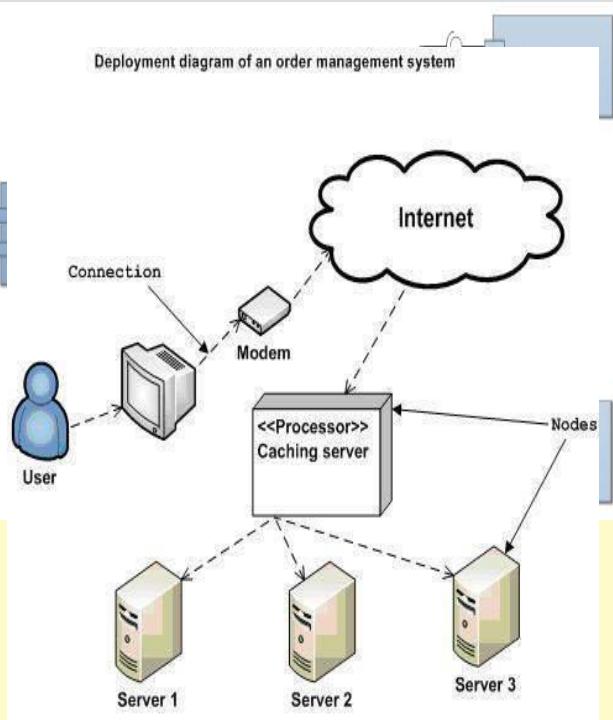
# Architectural View

- ▶ There are four views from which the architecture of a software can be observed
- ▶ Each architectural diagram only shows one view or perspective of the system.



# 4 + 1 view model of software architecture

- ▶ A logical view, which shows the key abstractions in the system as objects or object classes. (class/state diagrams)
- ▶ A process view, which shows how, at run-time, the system is composed of interacting processes. (activity diagram)
- ▶ A development view, which shows how the software is decomposed for development. (Component/package diagram)
- ▶ A physical view, which shows the system hardware and how software components are distributed across the processors in the system. (Deployment diagram.)
- ▶ Related using use cases or scenarios (+1)



# Architectural Pattern

- ▶ Lets think of some problems -



# Architectural Pattern

---

- ▶ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ▶ It's a solution to a existing and commonly occurring problem.
- ▶ Patterns include information about when they are and when they are not useful.

# MVC Pattern

- ▶ MVC goes for Model-View-Controller Pattern
- ▶ Separates presentation and interaction from the data handling logic.
- ▶ The system is structured into three logical components that interact with each other- Model, View and Controller



# Controller

- ▶ Handles all the user inputs through url
- ▶ The interaction to an application starts here by the user interactions – mouse click, key press etc
- ▶ Process http url requests (*GET, POST, PUT, DELETE*)
  - ▶ *GET*: for getting a data
  - ▶ *POST*: for posting / inserting a data
  - ▶ *PUT*: for updating a data
  - ▶ *DELETE*: for removing a data
- ▶ Communicates with both Model and View
- ▶ Contains all server side logic
- ▶ In the example – ProductController, UserController, AccountController etc.

# Model

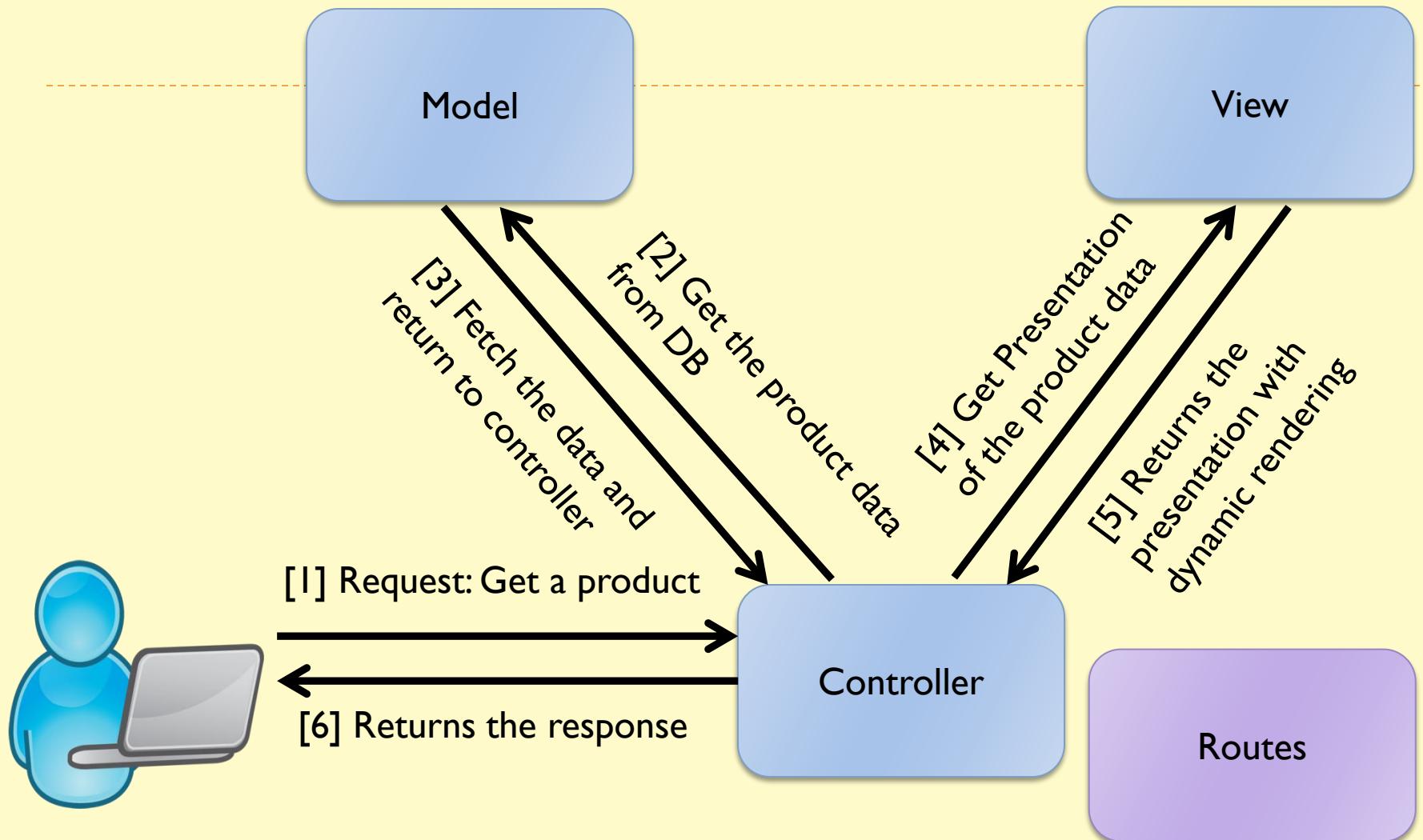
---

- ▶ It refers to the Data Related Logic
- ▶ Interaction with database (such as *SELECT, INSERT, UPDATE, DELETE*)
- ▶ It communicates with the controllers
- ▶ Can sometimes update or collaborate with the view  
(Depends on framework)
- ▶ In the example – Product, User, Transaction, Cart etc are model classes.

# View

---

- ▶ What the end users see (UI)
- ▶ Usually Consists of *html/css*
- ▶ Communicates with the controller
- ▶ While coding are passed as dynamic values from the controller
- ▶ A controller can have multiple associated views.
- ▶ In the example – product.html, user.html to view a html file



# Routes

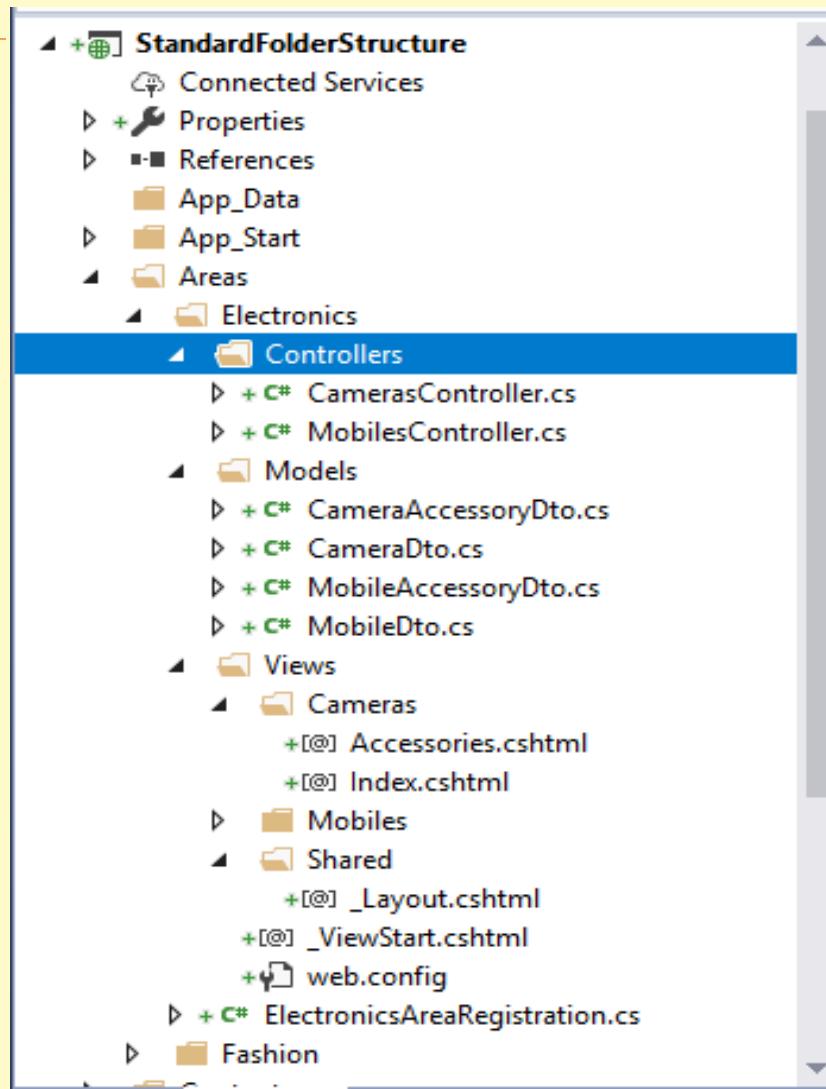
- ▶ Are urls to access a resource

- ▶ General structure -

`http://domainName/{controller}/{action}/{id}`

- ▶ `http://YourApp.com/Users/Profile/25`

```
public class RouteConfig {  
  
    public static void RegisterRoutes(RouteCollection routes) {  
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
  
        routes.MapRoute(  
            name: "Default",  
            url: "{controller}/{action}/{id}",  
            defaults: new { controller = "Home", action = "Index",  
                id = UrlParameter.Optional }  
        );  
    }  
}
```



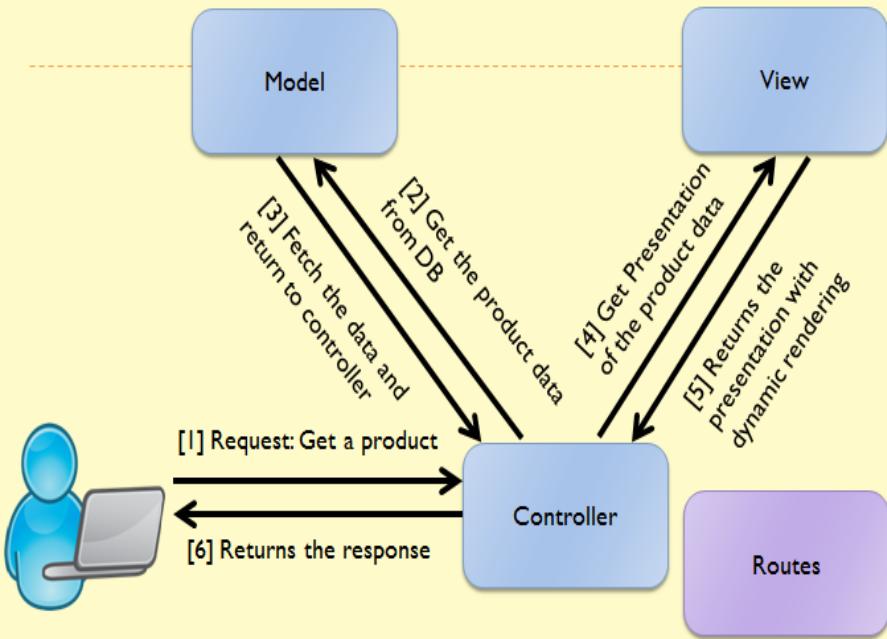
<http://yourapp.com/users/profile/1>

```
/routes  
  users/profile/:id = Users.getProfile(id)
```

```
/controllers  
  class Users{  
    function getProfile(id){  
      profile = this.UserModel.getProfile(id)  
  
      renderView('users/profile', profile)  
    }  
  }
```

```
/models  
  Class UserModel{  
    function getProfile(id){  
      data = this.db.get('SELECT * FROM users WHERE id = id')  
      return data;  
    }  
  }
```

```
/views  
  /users  
    /profile  
      <h1>{{profile.name}}</h1>  
      <ul>  
        <li>Email: {{profile.email}}</li>  
        <li>Phone: {{profile.phone}}</li>
```



### **When Used:**

Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.

### **Advantages:**

Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.

### **Disadvantages:**

Can involve additional code and code complexity when the data model and interactions are simple.

# Summary

---

- ▶ MVC is one of the most used architectural patterns.
- ▶ It divides the system into three parts – model, view and controller
- ▶ It provides a extensible separation of concern (SOC) from presentation view to data processing logic.



# SOFTWARE ENGINEERING

CSE 470 – Layered Architecture

BRAC University



Inspiring Excellence

# Lets recall Monolithic Software

- ▶ The end product come at end of the process model
- ▶ There is no separation of concern (code) of different software components.
- ▶ All code may be written in a single file with html, sql queries, logic checking etc.

Persistence Layer

Read/ write data  
on the storage

Presentation Layer

Business Layer

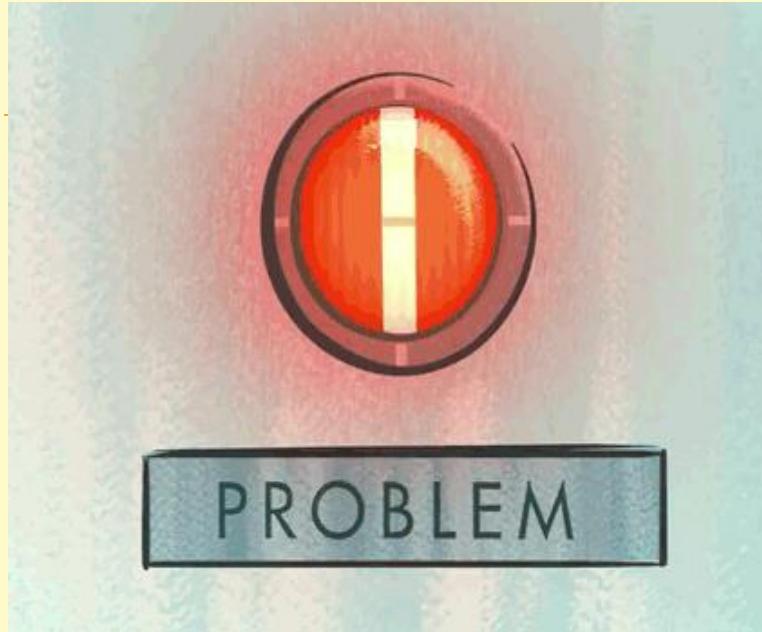
Database Layer

Operation on data –  
validate, aggregate,  
calculate

Actual residence  
of data

Customer  
Screen, Browser,  
Keyboard input





- ▶ No separation between components
- ▶ Changing a component affects other components. For example – What if I want to change the UI from JavaScript to Angular ?

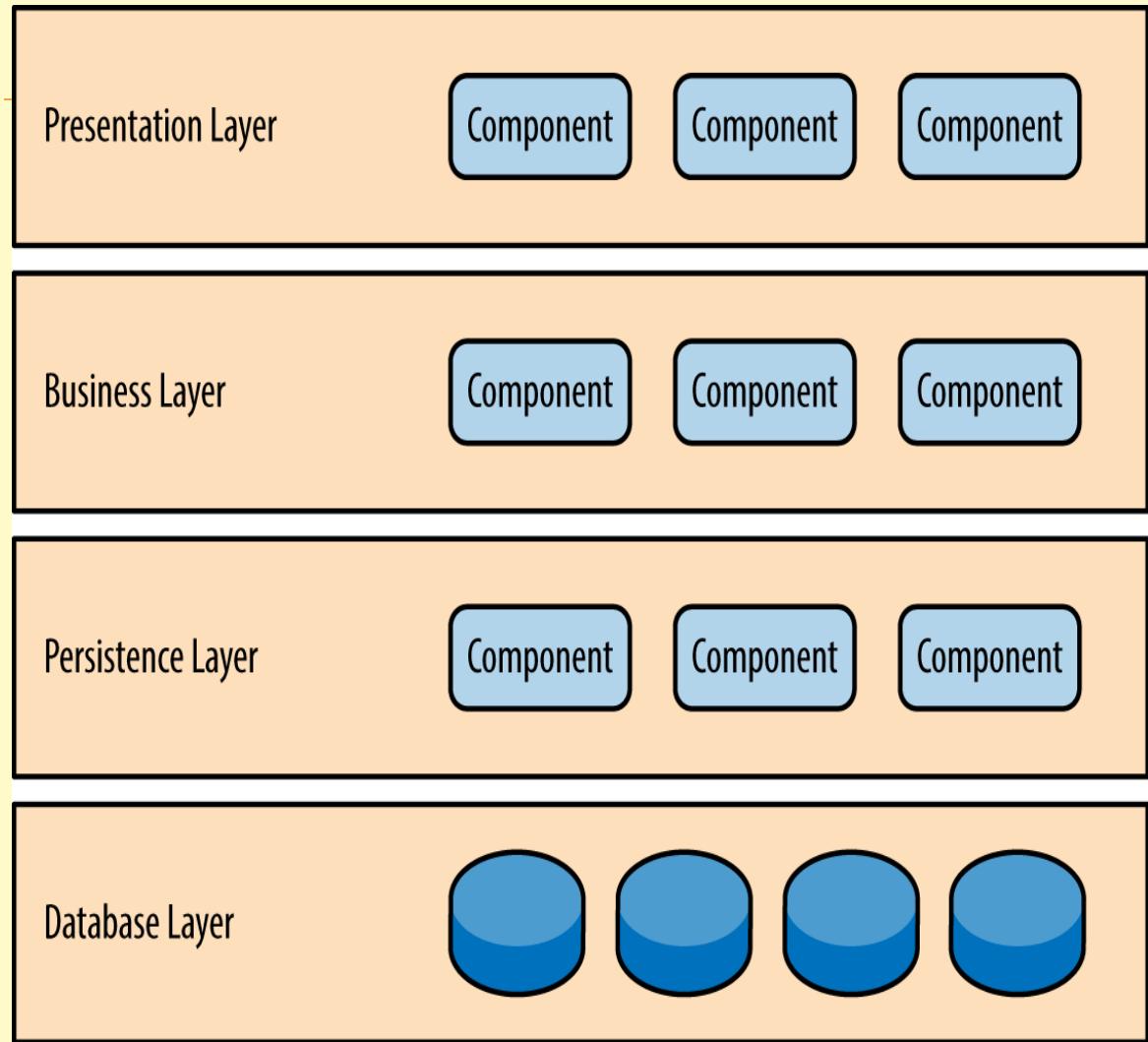
# Layered Architecture

---

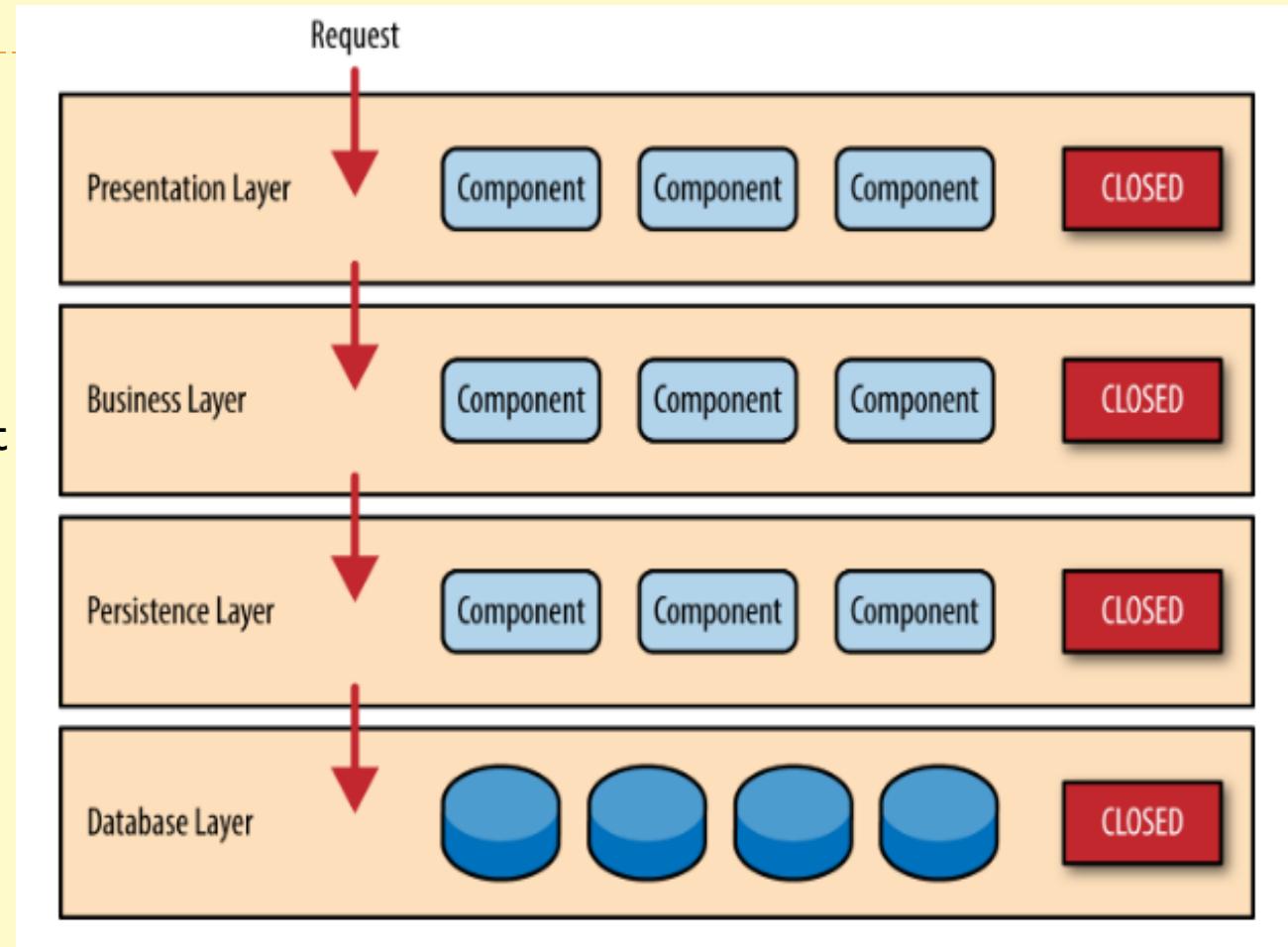
- ▶ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ▶ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.



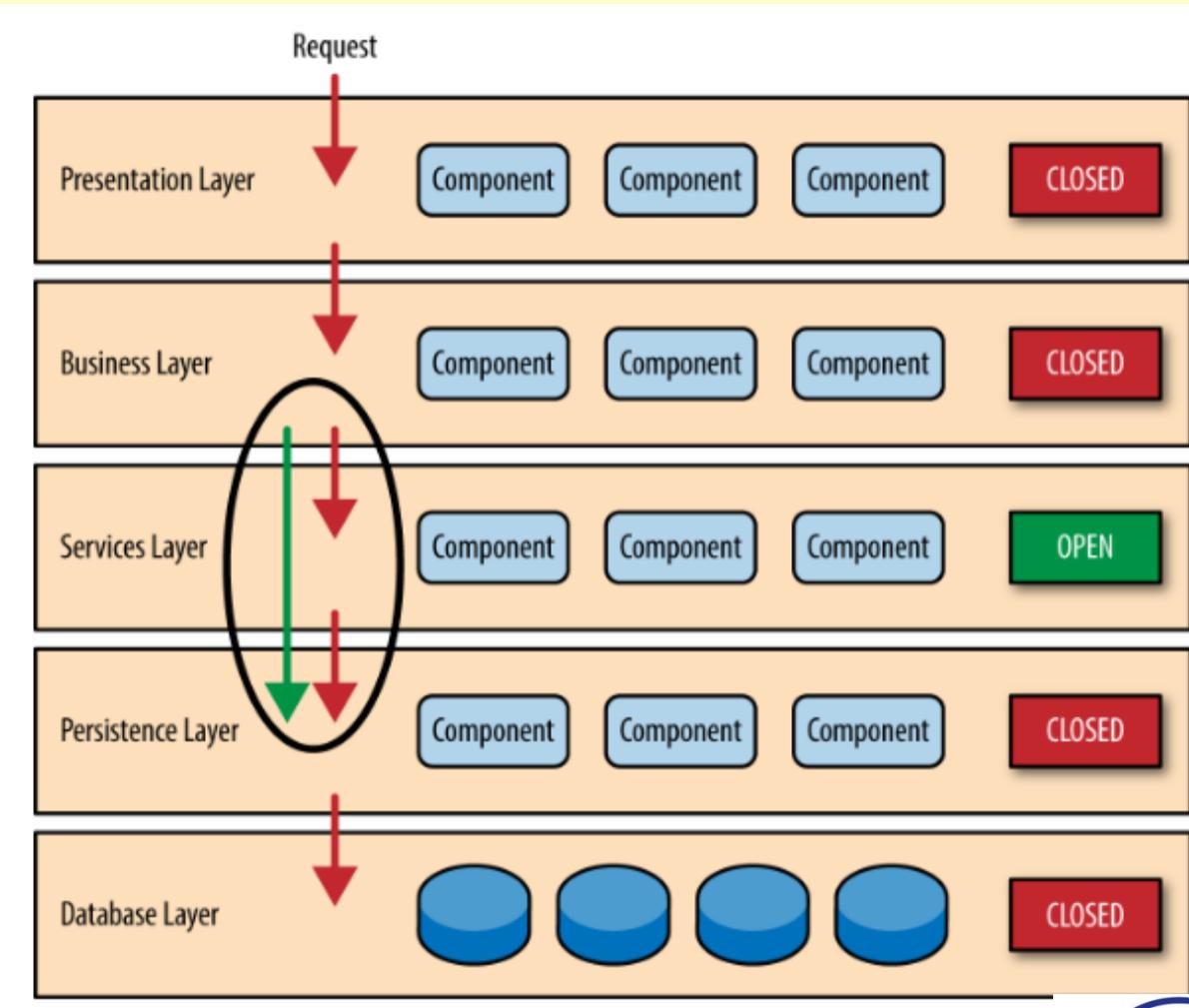
- ▶ Organizes the system into layers with related functionality associated with each layer.
- ▶ A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.



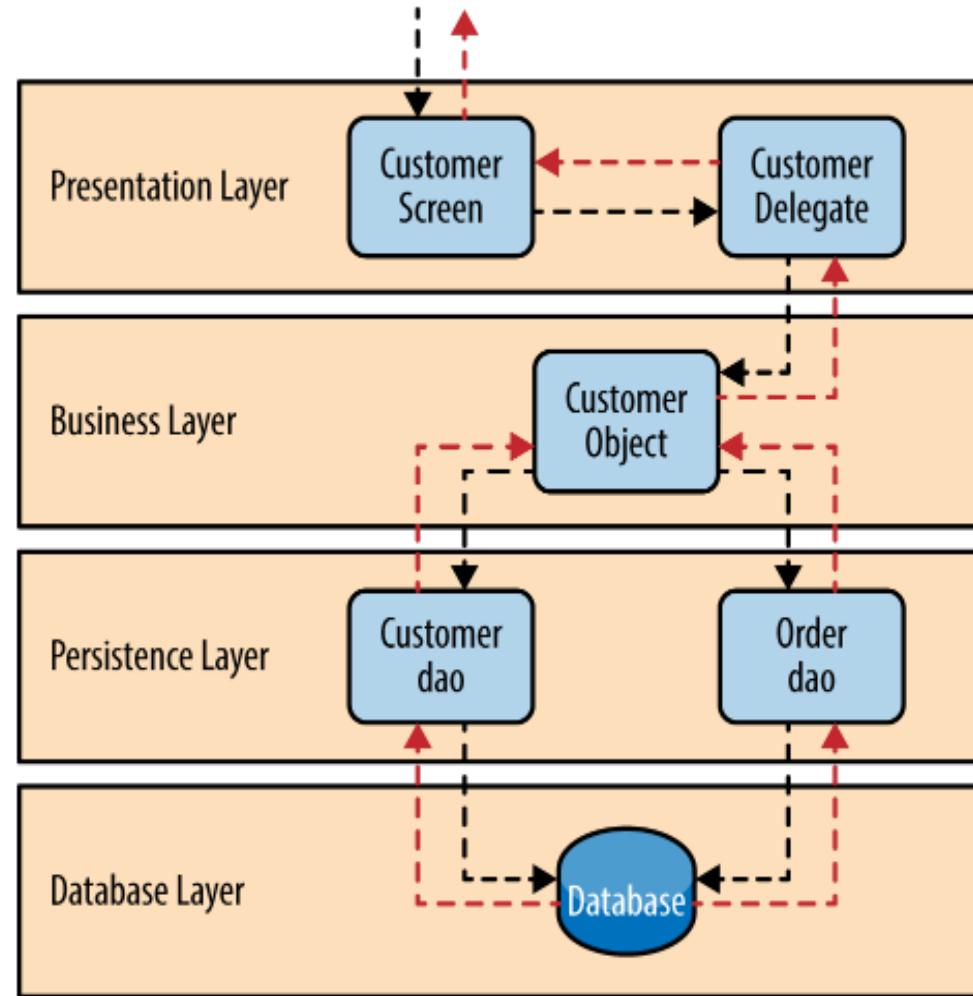
- ❑ Layers can be open or closed
- ❑ A closed layer can only be accessed by the layer above.
- ❑ A change in one layer does not affect others. It provides isolation.
- ❑ **However**, what if we want to add a new layer where shared utilities will be provided to be used by the **Business Layer**. But, we need to use it sometimes.



- ❑ Here comes the concept of open layers.
- ❑ An open can be bypassed by upper layers.
- ❑ Too many open layers may affect the actual essence of layered architecture.



# Example !!



## **When Used:**

1. Used when building new facilities on top of existing systems
2. When the development is spread across several teams with each team responsibility for a layer of functionality
3. When there is a requirement for multi-level security.

## **Advantages:**

1. Allows easy replacement or addition of entire layers so long as the interface is maintained.
2. Testing is easy as components are isolated

## **Disadvantages:**

1. In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it.
2. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.
3. A change in any layer still requires to restart the application.



# SOFTWARE ARCHITECTURE

CSE470 slide presented by

A.M.ESFAR-E-ALAM

AFRINA KHATUN

DR.ZAVID PARVEZ

HOSSAIN ARIF





01

REPOSITORY PATTERN

02

CLIENT SERVER PATTERN

03

PIPE AND FILTER PATTERN

04

Advantages and Disadvantages



# REPOSITORY PATTERN

# REPOSITORY

Before going into repository design pattern lets first understand what is repository:

In generic terms repository simply means a place where things are or can be stored.

This term has wide array of use in computer science for example code repository.

We are all familiar with GITHUB. it's a code repository

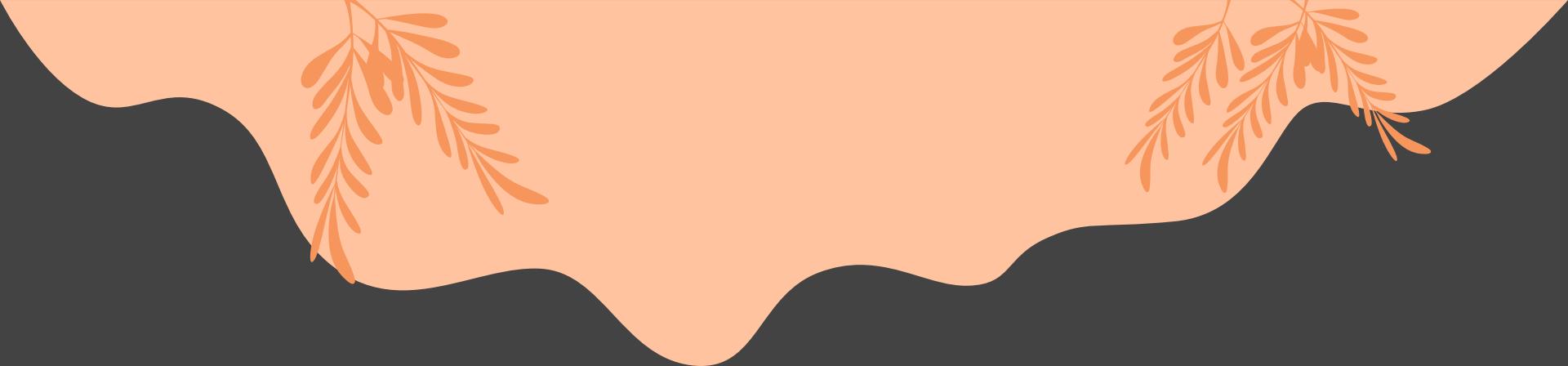


# DEFINING REPOSITORY PATTERN

So now we have a idea of what repository means lets think how that is connected to software architecture:

- Here we will have multiple sub systems.
- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.
- SO lets define repository pattern formally



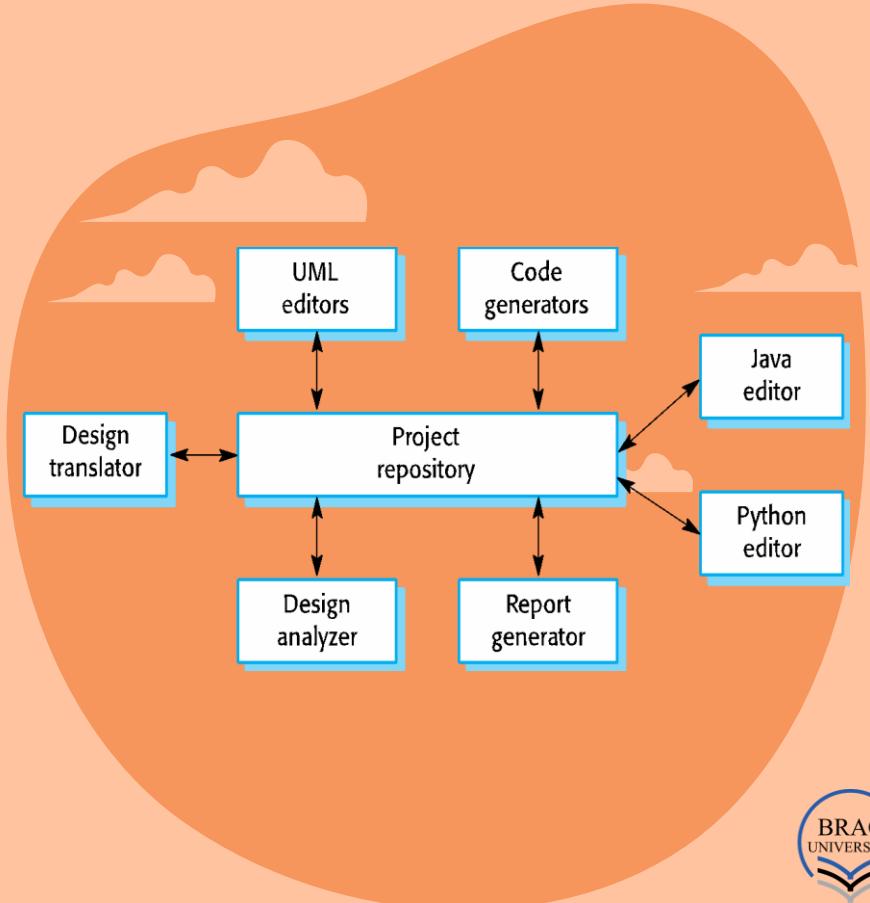


“A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes.”

## —FORMAL DEFINITION

# EXAMPLE!

- Lets think you are developing an IDE like 'eclipse'. What will you have there:
  - You will have editor for different language like java,c++.python
  - You will have code generator, Auto complete system
  - UML editor
  - Translator to compile the code etc.
- So here in the IDE where the components use a repository of system design information.
- Each software tool generates information which is then available for use by other tools.
- Have a look at figure for the example



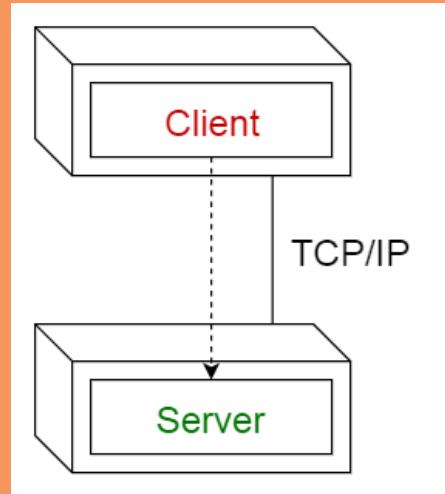
# Summarising

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	<ul style="list-style-type: none"><li>Components can be independent—they do not need to know of the existence of other components.</li><li>Changes made by one component can be propagated to all components.</li><li>All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.</li></ul>
Disadvantages	<ul style="list-style-type: none"><li>The repository is a single point of failure so problems in the repository affect the whole system.</li><li>May be inefficiencies in organizing all communication through the repository.</li><li>Distributing the repository across several computers may be difficult.</li></ul>

# 2. Client-server pattern

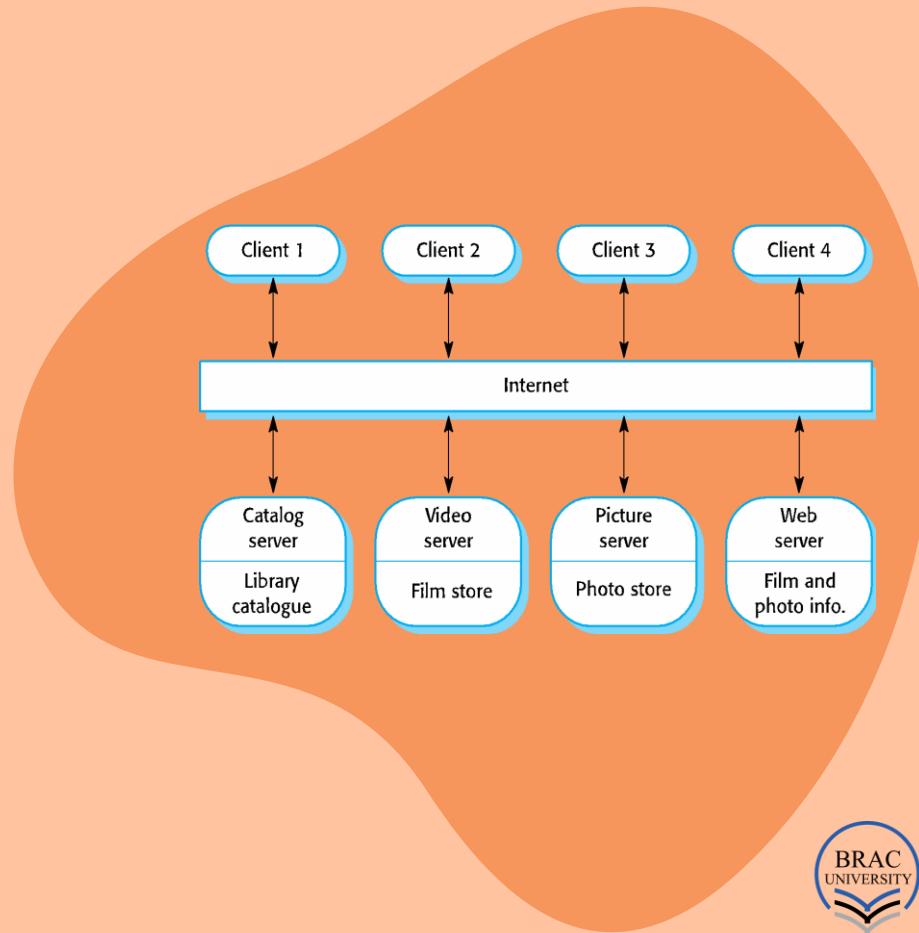
# Client-server pattern

- This pattern consists of two parties; a **server** and multiple **clients**.
- The server component will provide services to multiple client components.
- Clients request services from the server and the server provides relevant services to those clients.
- Furthermore, the server continues to listen to client requests.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.



# EXAMPLE

- › Online applications such as email, document sharing and banking.
- › Figure is an example of a film and video/DVD library organized as a client–server system.



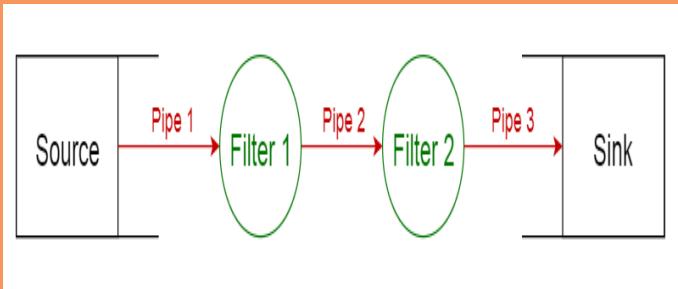
# Summerising Client-Server Architecture

<b>Description</b>	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
<b>When used</b>	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
<b>Advantages</b>	<ul style="list-style-type: none"><li>The principal advantage of this model is that servers can be distributed across a network.</li><li>General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.</li></ul>
<b>Disadvantages</b>	<ul style="list-style-type: none"><li>Each service is a single point of failure so susceptible to denial of service attacks or server failure.</li><li>Performance may be unpredictable because it depends on the network as well as the system.</li><li>May be management problems if servers are owned by different organizations.</li></ul>

# 3. Pipe-filter pattern

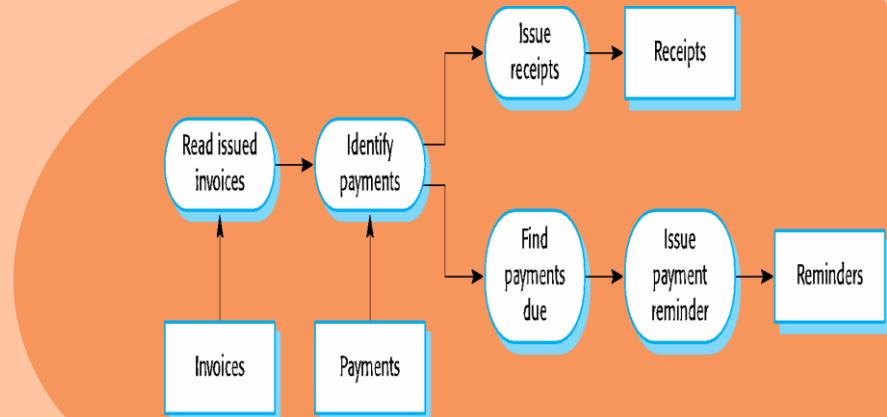
# What is it about?

This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a **filter** component. Data to be processed is passed through **pipes**. These pipes can be used for buffering or for synchronization purposes.



# Usage

- Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
- Workflows in bioinformatics.
- **Figure beside is an example of the pipe and filter architecture used in a payments system**



# SUMMERISING PIPE-FILTER

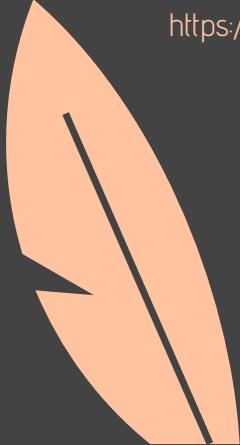
<b>Name</b>	<b>Pipe and filter</b>
<b>Description</b>	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
<b>When used</b>	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
<b>Advantages</b>	<ul style="list-style-type: none"><li>• Easy to understand and supports transformation reuse.</li><li>• Workflow style matches the structure of many business processes.</li><li>• Evolution by adding transformations is straightforward.</li><li>• Can be implemented as either a sequential or concurrent system.</li></ul>
<b>Disadvantages</b>	<ul style="list-style-type: none"><li>• The format for data transfer has to be agreed upon between communicating transformations.</li><li>• Each transformation must parse its input and unparse its output to the agreed form.</li><li>• This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.</li></ul>

# Thank you

Read supplementary slides

More Reading:

<https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>



Inspiring Excellence

# Design Pattern

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

## What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

## Usage of Design Pattern

Design Patterns have two main usages in software development.

### Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

### Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.

## Types of Design Patterns

As per the design pattern reference book *Design Patterns - Elements of Reusable Object-Oriented Software* , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns.

We'll also discuss another category of design pattern: J2EE design patterns.

# Types of Patterns

## Pattern & Description

### Creational Patterns

These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

### Structural Patterns

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

## Behavioral Patterns

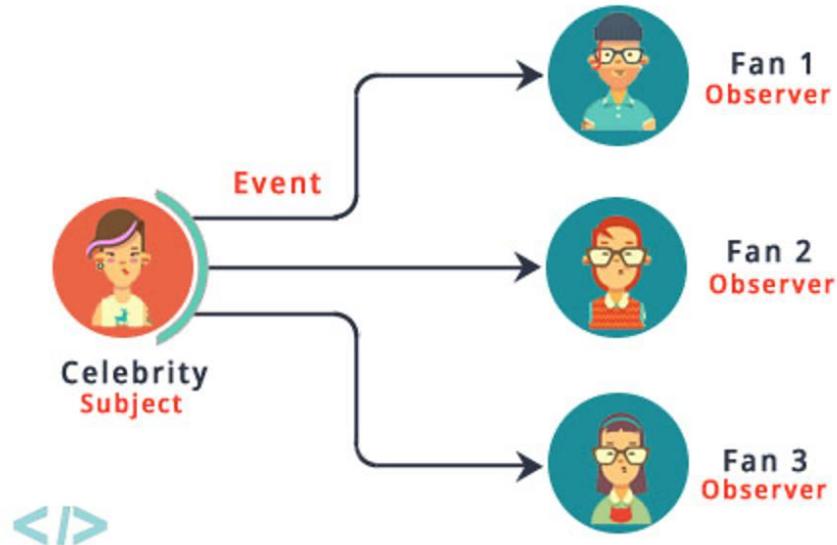
These design patterns are specifically concerned with communication between objects.

## J2EE Patterns

These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

# Observer Pattern

**Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



# Observer Pattern

8

```
public class Celebrity{  
    private List<Fan> fans = new ArrayList<Fan>();  
    private int state;  
    void attach(Fan f){  
        fans.add(f);  
    }  
    void remove(Fan f){  
        fans.remove(f);  
    }  
    void notify(){  
        foreach( Fan f: fans)  
            f.update(this);  
    }  
    void setState(int newState){  
        state = newState;  
        notify();  
    }  
    int getState(){  
        return state;  
    }  
}
```

*Celebrity class may look like this.*

# Observer Pattern

9

```
public class Celebrity{  
    private List<Fan> fans = new ArrayList<Fan>();  
    private int state;  
    void attach(Fan f){  
        fans.add(f);  
    }  
    void remove(Fan f){  
        fans.remove(f);  
    }  
    void notify(){  
        foreach( Fan f: fans)  
            f.update(this);  
    }  
    void setState(int newState){  
        state = newState;  
        notify();  
    }  
    int getState(){  
        return state;  
    }  
}
```

*Celebrity class may look like this.*

**Now add the Fan class**

# Observer Pattern

10

```
public class Celebrity{  
    private List<Fan> fans = new ArrayList<Fan>();  
    private int state;  
    void attach(Fan f){  
        fans.add(f);  
    }  
    void remove(Fan f){  
        fans.remove(f);  
    }  
    void notify(){  
        foreach( Fan f: fans)  
            f.update(this);  
    }  
    void setState(int newState){  
        state = newState;  
        notify();  
    }  
    int getState(){  
        return state;  
    }  
}  
  
public class Fan{  
    private List<Celebrity> celebrities= new ArrayList<Celebrity>();  
    void update(Celebrity c){  
        c.getState();  
    }  
    void addCelebrity(Celebrity c){  
        celebrities.add(c);  
        c.attach(this);  
    }  
    void removeCelebrity(Celebrity c){  
        celebrities.remove(c);  
        c.remove(this);  
    }  
}
```

# Observer Pattern

11

**From main method –**

```
Fan f1 = new Fan();  
Fan f2 = new Fan();
```

```
Celebrity c1 = new Celebrity ();  
Celebrity c2 = new Celebrity ();
```

```
f1.addCelebrity(c1);  
f1.addCelebrity(c2);  
f2.addCelebrity (c1);
```

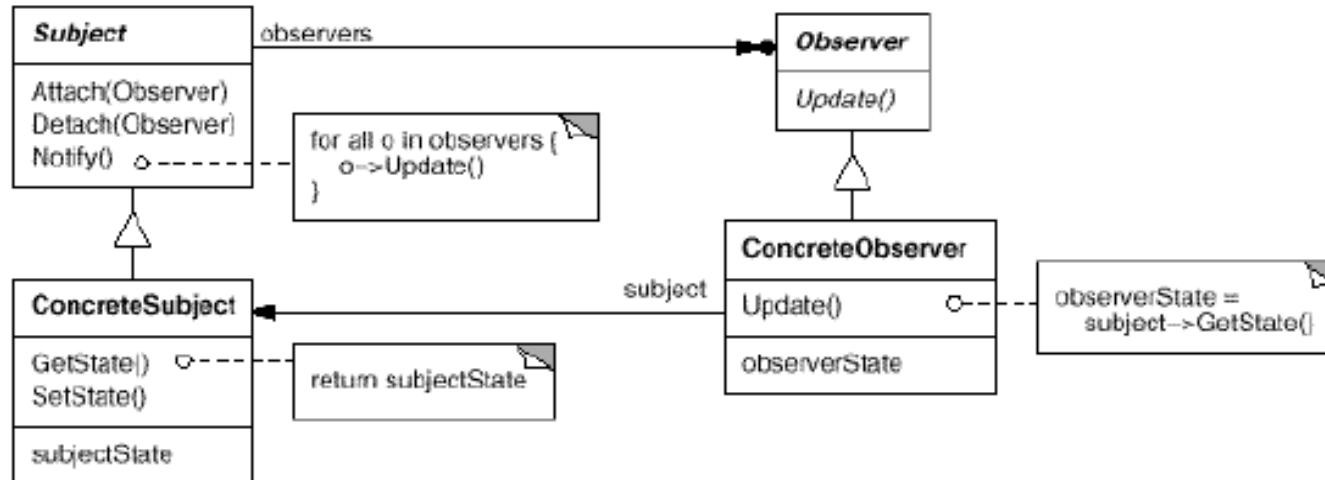
# Participants

12

- **Subject:** knows its observers. Any number of Observer objects may observe a subject. It sends a notification to its observers when its state changes. (ex: Celebrity)
- **Observer:** defines an updating interface for objects that should be notified changes in a subject. (ex: Fans)
- **ConcreteSubject:** (ex: FilmCelebrity, FashionCelebrity)
- **ConcreteObserver** (ex: FilmFan, FashsionFan)

# Structure

13



# Advanced Observer with Concrete subjects and observers

Public FilmCelebrity extends

Celebrity{

    private int state;

    void setState(int newState){

        state = newState;

        notify();

}

    int getState(){

        return state;

}

}

# Advanced Observer with Concrete subjects and observers

```
Public FilmCelebrity extends  
Celebrity{  
    private int state;  
    void setState(int newState){  
        state = newState;  
        notify();  
    }  
    int getState(){  
        return state;  
    }  
}
```

```
public class FilmFan extends Fan{  
    private List<FilmCelebrity> filmCelebrities= new  
ArrayList<FilmCelebrity>();  
    void update(FilmCelebrity fc){  
        if(filmCelebrities.contains(fc)){  
            fc.getState();  
        }  
    }  
    void addCelebrity(FilmCelebrity fc){  
        celebrities.add(fc);  
        fc.attach(this);  
    }  
    void removeCelebrity(FilmCelebrity fc){  
        celebrities.remove(fc);  
        fc.remove(this);  
    }  
}
```

# CSE470 – Software Engineering

## SINGLETON AND ADAPTER PATTERN



# Singleton Pattern

```
public class HelpDesk{  
    public void getService(){  
        // Implement the service  
    }  
}
```

```
public class Student{  
    HelpDesk hd = new HelpDesk();  
    hd.getService();  
}
```

```
public class Teacher{  
    HelpDesk hd = new HelpDesk();  
    hd.getService();  
}
```

***In reality one Single HelpDesk is serving all. No need for multiple objects.***

```
public class HelpDesk{  
    public void getService(){  
        // Implement the service  
    }  
    private static HelpDesk helpDesk;  
  
    private HelpDesk(){  
        // No other class will be able to create  
        instance  
    }  
    public static HelpDesk getInstance(){  
        if(helpDesk == null){  
            helpDesk = new HelpDesk(); // Lazy  
            instance  
        }  
        return helpDesk;  
    }  
  
    public class Student{  
        //HelpDesk hd = new HelpDesk();  
        HelpDesk hd = HelpDesk.getInstance();  
        hd.getService();  
    }  
}
```

```
public class Teacher{  
    //HelpDesk hd = new  
    HelpDesk();  
    HelpDesk hd =  
    HelpDesk.getInstance();  
    hd.getService();  
}
```



# Singleton Pattern

**Intent:** Ensure a class only has one instance, and provide a global point of access to it.

**Motivation:** The reason behind is

- ▶ More than one instance will result in incorrect program behaviour. (thread specific)
- ▶ More than one instance will result in the overuse of resources. (ex: database connection string)
- ▶ Some classes should have only one instance throughout the system for (ex: printer spooler)

**Classification:** Classified as one of the most known Creational Pattern



# Singleton PatternStructure

```
public class Singleton{  
    private static Singleton singleInstance;  
  
    private Singleton(){  
        //nothing to do as object initiation will be done  
        once  
    }  
    public static Singleton getInstance(){  
        if(singleInstance == null){  
            singleInstance = new Singletong(); // Lazy  
            instance  
        }  
        return singleInstance;  
    }  
}
```

**From main method call -**

```
Singleton instance =  
Singleton.getInstance();
```

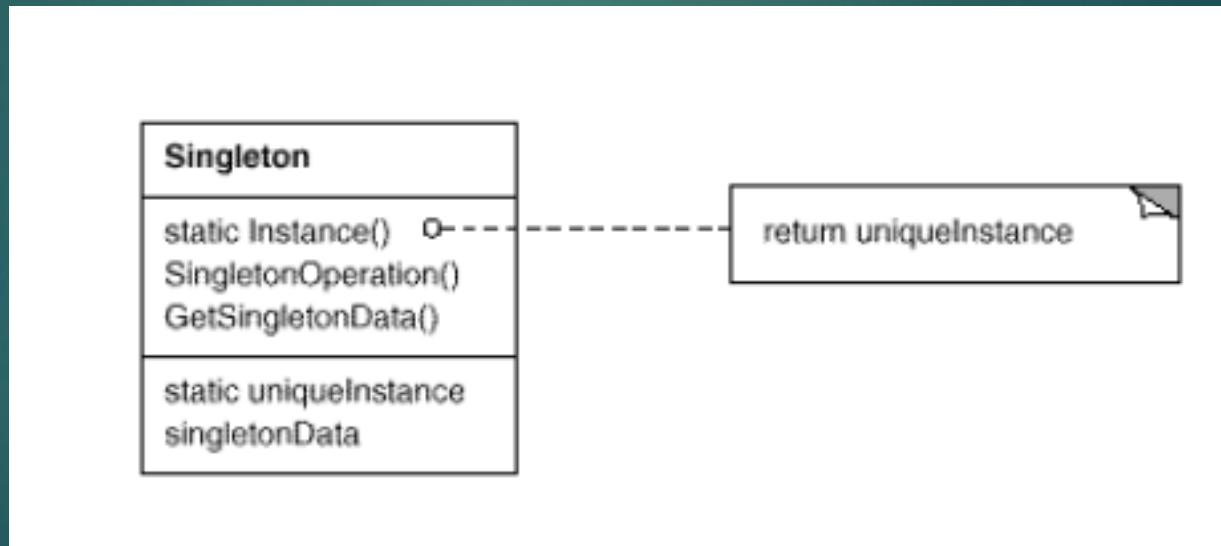


# Singleton

## Participants:

- **Singleton**-defines an Instance operation that lets clients access its unique instance. Instance is a class operation .It may be responsible for creating its own unique instance. (ex: Singleton)

## Structure:



# Singleton

**Lazy instance:** Singleton make use of lazy initiation of the class. It means that its creation is deferred until it is first used. It helps to improve performance, avoid wasteful computation and reduce program memory requirement.

```
if(singleInstance == null){  
    singleInstance = new Singletong(); // Lazy initialization  
}
```



# Adapter Pattern

In the real world...

we are very familiar with adapters and what they do

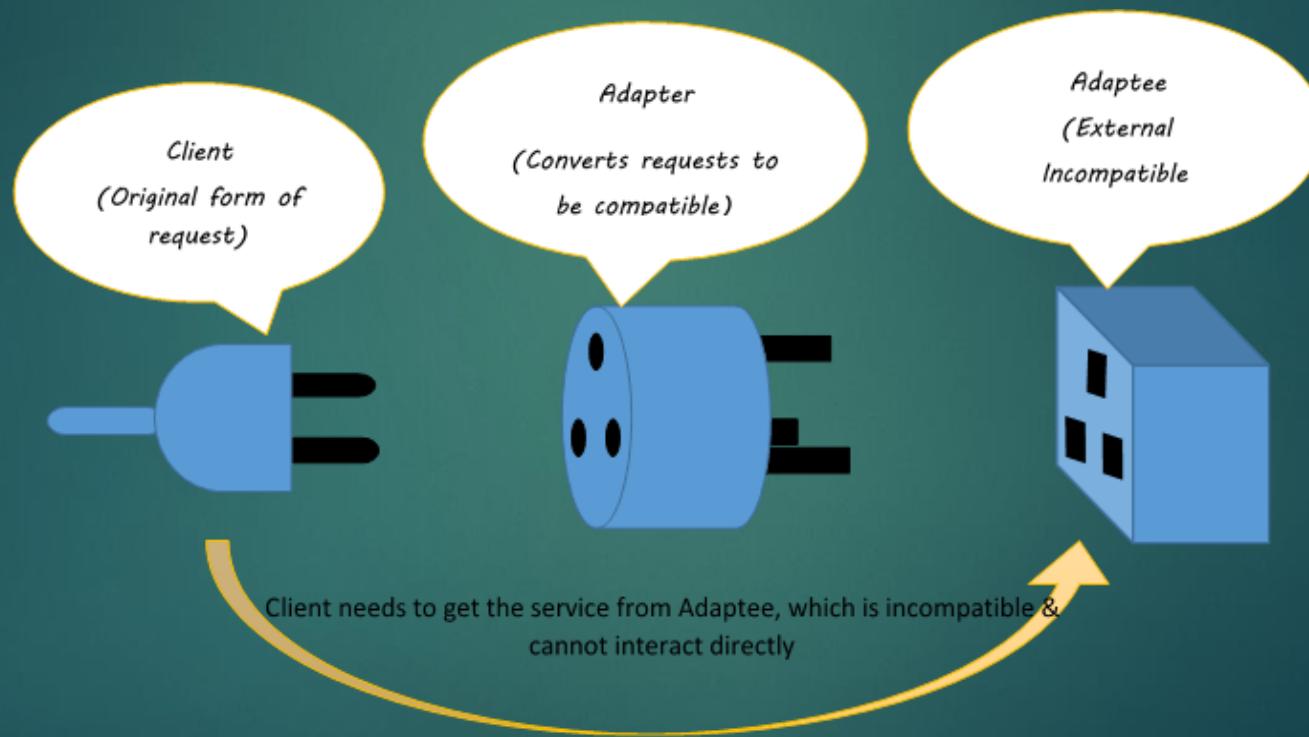


Figure 1-Adapter Pattern Concept

# What about object oriented adapters?

## **Intent:**

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

## **Classified as:**

A Structural Pattern

(Structural patterns are concerned with how classes and objects are composed to form larger structures.)

## **Also Known As:**

Wrapper

# Adapter Pattern

Adapter pattern can be solved in one of two ways:

- ▶ **Class Adapter**: its Inheritance based solution
- ▶ **Object Adapter**: its Object creation based solution



# Class Adapter

11

**Scenario: I have a pizza making store that creates different pizzas based on the choices of people of different locations. For example – people of Dhaka like DhakaStylePizza, people of Sylhet like SylhetStylePizza.**



# Class Adapter

12

**Scenario:** *I have a pizza making store that creates different pizzas based on the choices of people of different locations. For example – people of Dhaka like DhakaStylePizza, people of Sylhet like SylhetStylePizza.*

**Solution:** To meet the scenario, we can declare a Pizza interface and different location people can make their own style pizza by implementing the same interface.



Inspiring Excellence

# Class Adapter

```
Public Interface Pizza{  
    abstract void toppings();  
    abstract void bun();  
}
```

```
Public class DhakaStylePizza implements  
Pizza{  
    public void toppings(){  
        print("Dhaka chesse toppings");  
    }  
    public void bun(){  
        print("Dhaka bread bun");  
    }  
}
```



# Class Adapter

- ▶ Now we want to support ChittagongStylePizza.
- ▶ The customer of Chittagong are rigid. They want to use the authentic **existing class, ChittagongPizza**
- ▶ **But we can not call it directly**, as its not name same as **our Pizza interface**.

```
public class ChittagongPizza{  
    public void sausage(){  
        print("Ctg pizza");  
    }  
    public void bread(){  
        print("Ctg bread");  
    }  
}
```

```
Public Interface Pizza{  
    abstract void toppings();  
    abstract void bun();  
}
```



# Class Adapter

- ▶ We want to adapt the existing ChittagongPizza, so it's a Adaptee.
- ▶ To do so, **introduce a** Class Adapter, **ChittagongClassAdapter**

Public class ChittagongClassAdapter extends ChittagongPizza implements Pizza{

```
    public void toppings(){
        this.sausage(); → public class ChittagongPizza{
    }
    public void bun(){
        this.bread(); →     public void sausage(){
    }
    }                                print("Ctg pizza");
}                                }
}                                public void bread(){
                                print("Ctg bread");
}
}
```



# Class Adapter

- ▶ We want to adapt the existing ChittagongPizza, so it's a Adaptee.
- ▶ To do so, **introduce a** Class Adapter, **ChittagongClassAdapter**
- ▶ **Customer use the adapter to adapt the adaptee.**

Public class ChittagongClassAdapter extends ChittagongPizza implements Pizza{

```
    public void toppings(){
        this.sausage();
    }
    public void bun(){
        this.bread();
    }
}
```

public class ChittagongPizza{  
 public void sausage(){  
 print("Ctg pizza");  
 }
 public void bread(){  
 print("Ctg bread");  
 }
}

From main method, customer call – }

```
Pizza adaptedPizza = new ChittagongClassAdapter();
adaptedPizza.toppings();
adaptedPizza.bun();
```

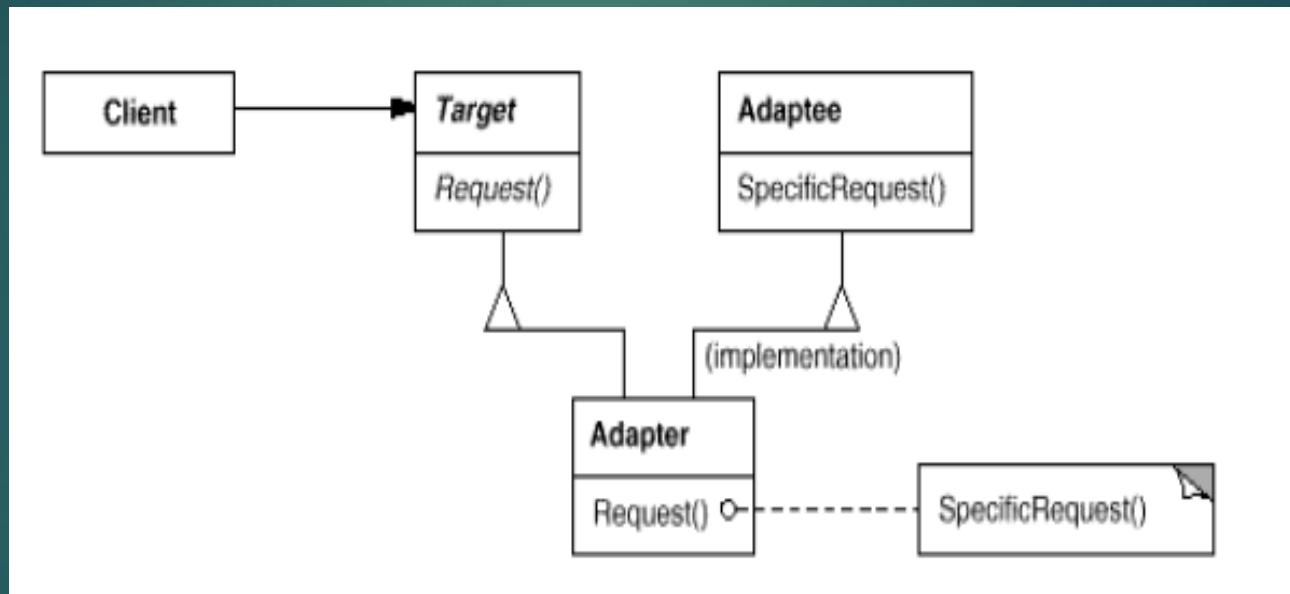
# Adapter Pattern

## Participants:

- ▶ **Target:** defines the domain-specific interface that Client uses.  
(ex: Pizza)
- ▶ **Client:** collaborates with objects conforming to the Target interface.
- ▶ **Adaptee:** defines an existing interface that needs adapting (ex: ChittagongPizza)
- ▶ **Adapter:** adapts the interface of Adaptee to the Target interface. (ex: ChittagongClassAdapter)

# Adapter Pattern

## ► Structure:



# Object Adapter

We have the existing adaptee.

Now, create a object adapter for adapting the same ChittagongPizza **existing class**.

```
public class ChittagongObjectAdapter implements Pizza{  
    private ChittagongPizza ctgPizza;  
  
    public ChittagongStylePizzaObjectAdapter(){  
        ctgPizza = new ChittagongPizza();  
    }  
    public void toppings(){  
        ctgPizza.sausage();  
    }  
    public void bun(){  
        ctgPizza.bread();  
    }  
}
```

```
public class ChittagongPizza{  
    public void sausage(){  
        print("Ctg pizza");  
    }  
    public void bread(){  
        print("Ctg bread");  
    }  
}
```

# Pros and Cons

- ▶ **Class Adapter:** in this case, as it extends the adaptee, it can override the adaptee's methods. But, It can not use the adaptee's subclasses.
- ▶ **Object Adapter:** As we use object, a parent class object can store subclass object. So, It can adapt the subclasses as well. However, it can not override any behaviour of adaptee.

# Software Testing

INTRODUCTION



Inspiring Excellence

# Intro

- ▶ Suppose you are being asked to lead the team to test the software that controls a new X-ray machine. Would you take that job?
- ▶ What if the contract says you'll be charged with murder in case a patient dies because of a mal-functioning of the software?



Inspiring Excellence

# State-of-the-Art

- ▶ 30-85 errors are made per 1000 lines of source code.
- ▶ extensively tested software contains 0.5-3 errors per 1000 lines of source code.
- ▶ testing is postponed, as a consequence: the later an error is discovered, the more it costs to fix it.
- ▶ error distribution: 60% design, 40% implementation. 66% of the design errors are not discovered until the software has become operational.

# Software testing

- ▶ Check software correctness.
- ▶ Testing is the process of evaluating a system or its component(s) with the intent to find that whether it satisfies the specified requirements or not.
- ▶ Who does testing?
  - ▶ Software Tester
  - ▶ Software Developer
  - ▶ Project Lead/Manager
  - ▶ End User



# Error, fault, failure

- ▶ An **error** is a human activity resulting in software containing a fault
- ▶ A **fault** is the manifestation of an error
- ▶ A fault may result in a **failure**
- ▶ Error → Fault → Failure



# Classification of testing techniques

- ▶ Classification based on the criterion to measure the adequacy of a set of test cases:
  - ▶ coverage-based testing
  - ▶ fault-based testing
  - ▶ error-based testing
- ▶ Classification based on the source of information to derive test cases:
  - ▶ black-box testing (functional, specification-based)
  - ▶ white-box testing (structural, program-based)
  - ▶ Grey-box testing

# Comparison

	<b>Black Box Testing</b>	<b>Grey Box Testing</b>	<b>White Box Testing</b>
1.	The Internal Workings of an application are not required to be known	Somewhat knowledge of the internal workings are known	Tester has full knowledge of the Internal workings of the application
2.	Also known as closed box testing, data driven testing and functional testing	Another term for grey box testing is translucent testing as the tester has limited knowledge of the insides of the application	Also known as clear box testing, structural testing or code based testing
3.	Performed by end users and also by testers and developers	Performed by end users and also by testers and developers	Normally done by testers and developers
4.	-Testing is based on external expectations -Internal behavior of the application is unknown	Testing is done on the basis of high level database diagrams and data flow diagrams	Internal workings are fully known and the tester can design test data accordingly
5.	This is the least time consuming and exhaustive	Partly time consuming and exhaustive	The most exhaustive and time consuming type of testing
6.	Not suited to algorithm testing	Not suited to algorithm testing	Suited for algorithm testing
7.	This can only be done by trial and error method	Data domains and Internal boundaries can be tested, if known	Data domains and Internal boundaries can be better tested

# Testing Myths



Inspiring Excellence

- ▶ **Myth:** Testing is too expensive.
- ▶ **Reality:** There is a saying, pay less for testing during software development or pay more for maintenance or correction later. Early testing saves both time and cost in many aspects however, reducing the cost without testing may result in the improper design of a software application rendering the product useless.
- ▶ **Myth:** Testing is time consuming.
- ▶ **Reality:** During the SDLC phases testing is never a time consuming process. However diagnosing and fixing the error which is identified during proper testing is a time consuming but productive activity.
- ▶ **Myth:** Testing cannot be started if the product is not fully developed.
- ▶ **Reality:** No doubt, testing depends on the source code but reviewing requirements and developing test cases is independent from the developed code. However iterative or incremental approach as a development life cycle model may reduce the dependency of testing on the fully developed software.

# Testing Myths (2)



- ▶ **Myth:** Complete Testing is Possible.
- ▶ **Reality:** It becomes an issue when a client or tester thinks that complete testing is possible. It is possible that all paths have been tested by the team but occurrence of complete testing is never possible. There might be some scenarios that are never executed by the test team or the client during the software development life cycle and may be executed once the project has been deployed
- ▶ **Myth:** If the software is tested then it must be bug free.
- ▶ **Reality:** This is a very common myth which clients, Project Managers and the management team believe in. No one can say with absolute certainty that a software application is 100% bug free even if a tester with superb testing skills has tested the application.

# Testing Myths (3)



- ▶ **Myth:** Missed defects are due to Testers.
- ▶ **Reality:** It is not a correct approach to blame testers for bugs that remain in the application even after testing has been performed. This myth relates to Time, Cost, and Requirements changing Constraints. However the test strategy may also result in bugs being missed by the testing team.
- ▶ **Myth:** Testers should be responsible for the quality of a product.
- ▶ **Reality:** It is a very common misinterpretation that only testers or the testing team should be responsible for product quality. Tester's responsibilities include the identification of bugs to the stakeholders and then it is their decision whether they will fix the bug or release the software. Releasing the software at the time puts more pressure on the testers as they will be blamed for any error.

# Testing Myths (4)



Inspiring Excellence

- ▶ **Myth:** Test Automation should be used wherever it is possible to use it and to reduce time.
- ▶ **Reality:** Yes it is true that Test Automation reduces the testing time but it is not possible to start Test Automation at any time during Software development. Test Automaton should be started when the software has been manually tested and is stable to some extent. Moreover, Test Automation can never be used if requirements keep changing.
- ▶ **Myth:** Any one can test a Software application.
- ▶ **Reality:** People outside the IT industry think and even believe that any one can test the software and testing is not a creative job. However testers know very well that this is myth. Thinking alternatives scenarios, try to crash the Software with the intent to explore potential bugs is not possible for the person who developed it.

# Level of testing

- ▶ Functional Testing
  - ▶ Unit Testing
  - ▶ Integration Testing
  - ▶ System Testing
  - ▶ Regression Testing
  - ▶ Acceptance Testing
  - ▶ Alpha testing
  - ▶ Beta Testing
- ▶ Non-Functional Testing
  - ▶ Performance Testing (Load and Stress)
  - ▶ Usability Testing (Usability vs UI testing)
  - ▶ Security Testing
  - ▶ Portability Testing



# Functional Testing

## ► Unit Testing

- ▶ This type of testing is performed by the developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is separate from the test data of the quality assurance team.
- ▶ The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

## ► Integration Testing

- ▶ The testing of combined parts of an application to determine if they function correctly together is Integration testing. There are two methods of doing Integration Testing Bottom-up Integration testing and Top Down Integration testing.
- ▶ **Bottom-up integration** testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.
- ▶ **Top-Down integration** testing, the highest-level modules are tested first and progressively lower-level modules are tested after that. In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing.

## ► System Testing

- ▶ This is the next level in the testing and tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets Quality Standards. This type of testing is performed by a specialized testing team.

# Non-Functional Testing

- ▶ **Performance Testing**
  - ▶ It is mostly used to identify any bottlenecks or performance issues rather than finding the bugs in software. There are different causes which contribute in lowering the performance of software:
    - ▶ Network delay.
    - ▶ Client side processing.
    - ▶ Database transaction processing.
    - ▶ Load balancing between servers.
    - ▶ Data rendering.
  - ▶ Performance testing is considered as one of the important and mandatory testing type in terms of following aspects:
    - ▶ Speed (i.e. Response Time, data rendering and accessing)
    - ▶ Capacity
    - ▶ Stability
    - ▶ Scalability
  - ▶ It can be either qualitative or quantitative testing activity and can be divided into different sub types such as Load testing and Stress testing.

# Non-Functional Testing

## ► **Load Testing**

- ▶ A process of testing the behavior of the Software by applying maximum load in terms of Software accessing and manipulating large input data. It can be done at both normal and peak load conditions. This type of testing identifies the maximum capacity of Software and its behavior at peak time.
- ▶ Most of the time, Load testing is performed with the help of automated tools such as Load Runner, AppLoader, IBM Rational Performance Tester, Apache JMeter, Silk Performer, Visual Studio Load Test etc.

## ► **Stress Testing**

- ▶ This testing type includes the testing of Software behavior under abnormal conditions. Taking away the resources, applying load beyond the actual load limit is Stress testing.
- ▶ The main intent is to test the Software by applying the load to the system and taking over the resources used by the Software to identify the breaking point. This testing can be performed by testing different scenarios such as:
- ▶ Shutdown or restart of Network ports randomly.
- ▶ Turning the database on or off.
- ▶ Running different processes that consume resources such as CPU, Memory, server etc.

# Test-Driven Development (TDD)

- ▶ First write the tests, then do the design/implementation
- ▶ Part of agile approaches like XP
- ▶ Supported by tools, eg. JUnit
- ▶ Is more than a mere test technique; it subsumes part of the design work

## Steps of TDD

1. Add a test
2. Run all tests, and see that the system fails
3. Make a small change to make the test work
4. Run all tests again, and see they all run properly
5. Refactor the system to improve its design and remove redundancies

# Thank you

# SOFTWARE ENGINEERING

CSE 470 – Unit Testing

BRAC University

# Testing

## Manual Testing

- Done by Users
- Refers to using the software and identifying problems
- Generally uses excel sheet or similar tools to keep track of traces

## Automated Testing

- Done by Development team – developers, testers
- Refers to writing and running test cases and to identify problems
- Can be done in many ways – testing libraries, automated testing tools

# Manual Testing Example

Test Case ID	Test Title	Testing Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TU01	Check Customer Login with valid Data	1. Go to site <a href="http://mySampleProject.com">http://mySampleProject.com</a>	UserId = guru99 Password = pass99	User should Login into application	As Expected	Pass
		2. Enter UserId				
		3. Enter Password				
		4. Click Submit				
TU02	Check Customer Login with invalid Data	1. Go to site <a href="http://mySampleProject.com">http://mySampleProject.com</a>	UserId = guru99 Password = glass99	User should not Login into application	As Expected	Pass
		2. Enter UserId				
		3. Enter Password				
		4. Click Submit				



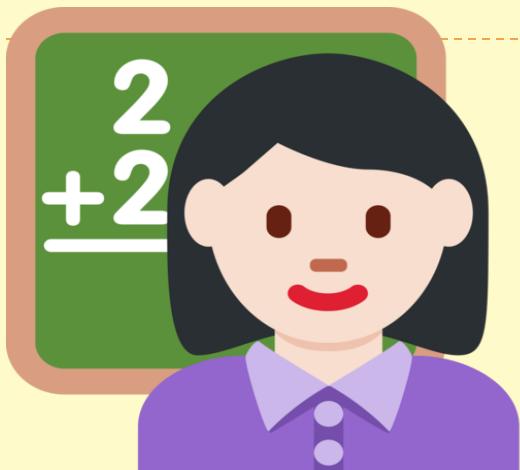
# Unit Testing

---

- ▶ Unit testing is a type of automated testing to check if the small piece of code is doing what it is suppose to do.
- ▶ Unit testing checks a single component of an application. The components are methods / functions
- ▶ The scope of Unit testing is narrow, it covers the Unit or small piece of code under test. Therefore while writing a unit test shorter codes are used that target just a single class.
- ▶ Unit testing comes under White box testing type.



# Example – Math Test



1. Teacher prepares the question set
  2. Give each question to students who appear under the test.
  3. Students do the math and return the calculated result
  4. The teacher knows the right result and compares it with the student's calculated result.
  5. Mark the student pass / fail
1. Developers prepares some prearrangements known as, **TEST SETUP**,
  2. For each unit, create a **TEST METHOD/CASE**. Then, write the **TEST STATEMENTS** in the method which will call the unit under test.
  3. The unit under test will execute and return a result known as, **ACTUAL RESULT**.
  4. The test method have the right result known as, **EXPECTED RESULT** and compares it with the previous actual result,
  5. The comparison identifies whether the unit passed or failed.

# How to write unit tests

---

- ▶ A *TEST CASE* is a set of conditions or variables or instructions under which a tester will determine whether a system under test satisfies requirements or works correctly.
- ▶ Setup / pre-conditions are usually defined as a separate property of the test case. Are generally written under the setup method.
- ▶ test steps: instructions / statements written with in the test method
- ▶ a predefined expected result is required for invoking the unit



- ▶ Please watch the next video on writing unit tests for detailed understanding



# SOFTWARE ENGINEERING

CSE 470 – Software Quality

BRAC University



# What is Software Quality?

---

- ▶ Quality is an attribute of software that implies the software meets its specification
- ▶ The assessment of software quality is a subjective process where the quality management team has to use their judgment to decide if an acceptable level of quality has been achieved.



# Software Quality Attributes

---

Important software quality attributes are:

- ▶ Safety
- ▶ Security
- ▶ Reliability
- ▶ Resilience
- ▶ Robustness
- ▶ Understandability
- ▶ Testability
- ▶ Adaptability
- ▶ Modularity
- ▶ Complexity
- ▶ Portability
- ▶ Usability
- ▶ Reusability
- ▶ Efficiency
- ▶ Learnability



# Software Quality Assurance

---

- ▶ To ensure quality in a software product, an organization must have a three-prong approach to quality management:
  - ▶ Organization-wide policies, procedures and standards must be established.
  - ▶ Project-specific policies, procedures and standards must be tailored from the organization-wide templates.
  - ▶ Quality must be controlled; that is, the organization must ensure that the appropriate procedures are followed for each project
- ▶ Standards exist to help an organization draft an appropriate software quality assurance plan.
  - ▶ ISO 9000-3
  - ▶ ANSI/IEEE standards
- ▶ External entities can be contracted to verify that an organization is standard-compliant.



Inspiring Excellence

	QA	QC	Testing
<b>Purpose</b>	Setting up adequate processes, introducing the standards of quality to prevent the errors and flaws in the product	Making sure that the product corresponds to the requirements and specs before it is released	Detecting and solving software errors and flaws
<b>Focus</b>	Processes	Product as a whole	Source code and design
<b>What</b>	Prevention	Verification	Detection
<b>Who</b>	The team including the stakeholders	The team	Test Engineers, Developers
<b>When</b>	Throughout the process	Before the release	At the testing stage or along with the development process

# SQA Activities

- ▶ Applying technical methods
  - ▶ To help the analyst achieve a high quality specification and a high quality design
- ▶ Testing Software
  - ▶ A series of test case design methods that help ensure effective error detection
- ▶ Enforcing standards
- ▶ Measurement
  - ▶ Track software quality and asses the ability of methodological and procedural changes to improve software quality
- ▶ Record keeping and reporting
  - ▶ Provide procedures for the collection and dissemination of SQA information



Inspiring Excellence

# SOFTWARE ENGINEERING

CSE 470 –Control Flow Graph (Path  
Based Testing )

BRAC University



# Control Flow Graph: Introduction

---

- ▶ An abstract representation of a structured program/function/method.
- ▶ Consists of two major components:
  - ▶ Node:
    - ▶ Represents a stretch of sequential code statements with no branches.
  - ▶ *Directed Edge* (also called *arc*):
    - ▶ Represents a branch, alternative path in execution.
- ▶ Path:
  - ▶ A collection of Nodes linked with *Directed Edges*.

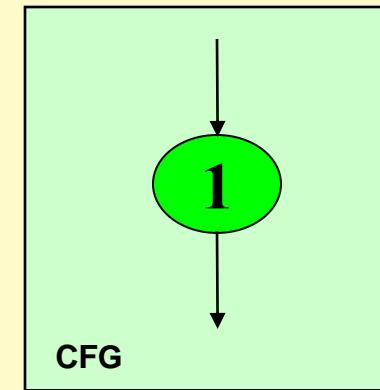
# Notation Guide for CFG

- ▶ A CFG should have:
  - ▶ 1 entry arc (known as a directed edge, too).
  - ▶ 1 exit arc.
- ▶ All nodes should have:
  - ▶ At least 1 entry arc.
  - ▶ At least 1 exit arc.
- ▶ **A Logical Node** that does not represent any actual statements can be added as a joining point for several incoming edges.
  - ▶ Represents a logical closure.
  - ▶ Example:
    - ▶ Node 4 in the if-then-else example in next slides

# Simple Examples

```
Statement1;  
Statement2;  
Statement3;  
Statement4;
```

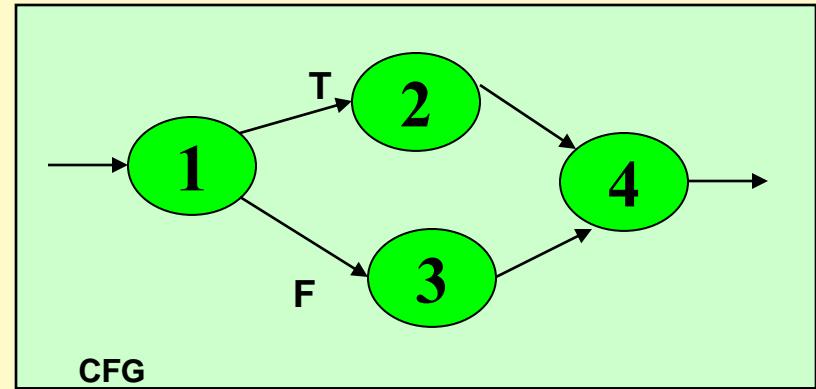
Can be represented as **one** node as there is no branch.



# More Examples

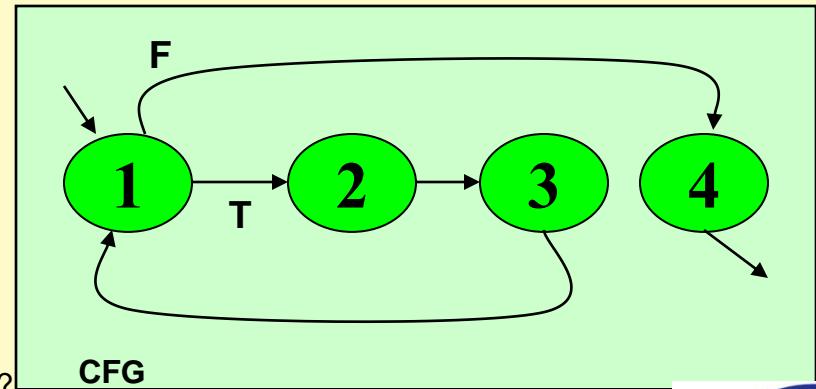
```
if x > 0 then  
    Statement1;  
else  
    Statement2;
```

1  
2  
3



```
while x < 10 {  
    Statement1;  
    x++; }
```

1  
2  
3

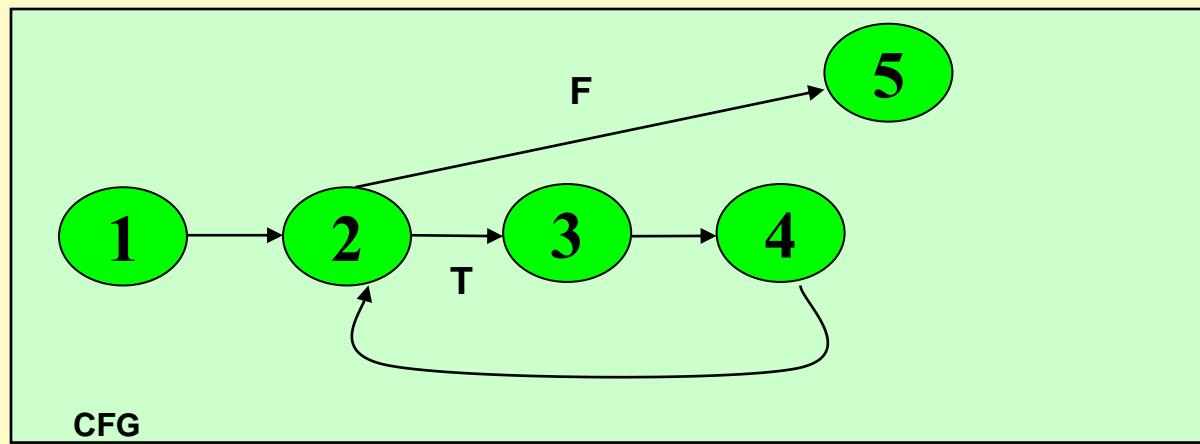


Question: Why is there a node 4 in both CFGs?

Answer: A logical node

# More Examples

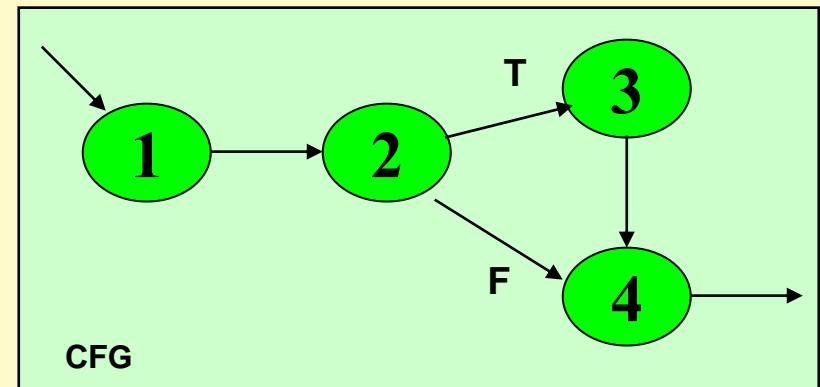
```
1           2           4  
for (int I = 0; I < 10 ; I ++) {  
    Statement1;  
    Statement2; } 3  
    Statement3;  
}  
Statement4; } 5
```



# Combined Examples

```
Statement1;  
Statement2;  
  
if x < 10 then  
    Statement3;  
  
Statement4;
```

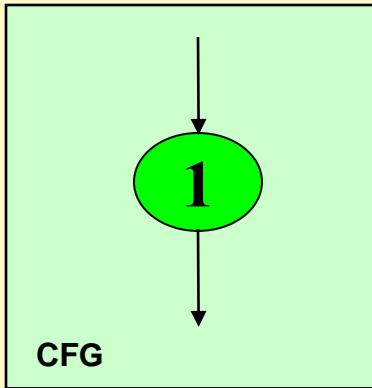
1  
2  
3  
4



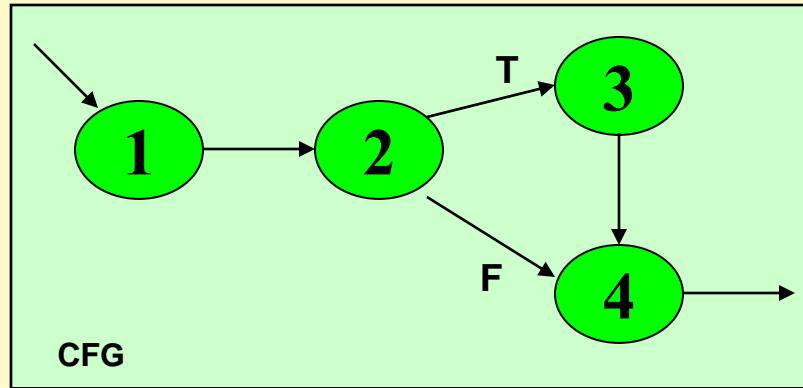
# Number of Paths through CFG

- ▶ Given a program, how do we exercise all statements and branches at least once?
- ▶ Translating the program into a CFG, an equivalent question is:
  - ▶ Given a CFG, how do we cover all arcs and nodes at least once?
- ▶ Since a path is a trail of nodes linked by arcs, this is similar to ask:
  - ▶ Given a CFG, what is the set of paths that can cover all arcs and nodes?

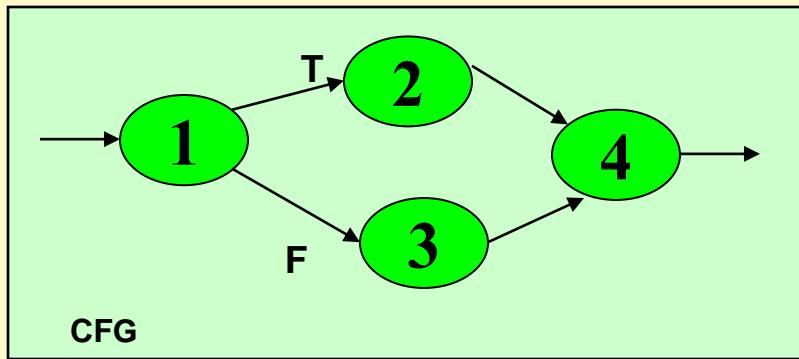
# Example



- ▶ Only **one** path is needed:
  - ▶ [ 1 ]



- **Two** paths are needed:
  - [ 1 - 2 - 4 ]
  - [ 1 - 2 - 3 - 4 ]



- **Two** paths are needed:
  - [ 1 - 2 - 4 ]
  - [ 1 - 3 - 4 ]

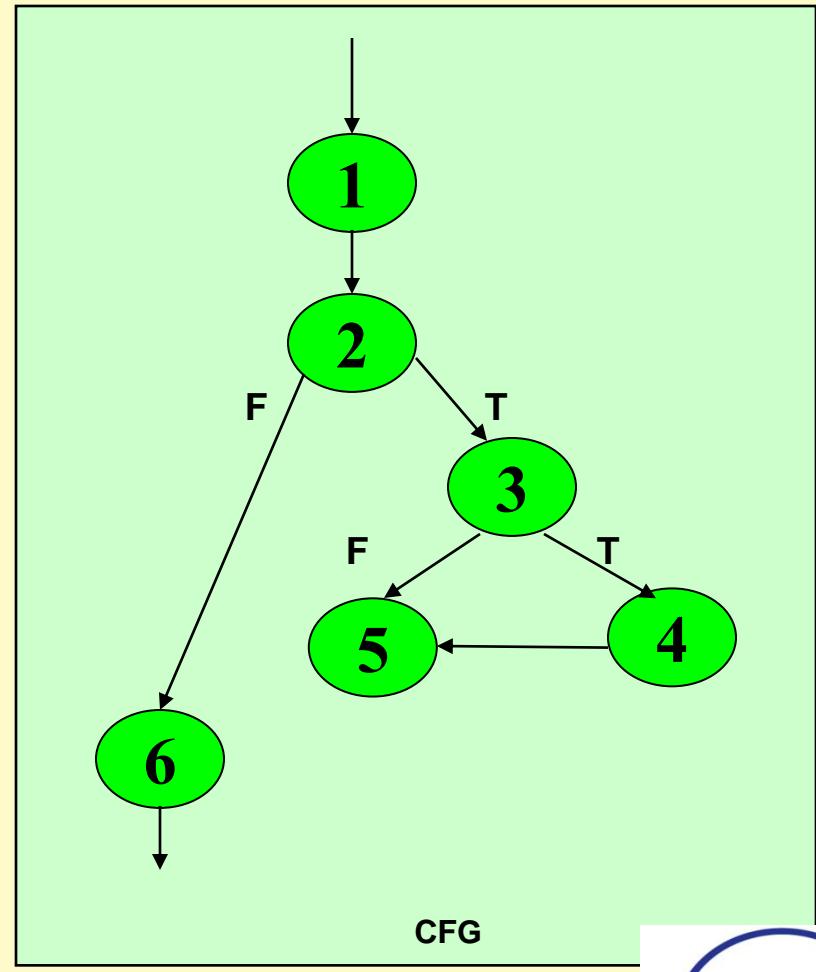
# White Box Testing: Path Based

- ▶ A generalized technique to find out the number of paths needed (known as *cyclomatic complexity*) to cover all arcs and nodes in CFG.
- ▶ Steps:
  1. Draw the CFG for the code fragment.
  2. Compute the *cyclomatic complexity number C*, for the CFG.
  3. Find at most **C** paths that cover the nodes and arcs in a CFG, also known as **Basic Paths Set**;
  4. Design test cases to force execution along paths in the **Basic Paths Set**.

# Path Based Testing: Step 1

```
min = A[0];  
I = 1;  
  
while (I < N) {  
    if (A[I] < min)  
        min = A[I];  
    I = I + 1;  
}  
print min
```

1  
2  
3  
4  
5  
6



# Path Base Testing: Step 2

I. The complexity  $M$  is then defined as

$$M = R + I,$$

where  $R$  = the number of regions in the graph.

2. The complexity  $M$  is then defined as

$$M = P + I,$$

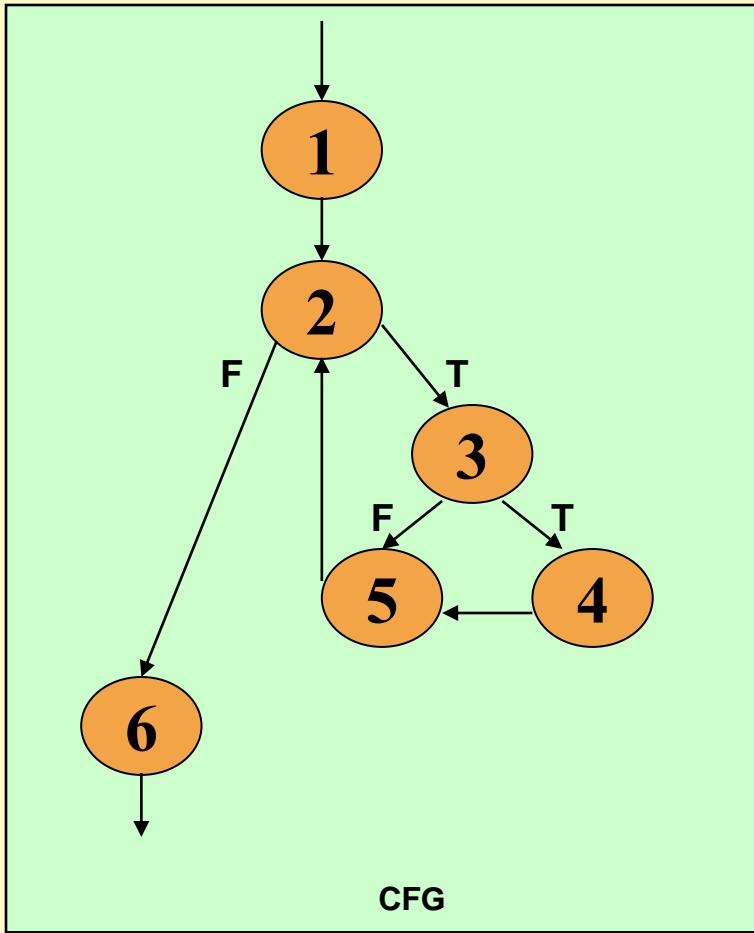
where  $P$  = the number of predicate nodes in the graph.

3. The complexity  $M$  is then defined as

$$M = E - N + 2P, \text{ where}$$

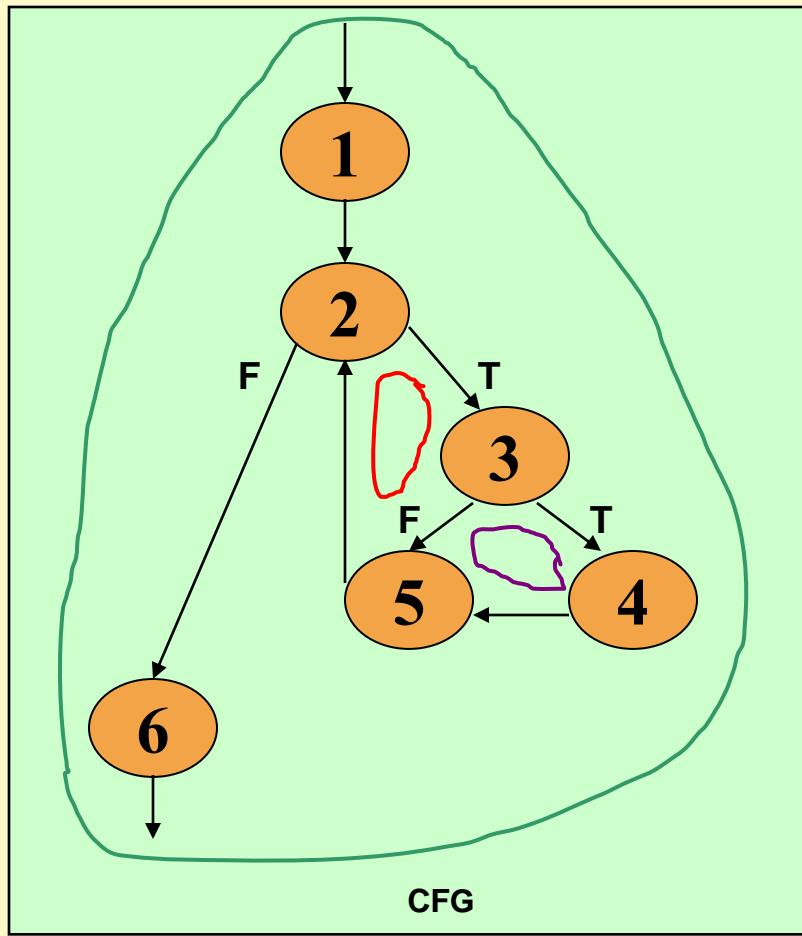
- ▶  $E$  = the number of edges of the graph.
- ▶  $N$  = the number of nodes of the graph.
- ▶  $P$  = the number of connected components.

# Path Base Testing: Step 2



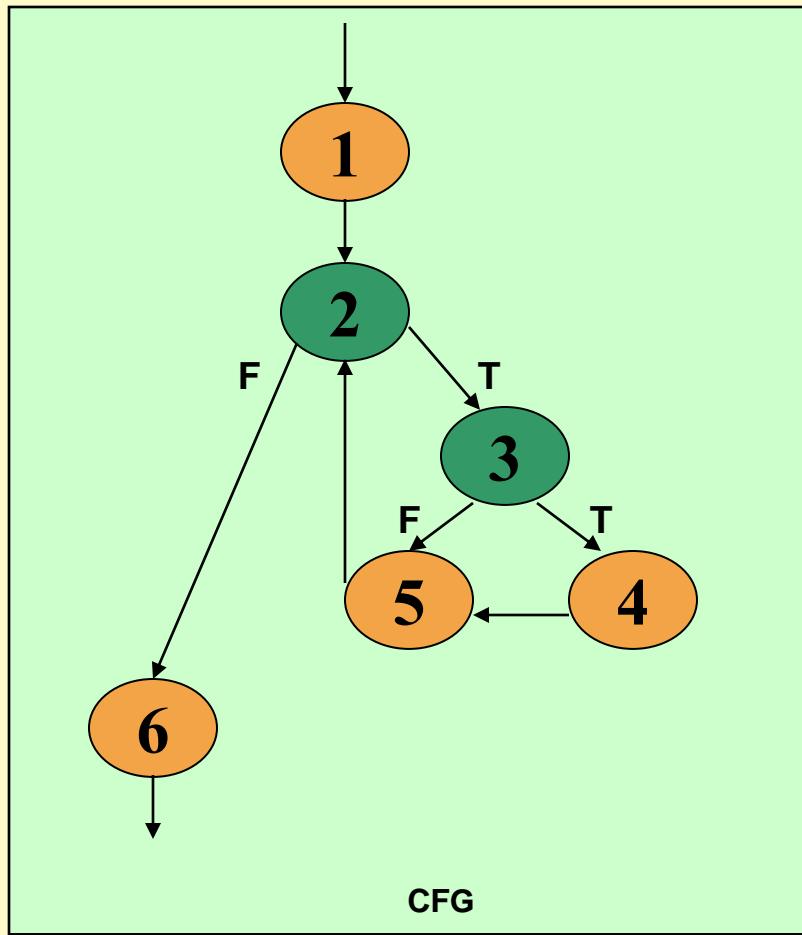
- ▶ **Cyclomatic complexity =**
  - ▶ The number of 'regions' in the graph( R ) + 1
  - ▶

# Path Base Testing: Step 2



- ▶ Cyclomatic complexity,  $M =$ 
  - ▶ The number of 'regions' in the graph(  $R$  ) + 1
  - ▶  $= 2 + 1 = 3$

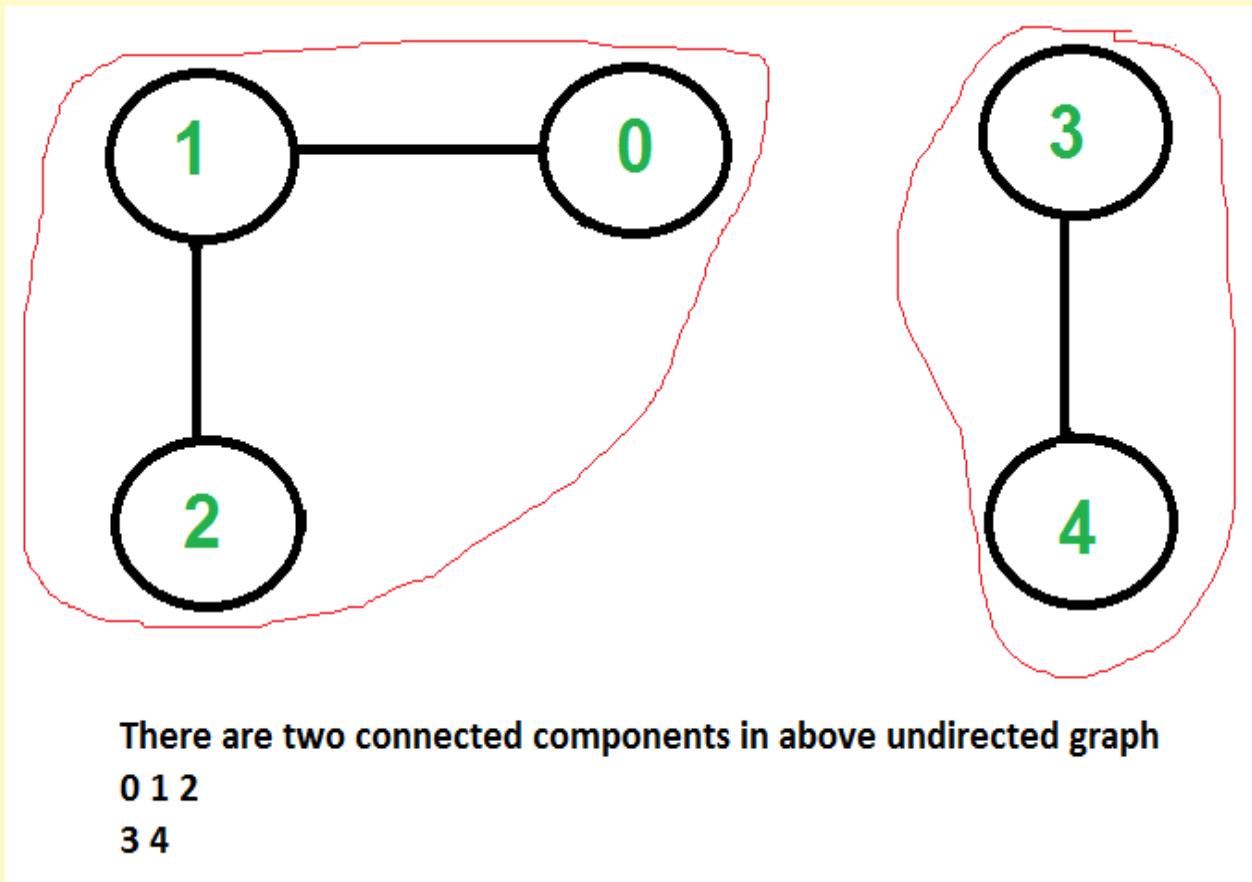
# Path Base Testing: Step 2



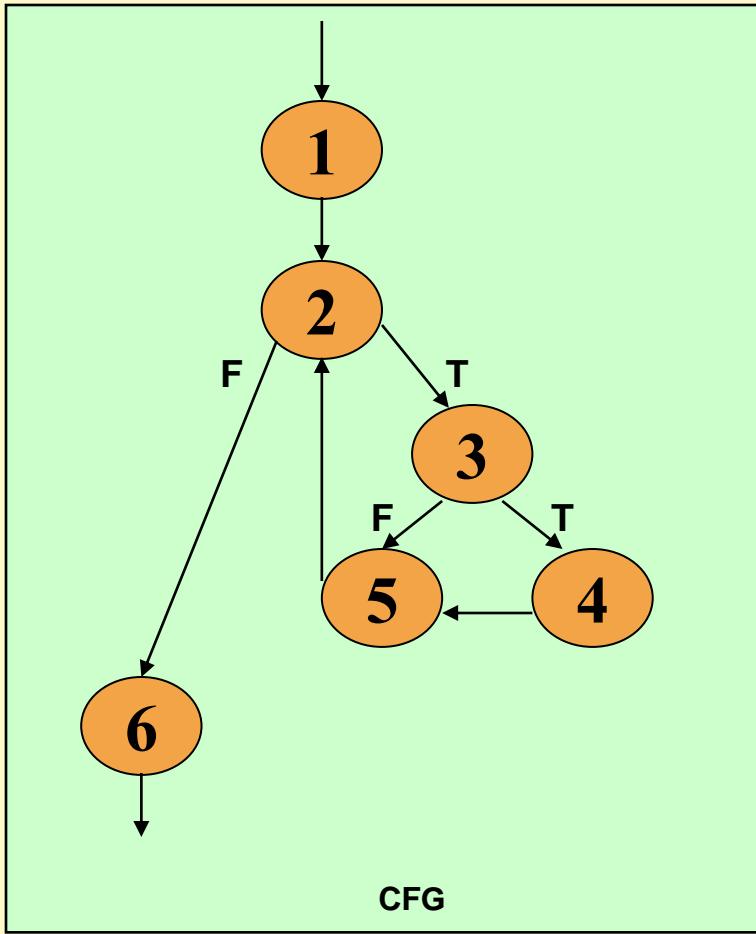
- ▶  $M = \text{Number of 'predicate' node (P)} + 1$
- ▶ In this example:
  - ▶ Predicates,  $P = 2$ 
    - ▶ (Node 2 and 3)
  - ▶ Cyclomatic Complexity,  $M = 2 + 1 = 3$

# Path Base Testing: Step 2

Connected Components in a Graph



# Path Base Testing: Step 2

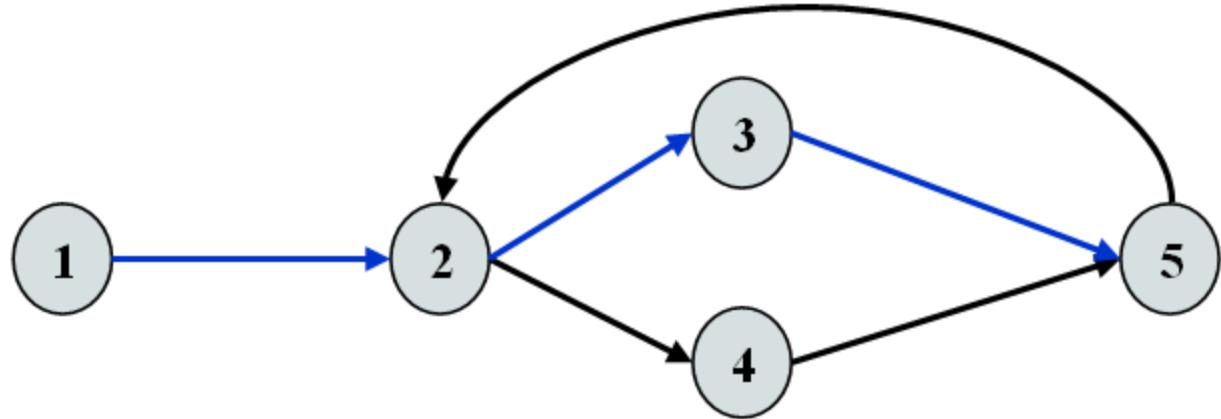


- ▶ Cyclomatic complexity,  $M = E - N + 2P$
- ▶ E, edges = 7
- ▶ N, nodes = 6
- ▶ P, connected components = 1
- ▶  $= 7 - 6 + (2 \times 1)$
- ▶ = 3

# Path Base Testing: Step 3

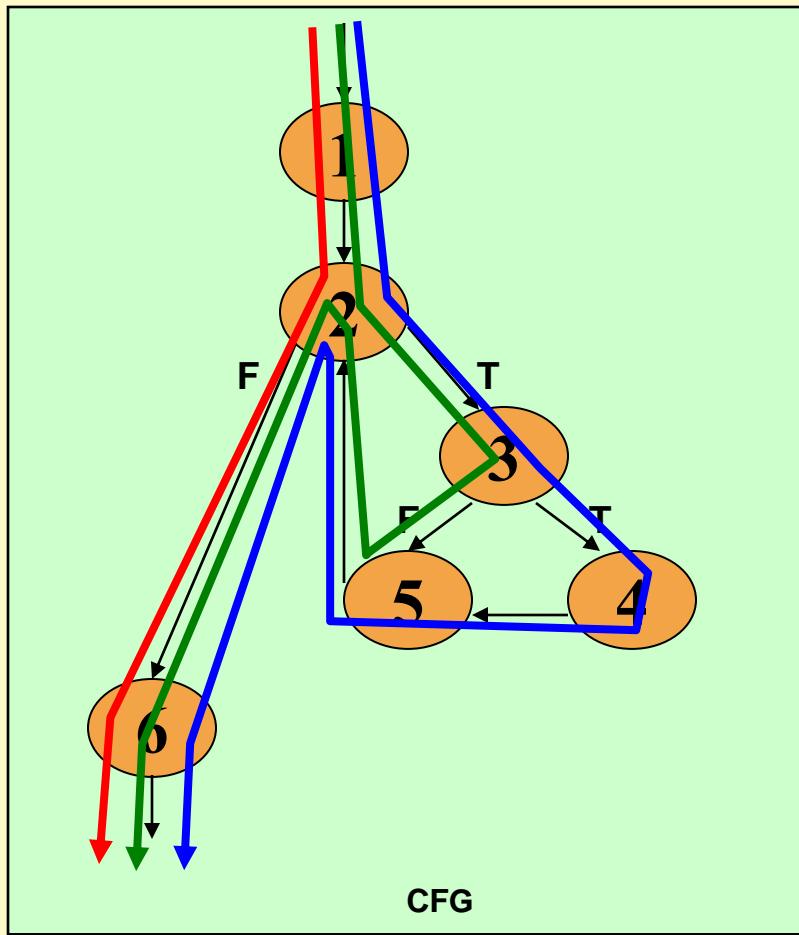
- ▶ **Independent path:**
  - ▶ An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
  - ▶ **Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
  - ▶ The objective is to cover all statements in a program by independent paths.
- ▶ The number of independent paths to discover  $\leq$  Cyclomatic complexity number,  $M$
- ▶ The set of Independent paths is called **Basic Path Set**

# Example



- ▶  $M = \text{Regions} + I = 2 + 1 = 3$
- ▶ 1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.
- ▶ Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.
- ▶ The number of independent paths therefore can vary according to the order we identify them.

# Path Base Testing: Step 3



- ▶ Cyclomatic complexity = 3.
- ▶ Need at most 3 independent paths to cover the CFG.
- ▶ In this example:
  - ▶ [ 1 - 2 - 6 ]
  - ▶ [ 1 - 2 - 3 - 5 - 2 - 6 ]
  - ▶ [ 1 - 2 - 3 - 4 - 5 - 2 - 6 ]

# Path Base Testing: Step 4

- ▶ Prepare a test case for each independent path.

- ▶ In this example:

- ▶ Path: [ 1 – 2 – 6 ]
  - ▶ Test Case: A = { 5,...}, N = 1
  - ▶ Expected Output: 5

```
min = A[0];  
I = 1;  
  
while (I < N) {  
    if (A[I] < min)  
        min = A[I];  
    I = I + 1;  
}  
print min
```

1  
2  
3  
4  
5  
6

Try to verify that the test cases actually force execution along a desired path.

# CSE470:Software Engineering

---

video lecture series produced by:

A.M.Esfar-E-Alam

Afrina Khatun

Dr.Muhammad Zavid Parvez

Hossain Arif

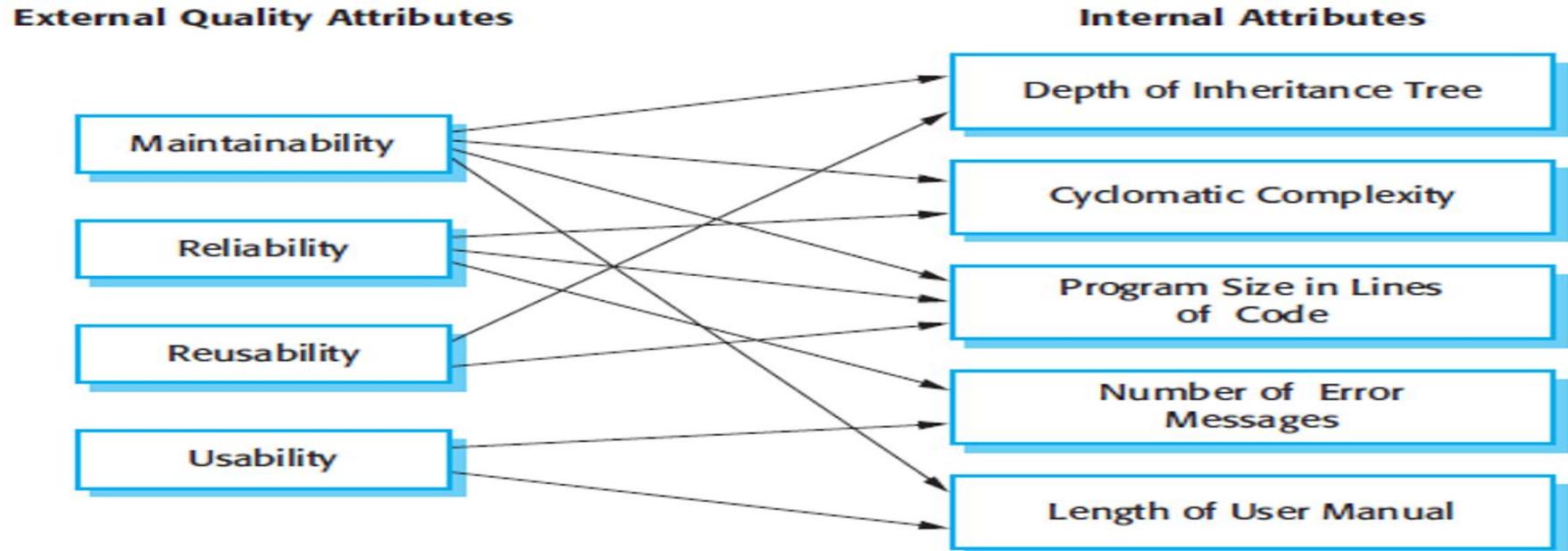
# **Software measurement and metrics**

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- There are few established standards in this area

# Software metric

- Any type of measurement which relates to a software system, process or related documentation
- Lines of code in a program, the Fog index, number of person-days required to develop a component.
- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components.

# Relationships between internal and external software



# **Product metrics**

- A quality metric should be a predictor of product quality.
- Classes of product metric
- Dynamic metrics which are collected by measurements made of a program in execution;
- Static metrics which are collected by measurements made of the system representations;
- Dynamic metrics help assess efficiency and reliability
- Static metrics help assess complexity, understandability and maintainability.

# Fan-in/Fan-out, Length of code

## → Fan-in/Fan-out

- ◆ Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.

## → Length of code

- ◆ This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components

# **CYCLOMATIC COMPLEXITY**

- Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors.
- It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.
- Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand. It can be represented using the below formula:

$$M = E - N + 2P, \text{ where}$$

*E* = the number of edges of the graph.

*N* = the number of nodes of the graph.

*P* = the number of connected components.

The complexity  $M$  is then defined as

$$M = R + 1,$$

where  $R$  = the number of regions in the graph.

The complexity  $M$  is then defined as

$$M = P + 1,$$

where  $P$  = the number of predicate nodes in the graph.

These two formulas are easy to use

# SPECIALIZATION INDEX (SIX)

- The **Specialization Index metric** measures the extent to which subclasses override their ancestors classes. This **index** is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class
- The metric provides a percentage, where the class contains at least one operation. For a root class, the specialization indicator is zero. Nominal range is between 0 % and 120 %.

The ... variable	represents the ...
DIT	depth of inheritance
NMA	the number of operations added to the inheritance
NMI	the number of inherited operations
NMO	the number of overloaded operations

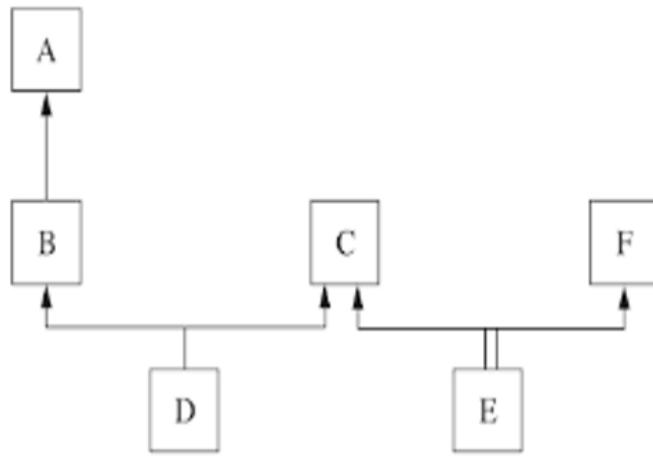
NMO – Number of Overridden Methods not Overloaded.

Example:

$$SIX = \frac{3 \times 4}{3 + 4 + 3} \times 100 = 120$$

$$\text{DIT}(D) = 2$$

$$\text{DIT}(E) = 1$$



<b>The ... variable</b>	<b>represents the ...</b>
DIT	depth of inheritance
NMA	the number of operations added to the inheritance
NMI	the number of inherited operations
NMO	the number of overloaded operations

```
Class Person{
```

```
    void read();
```

```
    void display();
```

```
}
```

```
Class Student extends Person{
```

```
    void read();
```

```
    void display();
```

```
    Void getAverage();
```

```
}
```

```
Class GraduateStudent extends Student{
```

```
    void read();
```

```
    void display();
```

```
Void workStatus();
```

**[(4/4)\*100]**

$$SIX = \frac{NMO \times DIT}{NMO + NMA + NMI}$$

$$SIX = \frac{2 \times 2}{2 + 1 + 1}$$

-> 100%

# DEFECT REMOVAL EFFICIENCY

- A **defect** is found when the application does not conform to the requirement specification.
- A mistake in coding is called **Error**
- An average DRE score is usually around 85% across a full testing program.
- $DRE = E / (E + D)$  where:
- **E** is the number of errors found before delivery of the software to the end-user
- **D** is the number of defects found after delivery.
- We found 100 defects during the testing phase and then later, say within 90 days after software release (in production), found five defects,
- $DRE = 100/(100+5) = 95.2\%$

**Software Engineering**

**COCOMO**

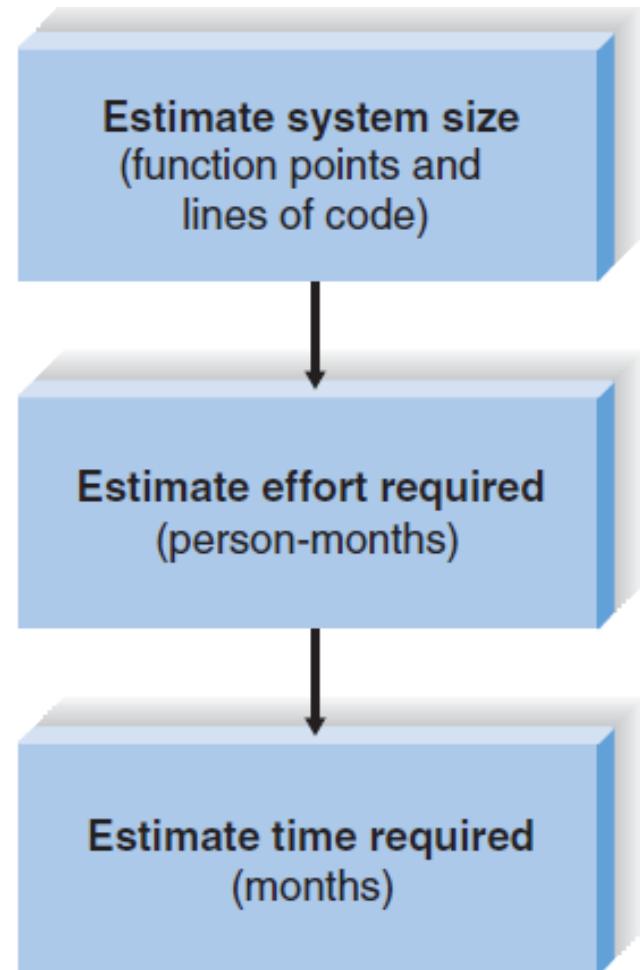
**CONSTRUCTIVE**

**COST**

**MODEL**

# The function point approach

- Used to
  - Estimate the size of the new system
  - The effort that will be required to complete the system
  - The time the project will require
- It is a three-step process.



# **Step 1: Estimate System Size**

- A function point is a measure of program size that is based on the system's number and complexity of inputs, outputs, queries, files, and program interfaces.
- The project manager records the total number of each component that the system will include, and then breaks down the number to show the number of components that have low, medium, and high complexity.
- To create a more realistic size for the project, a number of additional system factors such as end-user efficiency, reusability, and data communications are assessed in terms of their effect on the project's complexity.

## System Components:

Description	Total Number	Complexity			Total
		Low	Medium	High	
Inputs	<u>6</u>	<u>3</u> × 3	<u>2</u> × 4	<u>1</u> × 6	<u>23</u>
Outputs	<u>19</u>	<u>4</u> × 4	<u>10</u> × 5	<u>5</u> × 7	<u>101</u>
Queries	<u>10</u>	<u>7</u> × 3	<u>0</u> × 4	<u>3</u> × 6	<u>39</u>
Files	<u>15</u>	<u>0</u> × 7	<u>15</u> × 10	<u>0</u> × 15	<u>150</u>
Program Interfaces	<u>5</u>	<u>1</u> × 5	<u>0</u> × 7	<u>2</u> × 10	<u>25</u>
<b>Total Unadjusted Function Points (TUFFP):</b>					<u>338</u>
					/

Overall System:

Data communications	<u>3</u>
Heavy use configuration	<u>0</u>
Transaction rate	<u>0</u>
End-user efficiency	<u>0</u>
Complex processing	<u>0</u>
Installation ease	<u>0</u>
Multiple sites	<u>0</u>
Performance	<u>0</u>
Distributed functions	<u>2</u>
Online data entry	<u>2</u>
Online update	<u>0</u>
Reusability	<u>0</u>
Operational ease	<u>0</u>
Extensibility	<u>0</u>
<b>Total Processing Complexity (PC):</b>	<u>7</u>

(0 = no effect on processing complexity; 3 = great effect on processing complexity)

APC factor has a baseline value of 0.65

**Adjusted Project Complexity (APC):**

$$.65 + (0.01 \times 7) = .72$$

**Total Adjusted Function Points (T AFP):**

$$.72 \text{ (APC)} \times 338 \text{ (T UFP)} = 243 \text{ (T AFP)}$$

# Adjusted Project Complexity

- APC value that ranges from 0.65 for very simple systems to 1.00 for “normal” systems to as much as 1.35 for complex systems.
  - A very simple system that has 200 unadjusted function points would have a size of 130 adjusted function points ( $200 * .65 = 130$ ).
  - If the system with 200 unadjusted function points were very complex, its function point size would be 270 ( $200 * 1.35 = 270$ ).
- In the planning phase, the exact nature of the system has not yet been determined, so it is impossible to know exactly how many inputs, outputs, and so forth will be in the system. It is up to the project manager to make an intelligent guess.

# Lines of code

Convert the number of function points into the lines of code that will be required to build the system.

Language	Approximate Number of Lines of Code per Function Point	
C	130	243 function points.
COBOL	110	COBOL require approximately 26,730 lines of code to write it.
Java	55	
C++	50	
Turbo Pascal	50	Visual Basic take 7290 lines of code.
Visual Basic	30	
PowerBuilder	15	
HTML	15	
Packages (e.g., Access, Excel)	10-40	

Source: Capers Jones, Software Productivity Research, <http://www.spr.com>

# **Estimating Staff and Project size**

**COCOMO** ( Constructive Cost Model ) was proposed by Boehm. This model estimates the total effort in terms of “person-months” of the technical project staff.

Boehm introduces three forms of COCOMO. It can be applied in **three classes of software project**:

**Organic mode** : Relatively simple , small projects with a small team are handled . Such a team should have good application experience to less rigid requirements.

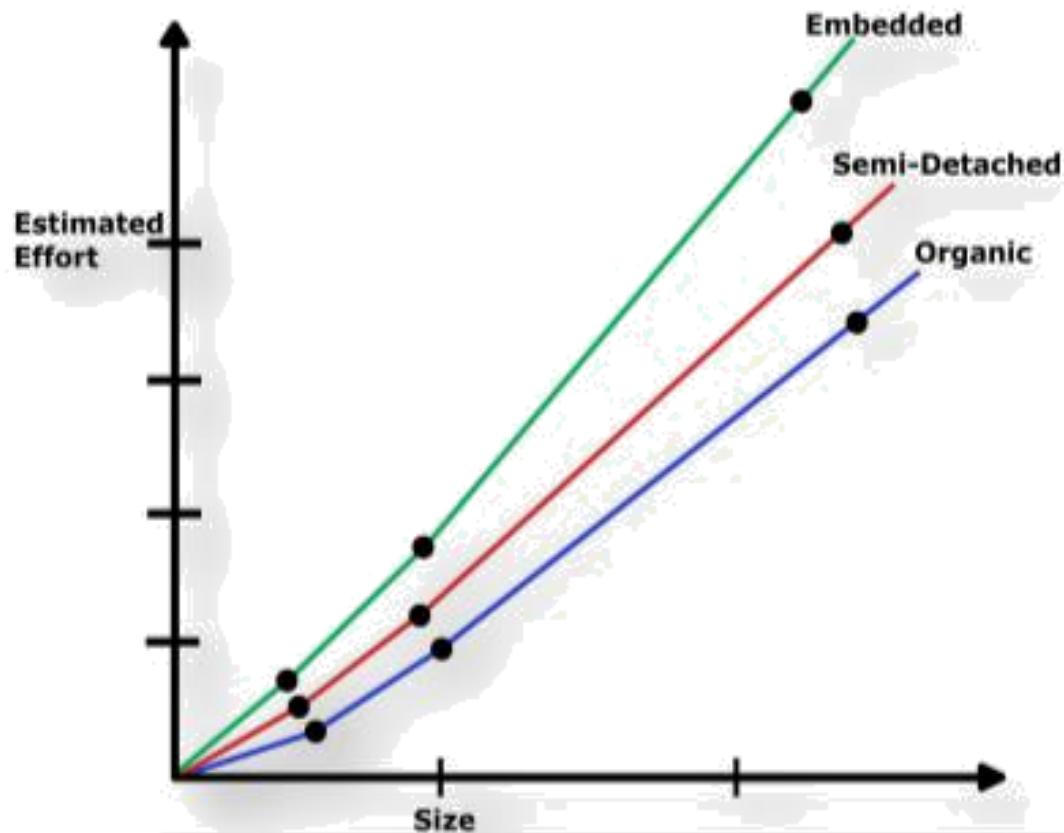
**Semidetached mode:** For intermediate software projects(little complex compared to organic mode projects in terms of size). Projects may have a mix of rigid and less than rigid requirements.

**Embedded mode**: When the software project must be developed within a tight set of hardware and software operational constraints. Ex of complex project: Air traffic control system

# **DEVELOPMENT MODE WITH PROJECT CHARACTERISTICS:**

Development Mode	Project Characteristics			
	Size	Innovation	Deadline	Dev. Environment
<b>ORGANIC</b>	Small	Little	Not Tight	Stable
<b>SEMI-DITACHED</b>	Medium	Medium	Medium	Medium
<b>EMBEDDED</b>	Large	Greater	Tight	Complex Hardware

From the following figure which shows a plot of **estimated effort versus product size**. We can observe that the effort is somewhat superlinear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.



Now the following figure plots the **development time versus the product size in KLOC** can be observed that the development time is a sublinear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately.

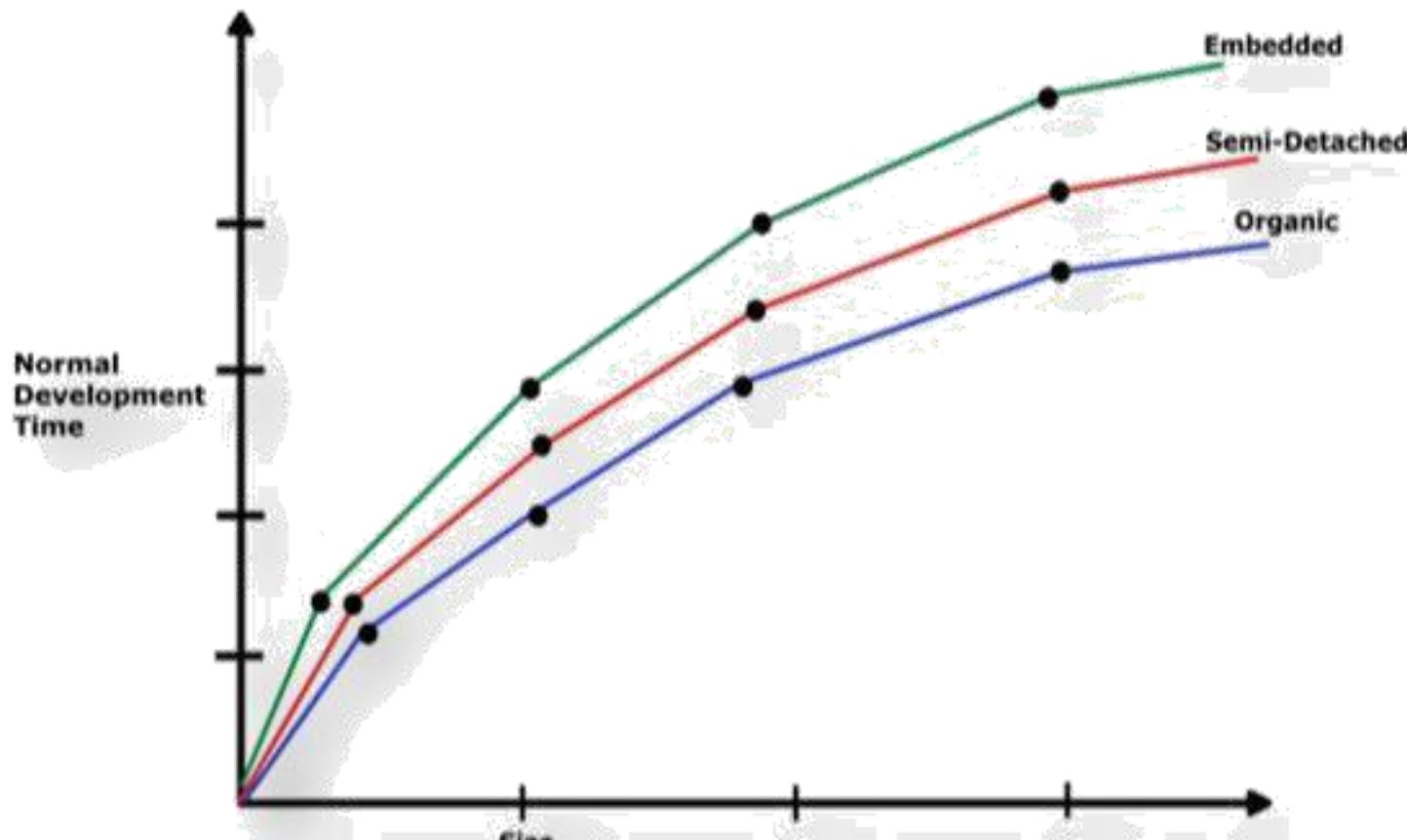


Fig. Development time Verses size

# FORMS OF COCOMO MODEL

1. Basic COCOMO Model
2. Intermediate COCOMO Model
3. Complete/Detailed COCOMO Model

## Basic COCOMO:

Computes software development effort and cost as a function of programme size expressed in terms of Lines Of Code (LOC).

The basic cocomo model takes the following form:

$$E = a_b * ( (KLOC)^{b_b} ) \text{ persons-months}$$

$$D = c_b * ( E )^{d_b} \text{ months}$$

Where

E = Stands for the effort applied in terms of person months

D = Development time in chronological months  
KLOC-Kilo lines of code of the project

a<sub>b</sub>, b<sub>b</sub>, c<sub>b</sub>, d<sub>b</sub> are the coefficients for three modes

The coefficients of  $a_b, b_b, c_b, d_b$  for the three modes are:

Software projects	a <sub>b</sub>	b <sub>b</sub>	c <sub>b</sub>	d <sub>b</sub>
Organic	2.4	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

## **MERITS AND DEMERITS OF BASIC COCOMO MODEL**

### **Advantages:**

1. Basic COCOMO model is good for quick, early, rough order of magnitude estimates of software project.
2. COCOMO is simple, because it requires a small amount of data (LOC) to determine the effort and cost. Hence it is a static single-valued model.

### **Limitations :**

1. The accuracy of this model is limited because it does not consider certain factors for cost estimation of software. These factors are hardware constraints, personal quality and experiences, modern techniques and tools.
2. Not suitable for rapid , recuse based developments.

**Example:** consider a software project using semi-detached mode with 30,000 lines of code. We will obtain estimation for this project as follows:

## (1) Effort estimation

$$E = a_b * (KLOC)^{b_b} \text{ person-months}$$

$$E = 3.0 * (30^{1.12}) , \text{ lines of code} = 30000 = 30000/1000 \text{ KLOC} = 30 \text{ KLOC}$$

$$E = 135 \text{ person-month}$$

## (2) Duration estimation

$$D = (C_b * (E^{d_b})) \text{ months}$$

$$= 2.5 * (135^{0.35})$$

$$D = 14 \text{ months}$$

## (3) Person estimation

$$N = E/D$$

$$= 135/14$$

$$N = 10 \text{ persons approx.}$$

## BASIC COCOMO MODEL

Software project	$a_b$	$b_b$	$c_b$	$d_b$
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Note: you have to memorize the value of these co-efficients

# Intermediate COCOMO:

Computes effort as a function of programme size and a lot of cost drivers that includes subjective assessment of product attributes, hardware attributes , personal attributes and project attributes.

The basic model is extended to consider a set of cost driver attributes grouped into 4 categories:

## **(1) Product Attributes:**

- I. Required software reliability
- II. Size of application software
- III. Complexity of the product

## **(2) Hardware Attributes:**

- I. Run-time performance constraints
- II. Memory constraints
- III. Required turn around time
- IV. Volatility of virtual machine

## **(3) Personal attributes:**

- I. Analyst capability
- II. Software Engineer Capability
- III. Applications Experience
- IV. Programming language experience
- V. Virtual machine Experience

## **(4) Project Attributes:**

- I. Use of software tools
- II. Required development schedule
- III. Application of software engineering methods

Now these 15 attributes get a 6-point scale ranging from “very low” to “extra high”. These ratings can be viewed as:

Very Low, Low, Nominal High, High, Very high, Extra high

**Based on the rating effort multipliers is determined. The product of all effort Multipliers result in “Effort Adjustment Factor” (EAF).**

**The intermediate COCOMO takes the form.**

$$E = a_i * (KLOC^b_i) * EAF \text{ where}$$

E: Effort applied in terms of person-months

KLOC : Kilo lines of code for the project

EAF : It is the effort adjustment factor

**The values of  $a_i$  and  $b_i$  for various class of software projects are:**

Software projects	$a_i$	$b_i$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Memorize the table for maths

**The duration and person estimate is same as in basic COCOMO model i.e;**

$$D = C_b \cdot (E^d_b) \text{ months}$$

i.e; use values of  $c_b$  and  $d_b$

coefficients. [Values of coefficients will not be given. Memorize it.]

$$N = E/D \text{ persons}$$

Software project	$a_b$	$b_b$	$c_b$	$d_b$
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

### **Merits:**

1. This model can be applied to almost entire software product for easy and rough cost estimation during early stage.
2. It can also be applied at the software product component level for obtaining more accurate cost estimation.

### **Limitations:**

1. The effort multipliers are not dependent on phases.
2. A product with many components is difficult to estimate.

## Example:

Consider a project having 30,000 lines of code which in an embedded software with critical area hence reliability is high (EAF=1.15).

The estimation can be

$$E = a_i * (KLOC^{b_i}) * EAF$$

As reliability is high EAF=1.15 (product attribute)

$$a_i = 2.8$$

$b_i = 1.20$  [for embedded software which will not be given]

$$E = 2.8 * (30^{1.20}) * 1.15$$

$$= 191 \text{ person month}$$

$$D = C_b * (E^{d_b}) = 2.5 * (191^{0.32})$$

$$= 13.422 = 14 \text{ months approximately}$$

$$N = E/D$$

$$= 191/14$$

$$N = 14 \text{ persons approx.}$$

Software project	$a_b$	$b_b$	$c_b$	$d_b$
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Software projects	$a_i$	$b_i$
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

## **DETAILED/ADVANCED COCOMO MODEL:**

A major shortcoming of both the basic and Intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics.

The Detailed COCOMO Model differs from the Intermediate COCOMO model in that it uses effort multipliers for each phase of the project. These phase dependent effort multipliers yield better estimates because the cost driver ratings may be different during each phase.

In Advanced COCOMO Model the cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

**Example:** A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following **sub-components**:

- **Database part**
- **Graphical User Interface (GUI) part**
- **Communication part**

Of these, the communication part can be considered as **Embedded software**. The database part could be **Semi-detached software**, and the GUI part **Organic software**. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.



# SOFTWARE ENGINEERING

## CSE 470 – Refactoring Code Smells

## BRAC UNIVERSITY



# What is Refactoring?

---

- ☑ A series of **small** steps, each of which changes the program's **internal structure** without changing its **external behavior** - Martin Fowler
- ☑ Verify no change in external behavior by
  - ☑ Testing
  - ☑ Using the right tool - IDE
  - ☑ Formal code analysis by tool
- Being very, very careful

# What if you hear...

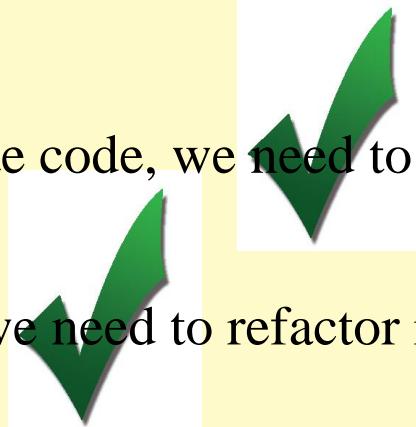
We'll just refactor the code to support logging

Can you refactor the code so that it authenticates against LDAP instead of Database?

We have too much duplicate code, we need to refactor the code to eliminate duplication

This class is too big, we need to refactor it

Caching?



# Why do we Refactor?

---

- Helps us deliver **more business value faster**
- Improves the **design** of our software
- Minimizes *technical debt*
- Keep **development** at *speed*
- To make the software easier to **understand**
- Write for people, not the compiler
- To help find **bugs**

# Readability

---

Which code segment is easier to read?

## Sample 1

```
if (date.Before(Summer_Start) || date.After(Summer_End)){
    charge = quantity * winterRate + winterServiceCharge;
else
    charge = quantity * summerRate;
}
```

## Sample 2

```
if (IsSummer(date)) {
    charge = SummerCharge(quantity);
else
    charge = WinterCharge(quantity);
}
```

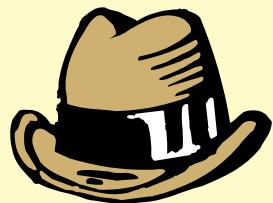
# When should you refactor?

---

- To add **new functionality**
- refactor existing code until you understand it
- refactor the design to make it simple to add
- To find **bugs**
- refactor to understand the code
- For **code reviews**
- immediate effect of code review
- allows for higher level suggestions

# The Two Hats

## Adding Function



- Add new capabilities to the system
- Adds new tests
- Get the test working

## Refactoring



- Does not add any new features
- Does not add tests (but may change some)
- Restructure the code to remove redundancy

# How do we Refactor?

---

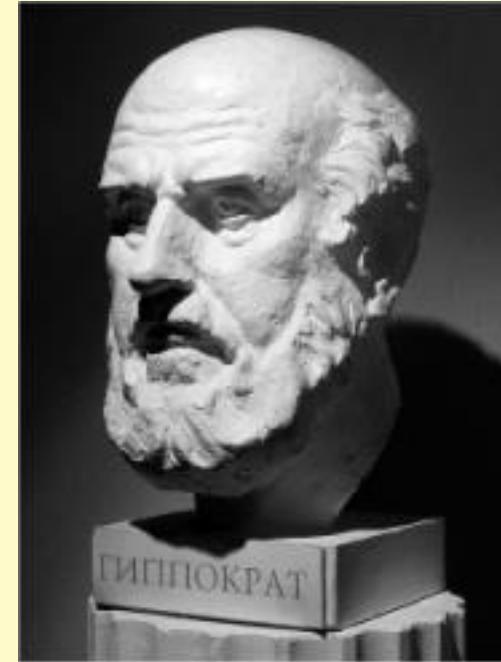
- ✓ We looks for **Code-Smells**
- ✓ Things that we suspect are not quite right or will cause us severe pain if we do not fix

# 2 Piece of Advice before Refactoring

---



Baby Steps



The Hippocratic Oath

First Do No Harm!

# Code Smells?

---

Code Smells identify *frequently* occurring **design problems** in a way that is more *specific or targeted* than general design guidelines (like “loosely coupled code” or “duplication-free code”). - Joshua K

A code smell is a design that duplicates, complicates, bloats or tightly couples code

# A short history of Code Smells

---

- If it stinks, change it!
- Kent Beck coined the term code smell to signify something in code that needed to be changed.



# Common Code Smells

- Inappropriate Naming
- Comments
- Dead Code
- Duplicated code
- Primitive Obsession
- Large Class
- Lazy Class
- Alternative Class with Different Interface
- Long Method
- Long Parameter List
- Switch Statements
- Speculative Generality
- Oddball Solution
- Feature Envy
- Refused Bequest
- Black Sheep

# Code Smell - Inappropriate Naming

---

- Names given to variables (fields) and methods should be clear and meaningful.
- A variable name should say exactly what it is.
- Which is better?
  - private string s;** OR **private string salary;**
- A method should say exactly what it does.
- Which is better?
  - public double calc (double s)**
  - public double calculateFederalTaxes (double salary)**

# Code Smell - Comments

- Comments are often used as deodorant
  - Comments represent a *failure to express an idea in the code*. Try to make your code self-documenting or intention-revealing
  - When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous"
- Remedies:**
- Extract Method
  - Rename Method
  - Introduce Assertion



# Comment: “Grow the Array” smells

```
public class MyList
{
    int INITIAL_CAPACITY = 10;
    bool m_readOnly;
    int m_size = 0;
    int m_capacity;
    string[] m_elements;

    public MyList()
    {
        m_elements = new string[INITIAL_CAPACITY];
        m_capacity = INITIAL_CAPACITY;
    }

    int GetCapacity() {
        return m_capacity;
    }
}
```

```
void AddToList(string element)
{
    if (!m_readOnly)
    {
        int newSize = m_size + 1;
        if (newSize > GetCapacity())
        {
            // grow the array
            m_capacity += INITIAL_CAPACITY;
            string[] elements2 = new string[m_capacity];
            for (int i = 0; i < m_size; i++)
                elements2[i] = m_elements[i];

            m_elements = elements2;
        }
        m_elements[m_size++] = element;
    }
}
```

# Comment Smells Make-over

```
void AddToList(string element)
{
    if (m_READONLY)
        return;
    if (ShouldGrow())
    {
        Grow();
    }
    StoreElement(element);
}
```

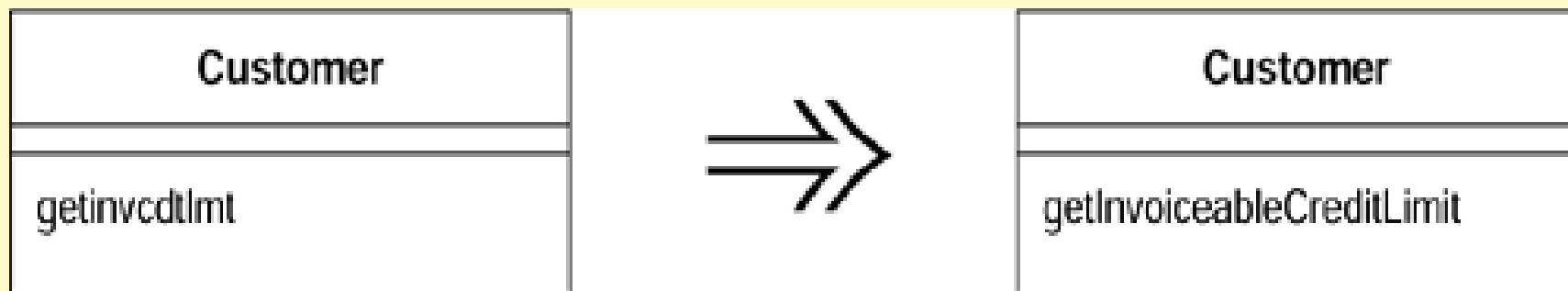
```
private bool ShouldGrow()
{
    return (m_size + 1) > GetCapacity();
}
```

```
private void Grow()
{
    m_capacity += INITIAL_CAPACITY;
    string[] elements2 = new string[m_capacity];
    for (int i = 0; i < m_size; i++)
        elements2[i] = m_elements[i];

    m_elements = elements2;
}

private void StoreElement(string element)
{
    m_elements[m_size++] = element;
}
```

# Rename Method

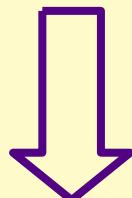


# Extract Method

```
void PrintOwning(double amount){  
    PrintBanner();  
  
    // print details  
    System.Console.Out.WriteLine("name: " + name);  
    System.Console.Out.WriteLine("amount: " + amount);  
}
```

# Extract Method

```
void PrintOwning(double amount){  
    PrintBanner();  
  
    // print details  
    System.Console.Out.WriteLine("name: " + name);  
    System.Console.Out.WriteLine("amount: " + amount);  
}
```



```
void PrintOwning(double amount){  
    PrintBanner();  
    PrintDetails(amount);  
}
```

```
void PrintDetails(double amount){  
    System.Console.Out.WriteLine("name: " + name);  
    System.Console.Out.WriteLine("amount: " + amount);  
}
```

# Introduce Assertion

---

# Introduce Assertion

```
double getExpenseLimit() {  
    // should have either expense limit or a primary project  
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  
        _primaryProject.GetMemberExpenseLimit();  
}
```

# Introduce Assertion

```
double getExpenseLimit() {  
    // should have either expense limit or a primary project  
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  
}    _primaryProject.GetMemberExpenseLimit();
```



```
double getExpenseLimit() {  
    Assert(_expenseLimit != NULL_EXPENSE || _primaryProject != null,  
    "Both Expense Limit and Primary Project must not be null");  
  
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  
        _primaryProject.GetMemberExpenseLimit();  
}
```

# Code Smell - Long Method

- A method is long when it is too hard to quickly comprehend.
- Long methods tend to hide behavior that ought to be shared, which leads to duplicated code in other methods or classes.
- Good OO code is easiest to understand and maintain with shorter methods with good names

## Remedies:

- Extract Method
- Replace Temp with Query
- Introduce Parameter Object
- Preserve Whole Object
- Decompose Conditional



# Long Method Example

```
private String toStringHelper(StringBuffer result)
{
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
    if (!value.equals(""))
        result.append(value);
    Iterator it = children().iterator();
    while (it.hasNext())
    {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);
    }
    result.append("</");
    result.append(name);
    result.append(">");
    return result.toString();
}
```

**Example Html tag:**  
**<name> Jannet Jhonson </name>**

# Long Method Makeover (Extract Method)

```
private String toStringHelper(StringBuffer result)
{
    writeOpenTagTo(result);
    writeValueTo(result);
    writeChildrenTo(result);
    writeEndTagTo(result);
    return result.toString();
}
```

```
private void writeOpenTagTo(StringBuffer result)
{
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
}

private void writeEndTagTo(StringBuffer result)
{
    result.append("</");
    result.append(name);
    result.append(">");
}
```

```
private void writeValueTo(StringBuffer result)
{
    if (!value.equals(""))
        result.append(value);
}

private void writeChildrenTo(StringBuffer result)
{
    Iterator it = children().iterator();
    while (it.hasNext())
    {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);
    }
}
```

# Replace Temp with Query

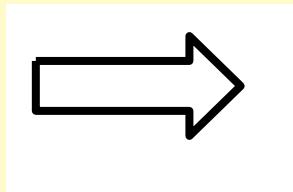
```
Method1(){  
    double basePrice = _quanity * _itemPrice;  
  
    if(basePrice > 1000) {  
        return basePrice * 0.95  
    }  
  
    else{  
        return basePrice*0.98  
    }  
}  
  
Method2(){  
    double basePrice = _quanity * _itemPrice;  
    return basePrice + 100;  
}
```

*What if the basePrice calculation equation changes ??*

*-- We would need to change two lines in the code*

# Replace Temp with Query

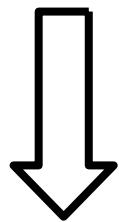
```
Method1(){  
    double basePrice = _quanity * _itemPrice;  
    if(basePrice > 1000) {  
        return basePrice * 0.95  
    }  
    else{  
        return basePrice*0.98  
    }  
}
```



```
Method1(){  
    if(getBasePrice() > 1000) {  
        return getBasePrice() * 0.95;  
    }  
    else {  
        return getBasePrice() * 0.98;  
    }  
}  
  
double getBasePrice() {  
    return _quaniyi * itemPrice;  
}
```

# Replace Temp with Query

```
Method2(){  
    double basePrice = _quantity * _itemPrice;  
    return basePrice + 100;  
}
```



```
double getBasePrice() {  
    return _quanity * itemPrice;  
}
```

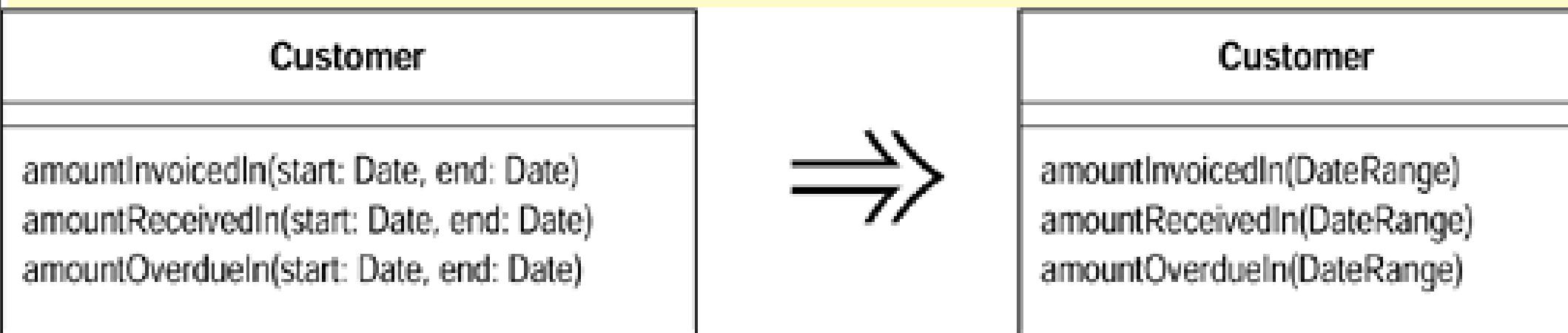


```
Method2(){  
    return getBasePrice() + 100;  
}
```

# Introduce Parameter Object

```
int MethodTooManyParameter (Date start, Date end, int value, string  
month, string yearStart, string yearEnd)  
{  
    // method body  
}
```

# Introduce Parameter Object



```
Class DateRange{  
    Date start;  
    Date end;  
}
```

# Preserve Whole Object

```
int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
```

```
withinPlan = plan.withinRange(low, high);
```

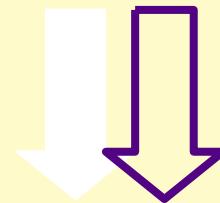
```
daysTempRange(){  
    return someObject;  
}
```

# Preserve Whole Object

```
int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
```

```
withinPlan = plan.withinRange(low, high);
```



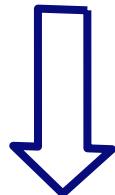
```
withinPlan = plan.withinRange(daysTempRange());
```

# Decompose Conditional

You have a complicated conditional (if-then-else) statement.

*Extract methods from the condition, then part, and else parts.*

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```



```
if (notSummer(date))  
    charge = winterCharge(quantity);  
else charge = summerCharge (quantity);
```

# Example of Conditional Complexity

```
public bool ProvideCoffee(CoffeeType coffeeType)
{
    if(_change < _CUP_PRICE || !AreCupsSufficient || !IsHotWaterSufficient || !IsCoffeePowderSufficient)
    {
        return false;
    }
    if((coffeeType == CoffeeType.Cream || coffeeType == CoffeeType.CreamAndSugar) && !IsCreamPowderSufficient)
    {
        return false;
    }
    if((coffeeType == CoffeeType.Sugar || coffeeType == CoffeeType.CreamAndSugar) && !IsSugarSufficient)
    {
        return false;
    }

    _cups--;
    _hotWater -= _CUP_HOT_WATER;
    _coffeePowder -= _CUP_COFFEE_POWDER;
    if(coffeeType == CoffeeType.Cream || coffeeType == CoffeeType.CreamAndSugar)
    {
        _creamPowder -= _CUP_CREAM_POWDER;
    }
    if(coffeeType == CoffeeType.Sugar || coffeeType == CoffeeType.CreamAndSugar)
    {
        _sugar -= _CUP_SUGAR;
    }

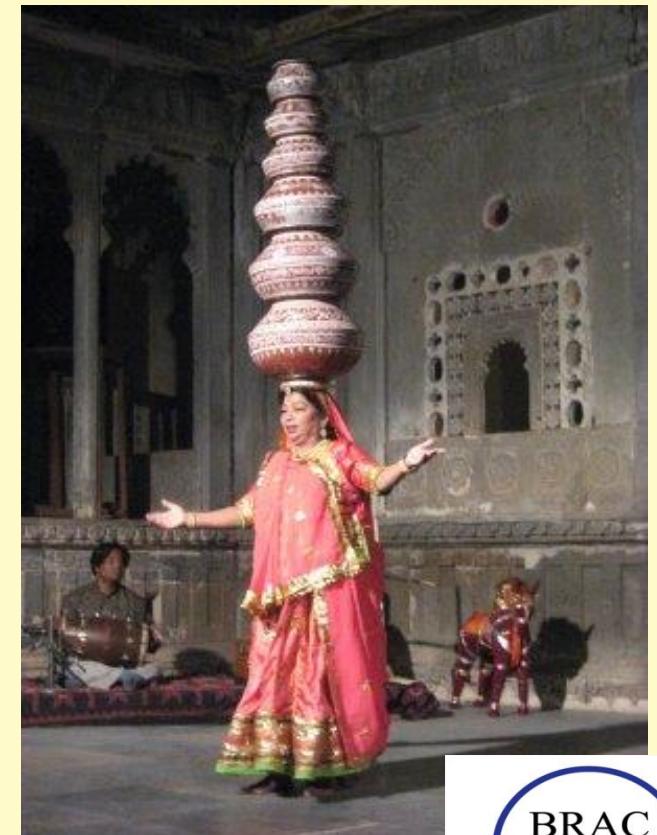
    ReturnChange();
    return true;
}
```

# Code Smell- Long Parameter List

- Methods that take too many parameters produce client code that is awkward and difficult to work with.

## Remedies:

- Introduce Parameter Object
- Replace Parameter with Method
- Preserve Whole Object



# Example

---

```
private void createUserInGroup() {  
    GroupManager groupManager = new GroupManager();  
    Group group = groupManager.create(TEST_GROUP, false,  
        GroupProfile.UNLIMITED_LICENSES, "",  
        GroupProfile.ONE_YEAR, null);  
    user = userManager.create(USER_NAME, group, USER_NAME, "jack",  
        USER_NAME, LANGUAGE, false, false, new Date(),  
        "blah", new Date());  
}
```

# Introduce Parameter Object

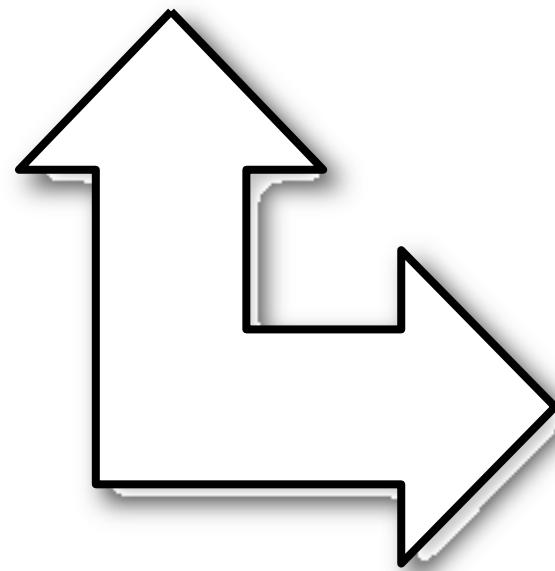
Customer

AmoutInvoicedIn(Date start, Date end)  
AmoutReceivedIn(Date start, Date end)  
AmoutOverdueIn(Date start, Date end)

# Introduce Parameter Object

Customer

AmoutInvoicedIn(Date start, Date end)  
AmoutReceivedIn(Date start, Date end)  
AmoutOverdueIn(Date start, Date end)



Customer

AmoutInvoicedIn(DateRange range)  
AmoutReceivedIn(DateRange range)  
AmoutOverdueIn(DateRange ran

# Replace Parameter with Method

```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel;  
    if (_quantity > 100)  
        discountLevel = 2;  
    else  
        discountLevel = 1;  
    double finalPrice = discountedPrice (basePrice, discountLevel);  
    return finalPrice;  
}
```

```
private double discountedPrice (int basePrice, int discountLevel) {  
    if (discountLevel == 2)  
        return basePrice * 0.1;  
    else  
        return basePrice * 0.05;
```

# Replace Parameter with Method

```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel = getDiscountLevel();  
    double finalPrice = discountedPrice (basePrice, discountLevel);  
    return finalPrice;  
}  
  
private int getDiscountLevel() {  
    if (_quantity > 100) return 2;  
    else return 1;  
}  
  
private double discountedPrice (int basePrice, int discountLevel) {  
    if (getDiscountLevel() == 2) return basePrice * 0.1;  
    else return basePrice * 0.05;  
}
```

# Replace Parameter with Method

```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel = getDiscountLevel();  
    double finalPrice = discountedPrice (basePrice);  
    return finalPrice;  
}
```

```
private double discountedPrice (int basePrice) {  
    if (getDiscountLevel() == 2) return basePrice * 0.1;  
    else return basePrice * 0.05;  
}
```

# Preserve Whole Object

---

```
int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
```

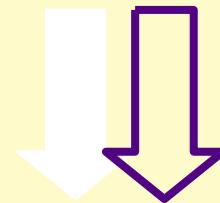
```
withinPlan = plan.withinRange(low, high);
```

# Preserve Whole Object

```
int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
```

```
withinPlan = plan.withinRange(low, high);
```

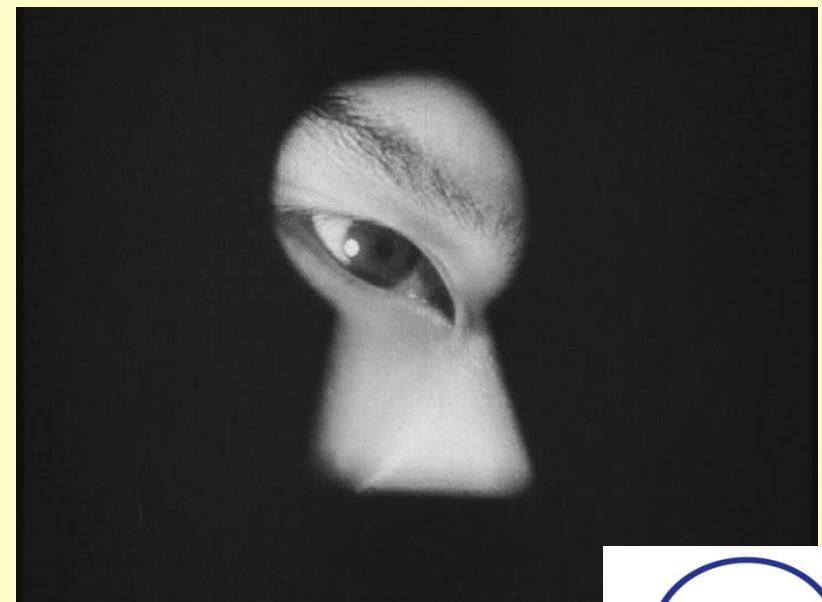


```
withinPlan = plan.withinRange(daysTempRange());
```

# Feature Envy

---

- A method that seems more interested in some other class than the one it is in.
  - Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air.
- 
- Remedies:
  - Move Field
  - Move Method
  - Extract Method



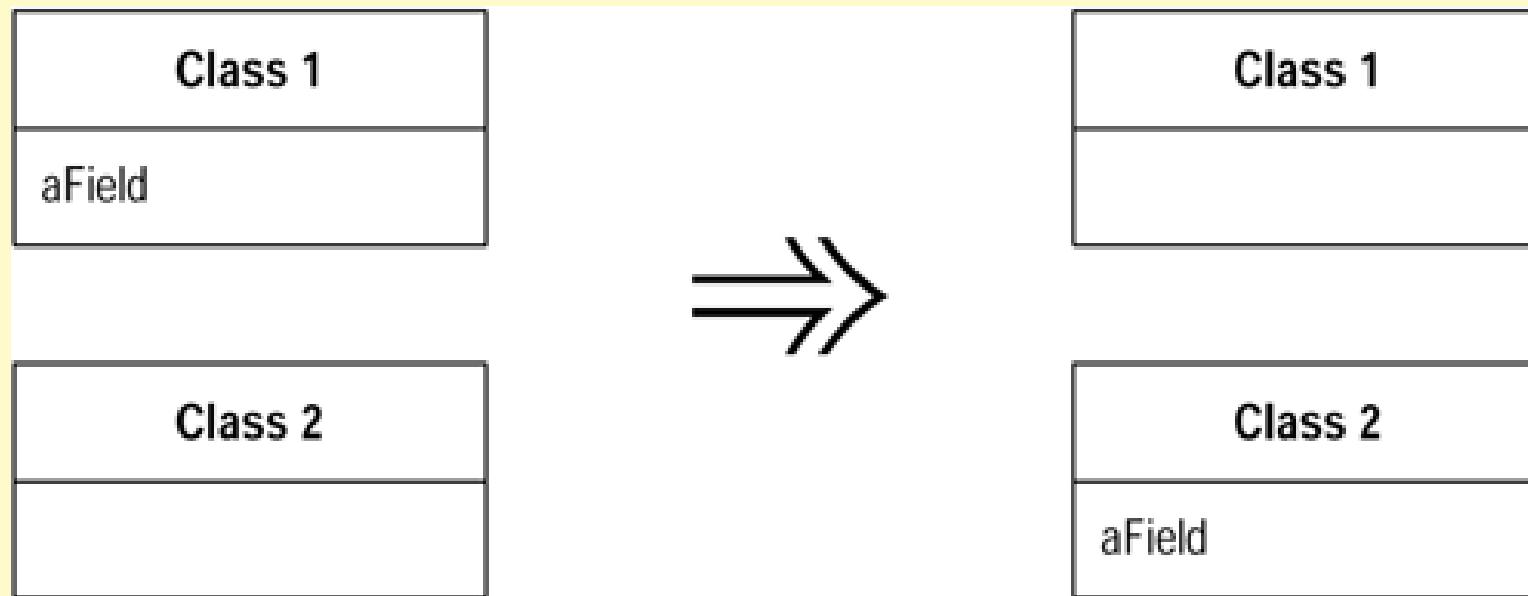
# Example

```
Public class CapitalStrategy{
    double capital(Loan loan)
    {
        if (loan.getExpiry() == NO_DATE && loan.getMaturity() != NO_DATE)
            return loan.getCommitmentAmount() * loan.duration() * loan.riskFactor();

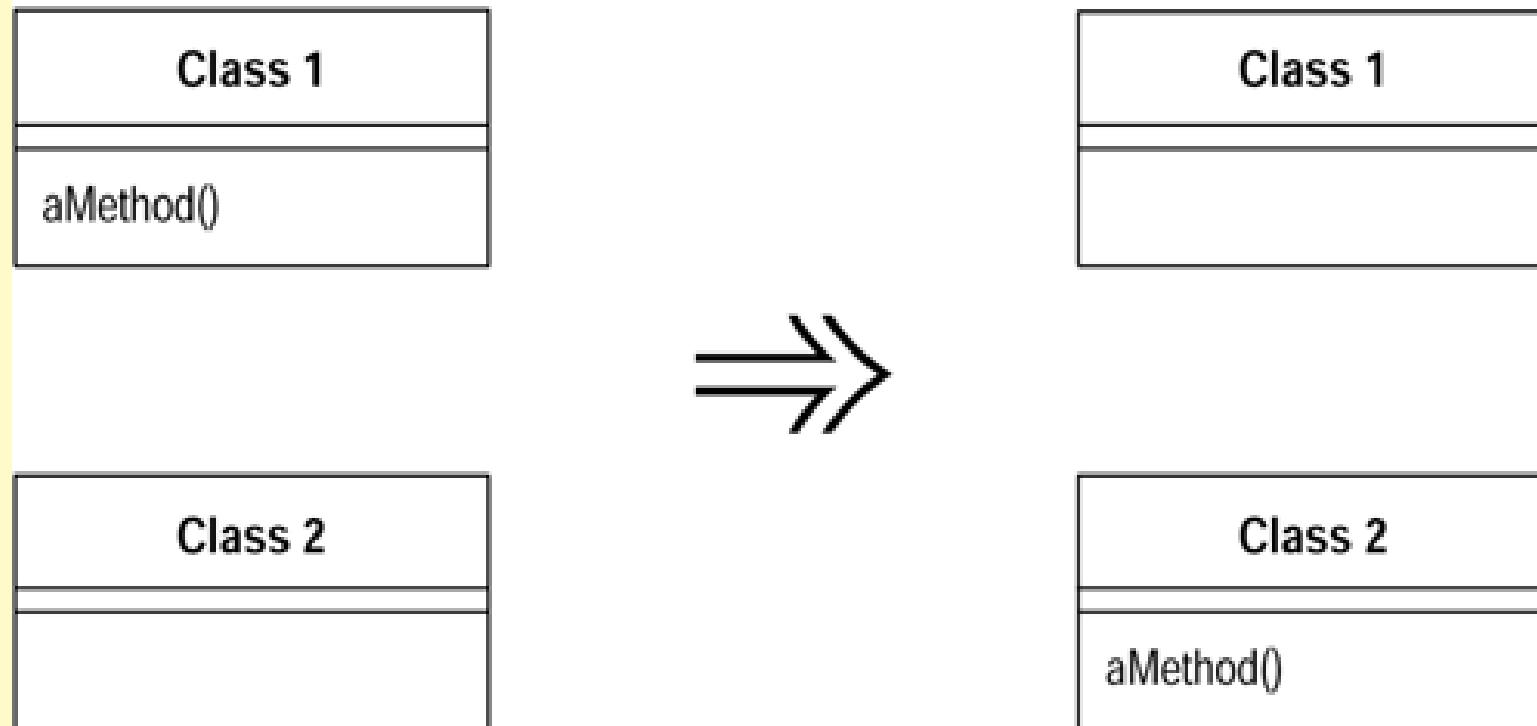
        if (loan.getExpiry() != NO_DATE && loan.getMaturity() == NO_DATE)
        {
            if (loan.getUnusedPercentage() != 1.0)
                return loan.getCommitmentAmount() * loan.getUnusedPercentage() *
loan.duration() * loan.riskFactor();
            else
                return (loan.outstandingRiskAmount() * loan.duration() * loan.riskFactor()) +
                (loan.unusedRiskAmount() * loan.duration() * loan.unusedRiskFactor());
        }

        return 0.0;
    }
}
```

# Move Field



# Move Method



# Duplicated Code

- 
- The *most pervasive and pungent smell* in software
  - There is obvious or blatant duplication
    - Such as copy and paste
  - There are subtle or non-obvious duplications
    - Similar algorithms

## Remedies

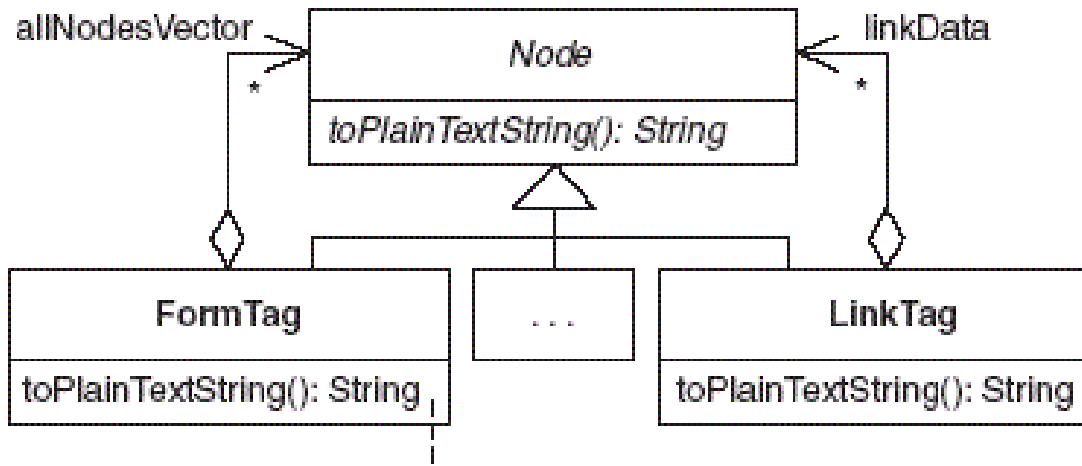
- 
- Extract Method
- Pull Up Field
- Form Template Method

## Substitute Algorithm

# Ctl+C Ctl+V Pattern

```
public static MailTemplate getStaticTemplate(Languages language) {  
    MailTemplate mailTemplate = null;  
    if(language.equals(Languages.English)) {  
        mailTemplate = new EnglishLanguageTemplate();  
    } else if(language.equals(Languages.French)) {  
        mailTemplate = new FrenchLanguageTemplate();  
    } else if(language.equals(Languages.Chinese)) {  
        mailTemplate = new ChineseLanguageTemplate();  
    } else {  
        throw new IllegalArgumentException("Invalid language type specified");  
    }  
    return mailTemplate;  
}  
  
public static MailTemplate getDynamicTemplate(Languages language, String content) {  
    MailTemplate mailTemplate = null;  
    if(language.equals(Languages.English)) {  
        mailTemplate = new EnglishLanguageTemplate(content);  
    } else if(language.equals(Languages.French)) {  
        mailTemplate = new FrenchLanguageTemplate(content);  
    } else if(language.equals(Languages.Chinese)) {  
        mailTemplate = new ChineseLanguageTemplate(content);  
    } else {  
        throw new IllegalArgumentException("Invalid language type specified");  
    }  
    return mailTemplate;  
}
```

# Example Of Obvious Duplication



```
StringBuffer textContents = new StringBuffer();
Enumeration e = allNodesVector.elements()
while (e.hasMoreElements()) {
    Node node = (Node)e.nextElement();
    textContents.append(node.toPlainTextString());
}
return textContents.toString();
```

```
StringBuffer sb = new StringBuffer();
Enumeration e = linkData.elements();
while (e.hasMoreElements()) {
    Node node = (Node)e.nextElement();
    sb.append(node.toPlainTextString());
}
return sb.toString();
```

```
private void AddOrderMaterials(int iOrderId)
{
    if (iOrderType == 1)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 2)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        OrderMaterial oOrderMaterialCream = new OrderMaterial();
        oOrderMaterialCream.MaterialId = 2;
        oOrderMaterialCream.OrderId = iOrderId;
        oOrderMaterialCream.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCream);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 3)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        OrderMaterial oOrderMaterialSugar = new OrderMaterial();
        oOrderMaterialSugar.MaterialId = 3;
        oOrderMaterialSugar.OrderId = iOrderId;
        oOrderMaterialSugar.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialSugar);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 4)
    {

```



# Levels of Duplication

# Literal Duplication

---

Same for loop in 2 places

# Semantic Duplication

```
for(int i : asList(1,3,5,10,15))  
    stack.push(i);
```

v/s

```
for(int i=0;i<5;i++){  
    stack.push(asList(i));  
}
```

1<sup>st</sup>Level - For and For Each Loop

2<sup>nd</sup>Level - Loop v/s Lines repeated

```
stack.push(1); stack.push(3);  
stack.push(5); stack.push(10);  
stack.push(15);
```

v/s

```
for(int i : asList(1,3,5,10,15))  
    stack.push(i);
```

# Data Duplication

---

Some constant declared in 2 classes (test and production)

# Conceptual Duplication

---

2 Algorithm to Sort elements (Bubble sort and Quick sort)

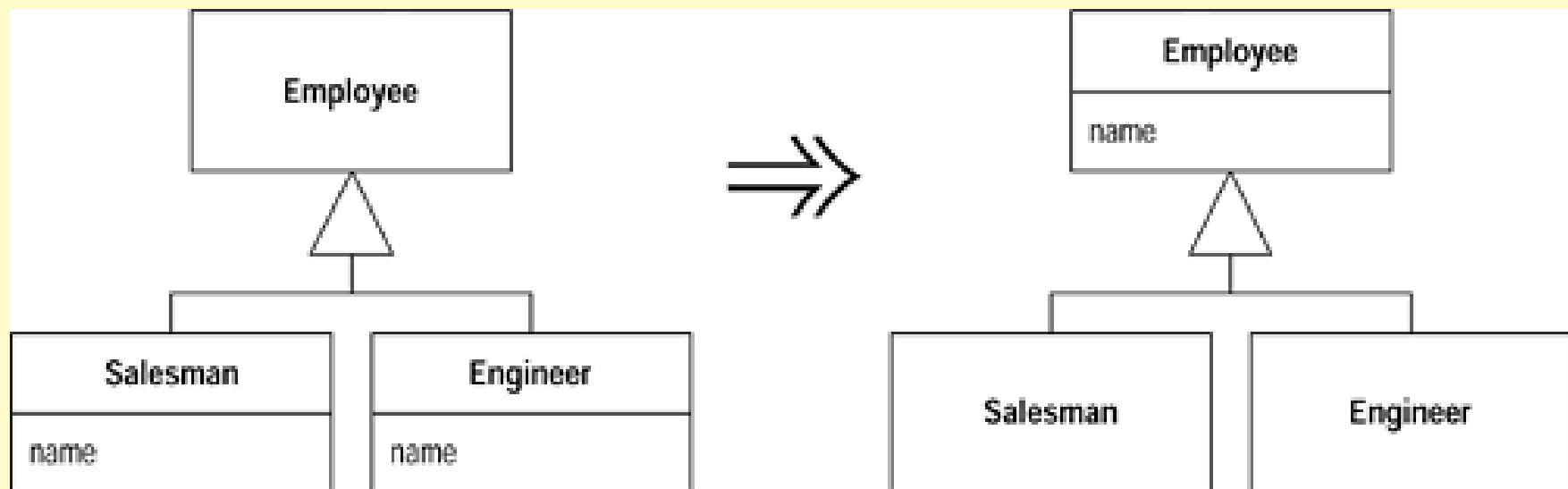
# Logical Steps - Duplication

---

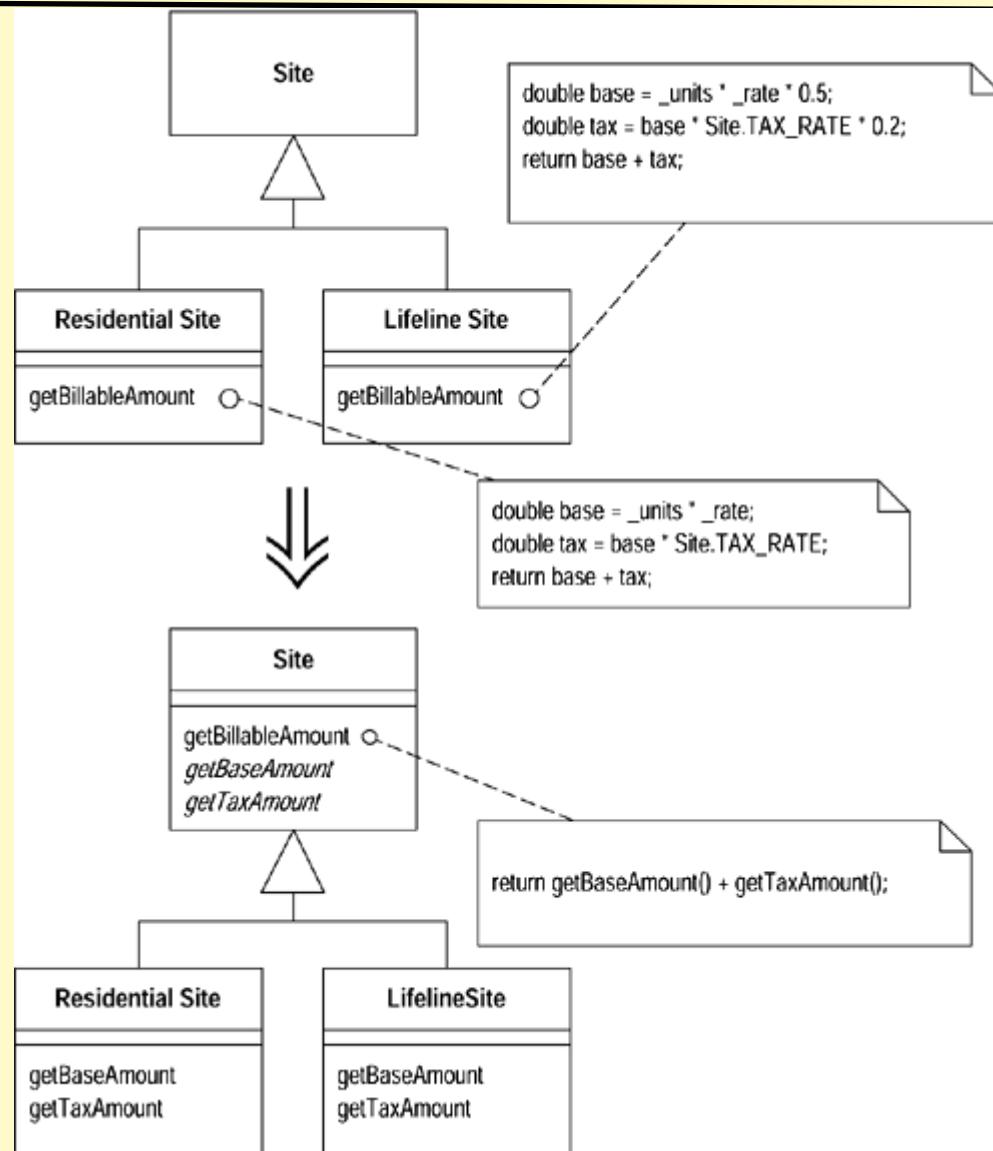
Same set of steps repeat in different scenarios.

Ex: Same set of validations in various points in your applications

# Pull Up Field



# Form Template Method



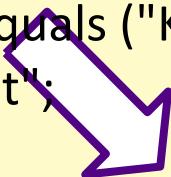
# Substitute Algorithm

```
String foundPerson(String[] people){  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals ("Don")){  
            return "Don";  
        }  
        if (people[i].equals ("John")){  
            return "John";  
        }  
        if (people[i].equals ("Kent")){  
            return "Kent";  
        }  
    }  
    return ""; }
```

# Substitute Algorithm

```
String foundPerson(String[] people){  
  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals ("Don")){  
            return "Don";  
        }  
        if (people[i].equals ("John")){  
            return "John";  
        }  
        if (people[i].equals ("Kent")){  
            return "Kent";  
        }  
    }  
    return "";  
}
```

```
String foundPerson(String[] people){  
    List candidates = Arrays.asList(new String[] {"Don",  
    "John", "Kent"});  
    for (String person : people)  
        if (candidates.contains(person))  
            return person;  
    return "";  
}
```

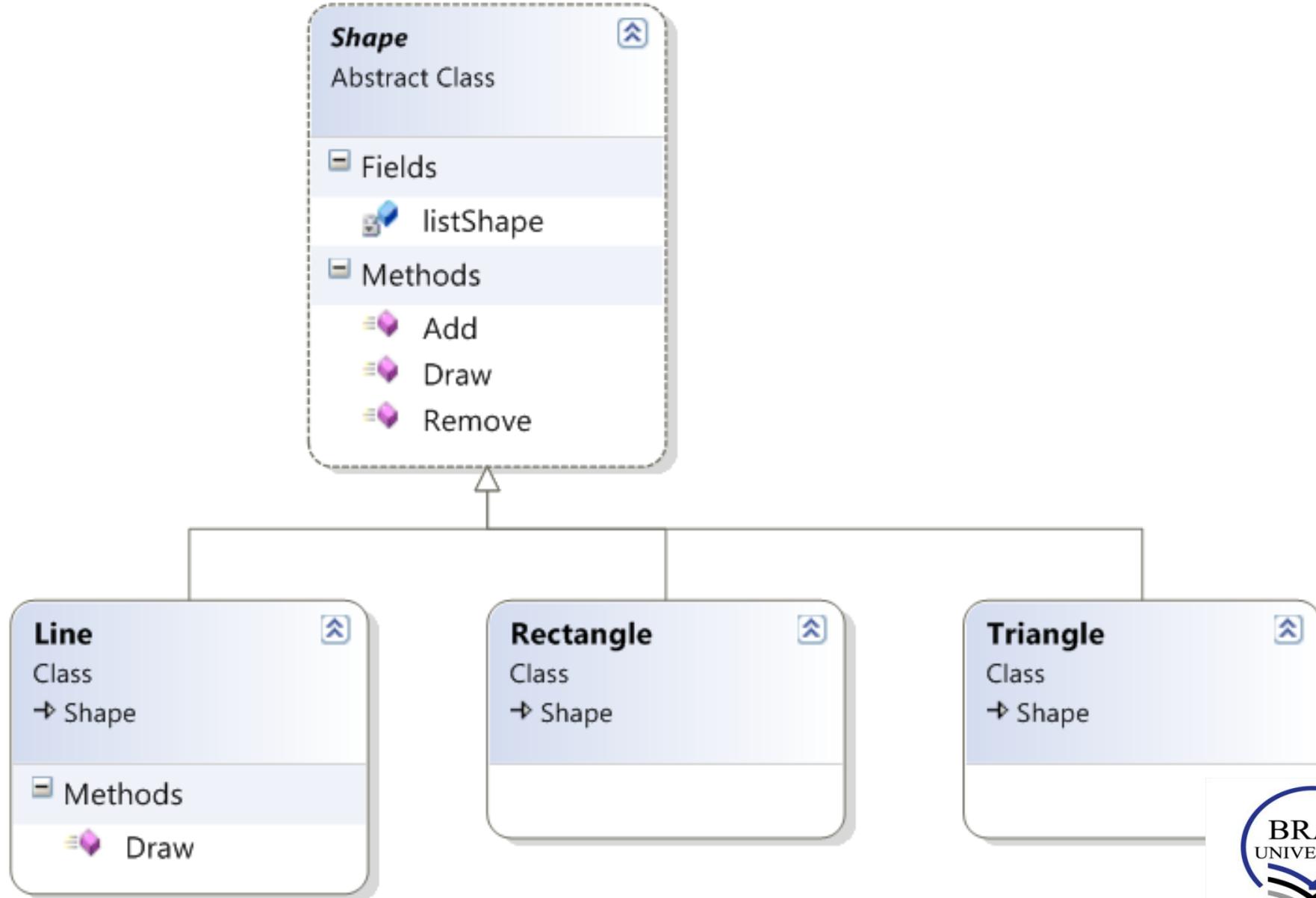


# Refused Bequest

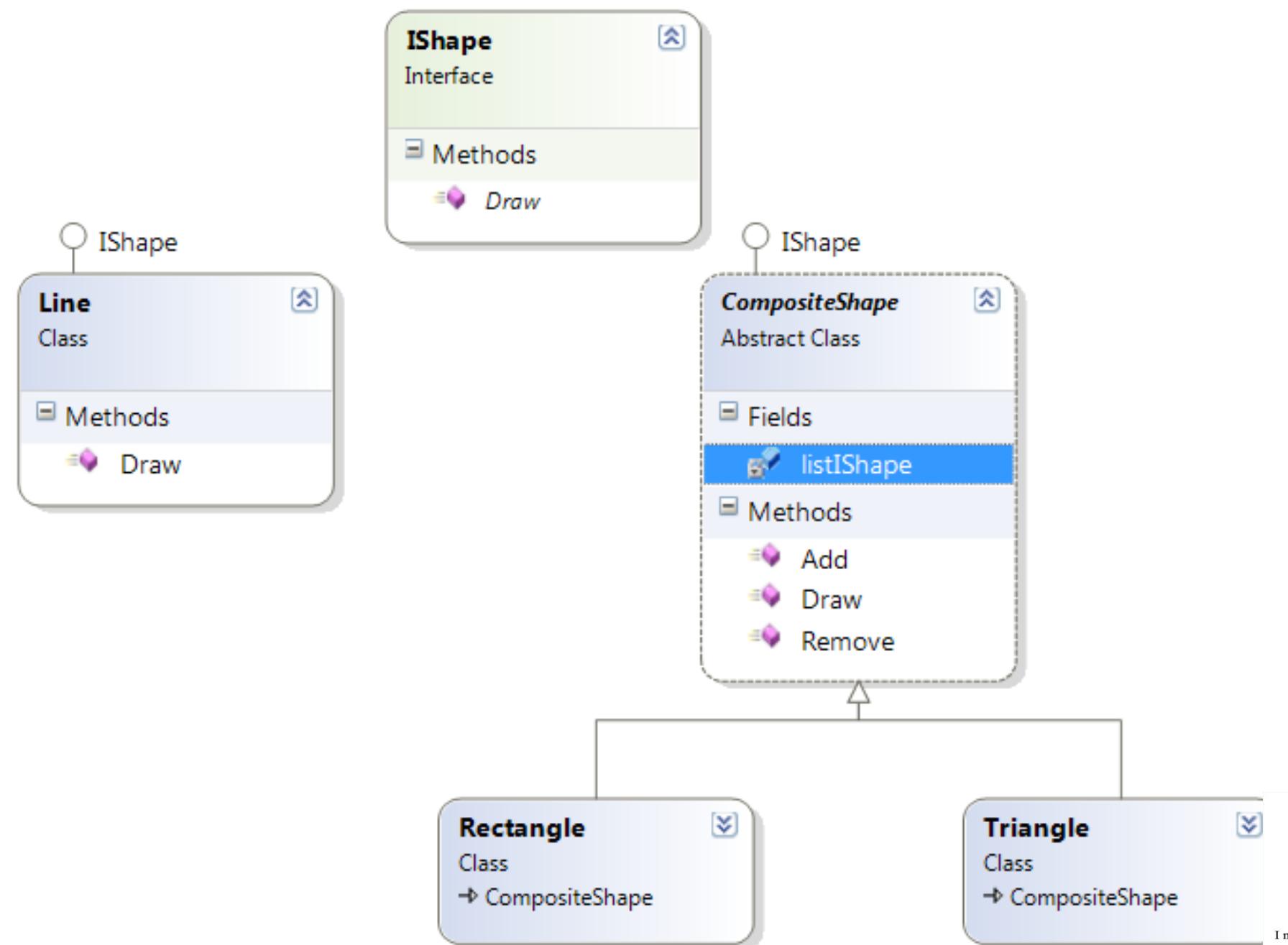
- This rather potent odor results when *subclasses inherit code that they don't want*.  
In some cases, a subclass may “refuse the bequest” by providing a *do-nothing implementation* of an inherited method.
- Remedies
  - Push Down Field
  - Push Down Method



# Example of Refused Bequest



# Refused Bequest Make Over



# References

---

- [F] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 2000
- [K] Kerievsky, Joshua. *Refactoring to Patterns*. Boston, MA: Addison-Wesley, 2005