SVR ENGINEERING COLLEGE

AYYALURUMETTA (V), NANDYAL, KURNOOL DT.
ANDHRA PRADESH – 518502

2020 – 2021

LABORATORY MANUAL

OF

SOFTWARE ENGINEERING LAB

(19A05304P)

(R-19 REGULATION)

Prepared by

Mr. K.AMARENDRANATH
Associ.
ProfessorFor

B.Tech II YEAR – IVTH SEM. (CSE)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SVR ENGINEERING COLLEGE

(AFFILIATED TO JNTUA ANANTHAPURAM- AICITE-INDIA)
AYYALURUMETTA (V), NANDYAL, KURNOOL DT.
ANDHRA PRADESH – 518502

LAB MANUAL CONTENT

SOFTWARE ENGINEERING LAB

(19A05304P)

1. Institute Vision & Mission, Department Vision & Mission
2. PO, PEO& PSO Statements.
3. List of Experiments
4. CO-PO Attainment
5. Experiment Code and Outputs

1. Institute Vision & Mission, Department Vision & Mission

Institute Vision:

To produce Competent Engineering Graduates & Managers with a strong base of Technical & Managerial Knowledge and the Complementary Skills needed to be Successful Professional Engineers & Managers.

Institute Mission:

To fulfill the vision by imparting Quality Technical & Management Education to the Aspiring Students, by creating Effective Teaching/Learning Environment and providing State – of the – Art Infrastructure and Resources.

Department Vision:

To produce Industry ready Software Engineers to meet the challenges of 21st Century.

Department Mission:

- Impart core knowledge and necessary skills in Computer Science and Engineering through innovative teaching and learning methodology.
- Inculcate critical thinking, ethics, lifelong learning and creativity needed for industry and society.
- Cultivate the students with all-round competencies, for career, higher education and self-employability.

2. PO, PEO& PSO Statements

PROGRAMME OUTCOMES (POs)

PO-1: Engineering knowledge - Apply the knowledge of mathematics, science, engineering fundamentals of Computer Science& Engineering to solve complex real-life engineering problems related to CSE.

PO-2: Problem analysis - Identify, formulate, review research literature, and analyze complex engineering problems related to CSE and reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO-3: Design/development of solutions - Design solutions for complex engineering problems related to CSE and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, cultural, societal and environmental considerations.

PO-4: Conduct investigations of complex problems - Use research-based knowledge and research methods, including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.

PO-5: Modern tool usage - Select/Create and apply appropriate techniques, resources and modern engineering and IT tools and technologies for rapidly changing computing needs, including prediction and modeling to complex engineering activities, with an understanding of the limitations.

PO-6: The engineer and society - Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the CSE professional engineering practice.

PO-7: Environment and Sustainability - Understand the impact of the CSE professional engineering solutions in societal and environmental contexts and demonstrate the knowledge of, and need for sustainable development.

PO-8: Ethics - Apply ethical principles and commit to professional ethics and responsibilities and norms of the relevant engineering practices.

PO-9: Individual and team work - Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO-10: Communication - Communicate effectively on complex engineering activities with the engineering community and with the society-at-large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, give and receive clear instructions.

PO-11: Project management and finance - Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO-12: Life-long learning - Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadcast context of technological changes.

Program Educational Objectives (PEOs):

PEO 1:Graduates will be prepared for analyzing, designing, developing and testing the software solutions and products with creativity and sustainability.

PEO 2: Graduates will be skilled in the use of modern tools for critical problem solvingand analyzing industrial and societal requirements.

PEO 3:Graduates will be prepared with managerial and leadership skills for career and starting up own firms.

Program Specific Outcomes (PSOs):

PSO 1: Develop creative solutions by adapting emerging technologies / tools for real time applications.

PSO 2: Apply the acquired knowledge to develop software solutions and innovative mobile apps for various automation applications

2.1 Subject Time Table

SVR ENGINEERING COLLEGE::NANDYAL DEPARTMENT OF CSE								
K.AMARENDRANATH						II-IVTH		
Day/ Time	9:30 AM	10:20 AM	11:30 AM	12:20 PM-	LUNCH BREAK	02:00 PM	02:50 PM	03:40 PM
	10:20 AM	11:10AM	12:20 PM	01:10 PM		02:50 PM	03:40 PM	04:30 PM
MON								
TUE								
WED								
THU								
FRI						SE LAB		
SAT								

Problem Statement: Draw the Work Breakdown Structure for the system to be automated

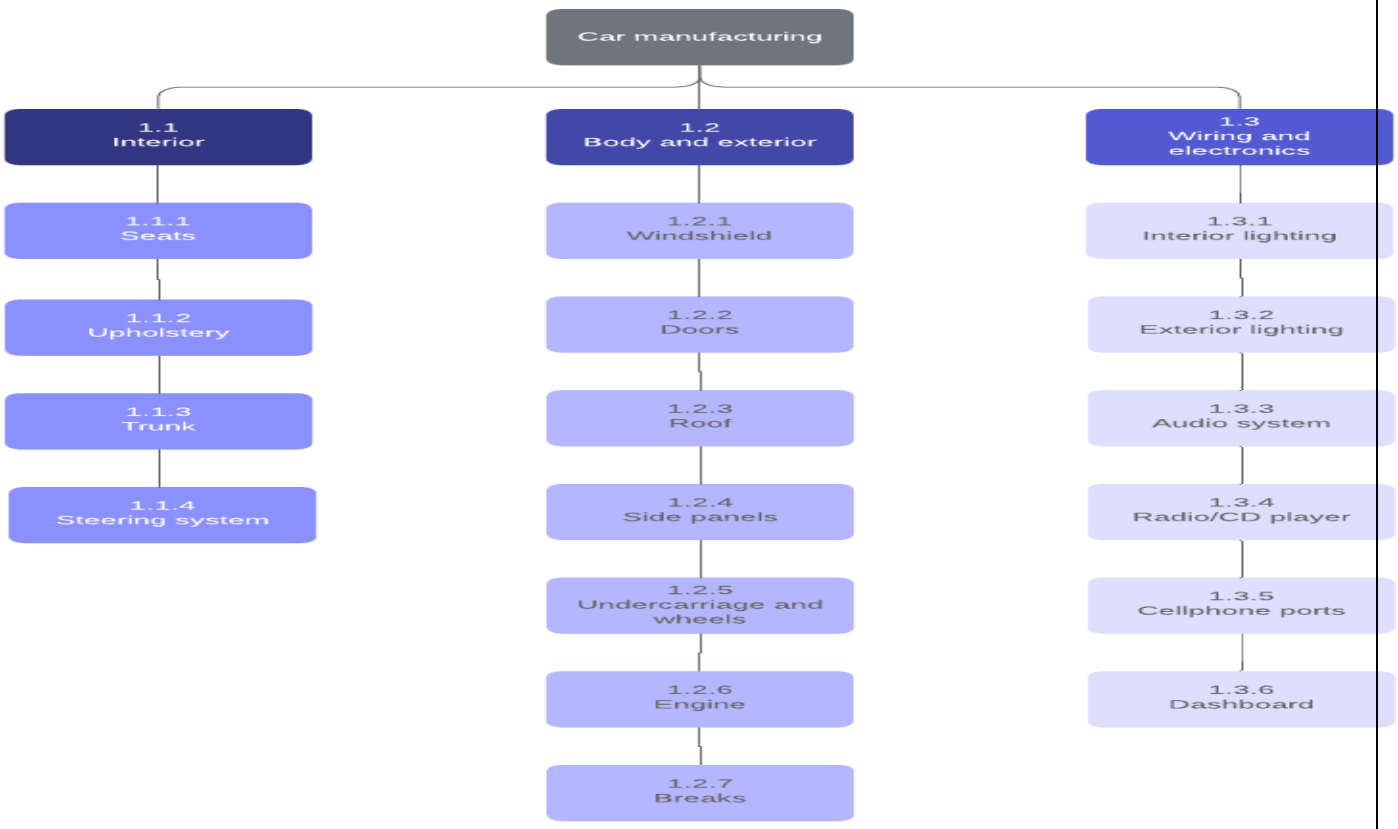
Procedure:

A) How to Create a Work Breakdown Structure and Why You Should?

1. Include 100% of the work necessary to complete the goal.
2. Don't account for any amount of work twice.
3. Focus on outcomes, not actions.
4. A work package should take no less than 8 hours and no more than 80 hours of effort.
5. Include about three levels of detail.
6. Assign each work package to a specific team or individual.

B) What is a work breakdown structure?

A work breakdown structure starts with a large project or objective and breaks it down into smaller, more manageable pieces that you can reasonably evaluate and assign to teams. Rather than focusing on individual actions that must be taken to accomplish a project, a WBS generally focuses on deliverables or concrete, measurable milestones.



C) Why use a WBS in project management?

1. Estimate the cost of a project.
2. Establish dependencies.
3. Determine a project timeline and develop a schedule.
4. Write a statement of work (or SOW, one of your other acronyms).
5. Assign responsibilities and clarify roles.
6. Track the progress of a project.
7. Identify risk.

D) How to create a work breakdown structure?

1. Record the overarching objective you are trying to accomplish. This objective could be anything from developing a new software feature to building a missile.
2. Divide the overarching project into smaller and smaller pieces, but stop before you get to the point of listing out every action that must be taken. Remember to focus on concrete deliverables rather than actions.
3. Depending on the nature of your project, start dividing by project phases, specific large deliverables, or sub-tasks.

E) Tips for making a work breakdown structure

- **The 100% rule.** The work represented by your WBS must include 100% of the work necessary to complete the overarching goal without including any extraneous or unrelated work. Also, child tasks on any level must account for all of the work necessary to complete the parent task.
- **Mutually exclusive.** Do not include a sub-task twice or account for any amount of work twice. Doing so would violate the 100% rule and will result in miscalculations as you try to determine the resources necessary to complete a project.
- **Outcomes, not actions.** Remember to focus on deliverables and outcomes rather than actions. For example, if you were building a bike, a deliverable might be “the braking system” while actions would include “calibrate the brake pads.”
- **The 8/80 rule.** There are several ways to decide when a work package is small enough without being too small. This rule is one of the most common suggestions—a work package should take no less than eight hours of effort, but no more than 80. Other rules suggest no more than ten days (which is the same as 80 hours if you work full time) or no more than a standard reporting period. In other words, if you report on your work every month, a work package should take no more than a month to complete. When in doubt, apply the “if it makes sense” rule and use your best judgment.

- **Three levels.** Generally speaking, a WBS should include about three levels of detail. Some branches of the WBS will be more subdivided than others, but if most branches have about three levels, the scope of your project and the level of detail in your WBS are about right.
- **Make assignments.** Every work package should be assigned to a specific team or individual. If you have made your WBS well, there will be no work overlap so responsibilities will be clear.

Experiment-2

Problem Statement: Schedule all the activities and sub-activities using the PERT/CPM charts

Procedure:

1. Introduction

Basically, CPM (Critical Path Method) and PERT (Programme Evaluation Review Technique) are project management techniques, which have been created out of the need of Western industrial and military establishments to plan, schedule and control complex projects.

Planning, Scheduling & Control

Planning, Scheduling (or organizing) and Control are considered to be basic Managerial functions, and CPM/PERT has been rightfully accorded due importance in the literature on Operations Research and Quantitative Analysis.

Far more than the technical benefits, it was found that PERT/CPM provided a focus around which managers could brain-storm and put their ideas together. It proved to be a great communication medium by which thinkers and planners at one level could communicate their ideas, their doubts and fears to another level. Most important, it became a useful tool for evaluating the performance of individuals and teams.

The Framework for PERT and CPM

Essentially, there are six steps which are common to both the techniques. The procedure is listed below:

1. Define the Project and all of its significant activities or tasks. The Project (made up of several tasks) should have only a single start activity and a single finish activity.
2. Develop the relationships among the activities. Decide which activities must precede and which must follow others.
3. Draw the "Network" connecting all the activities. Each Activity should have unique event numbers. Dummy arrows are used where required to avoid giving the same numbering to two activities.
4. Assign time and/or cost estimates to each activity
5. Compute the longest time path through the network. This is called the critical path.

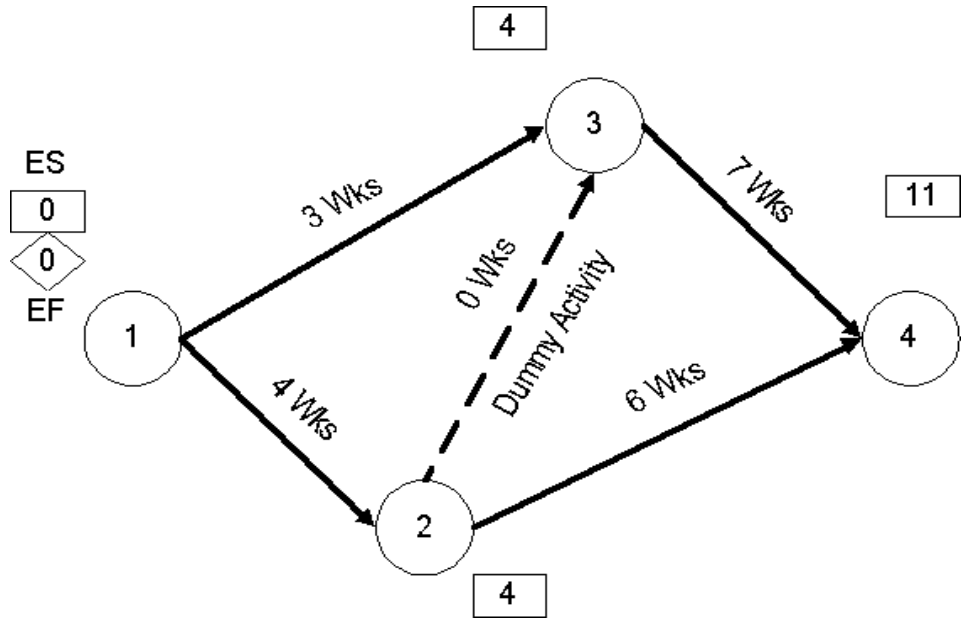
6. Use the Network to help plan, schedule, and monitor and control the project.

The Key Concept used by CPM/PERT is that a small set of activities, which make up the longest path through the activity network control the entire project. If these "critical" activities could be identified and assigned to responsible persons, management resources could be optimally used by concentrating on the few activities which determine the fate of the entire project.

Non-critical activities can be preplanned, rescheduled and resources for them can be reallocated flexibly, without affecting the whole project.

Drawing the CPM/PERT Network

Each activity (or sub-project) in a PERT/CPM Network is represented by an arrow symbol. Each activity is preceded and succeeded by an event, represented as a circle and numbered.

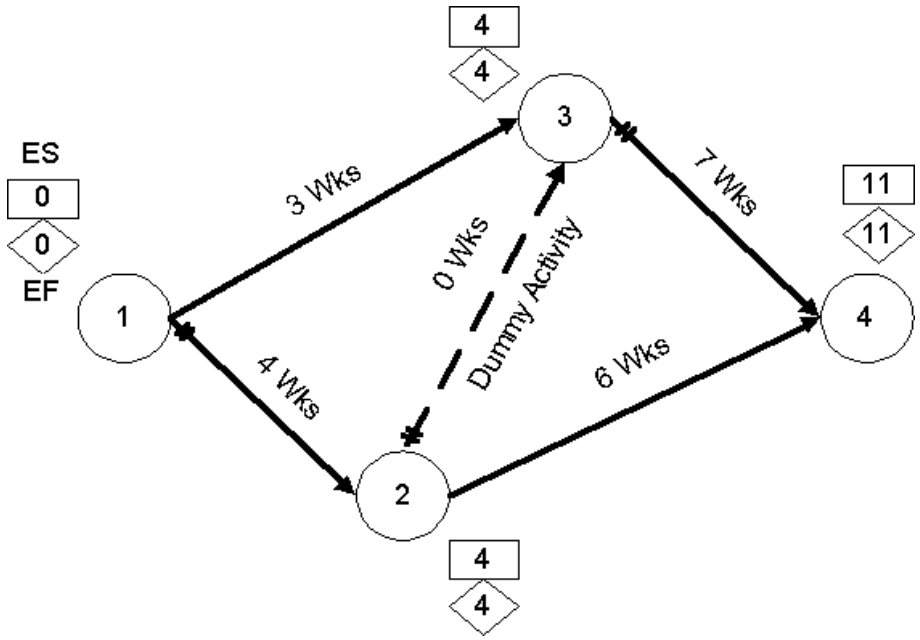


At Event 3, we have to evaluate two predecessor activities – Activity 1-3 and Activity 2-3, both of which are predecessor activities. Activity 1-3 gives us an Earliest Start of 3 weeks at Event 3. However, Activity 2-3 also has to be completed before Event 3 can begin. Along this route, the Earliest Start would be 4+0=4. The rule is to take the longer (bigger) of the two Earliest Starts. So the Earliest Start at event 3 is 4.

Similarly, at Event 4, we find we have to evaluate two predecessor activities – Activity 2-4 and Activity 3-4. Along Activity 2-4, the Earliest Start at Event 4 would be 10 wks, but along Activity 3-4, the Earliest Start at Event 4 would be 11 wks. Since 11 wks is larger than 10 wks, we select it as the Earliest Start at Event 4.We have now found the longest path through the
Network. It will take 11 weeks along activities 1-2, 2-3 and 3-4. This is the
Critical Path.

The Backward Pass – Latest Finish Time Rule

To make the Backward Pass, we begin at the sink or the final event and work backwards to the first event.



Tabulation & Analysis of Activities

We are now ready to tabulate the various events and calculate the Earliest and Latest Start and Finish times. We are also now ready to compute the SLACK or TOTAL FLOAT, which is defined as the difference between the Latest Start and Earliest Start.

Event	Duration(Weeks)	Earliest Start	Earliest Finish	Latest Start	Latest Finish	Total Float
1-2	4	0	4	0	4	0
2-3	0	4	4	4	4	0
3-4	7	4	11	4	11	0
1-3	3	0	3	1	4	1
2-4	6	4	10	5	11	1

- The Earliest Start is the value in the rectangle near the tail of each activity
- The Earliest Finish is = Earliest Start + Duration
- The Latest Finish is the value in the diamond at the head of each activity
- The Latest Start is = Latest Finish – Duration

There are two important types of Float or Slack. These are Total Float and Free Float. TOTAL FLOAT is the spare time available when all preceding activities occur at the

earliest possible times and all succeeding activities occur at the latest possible times.
Total Float = Latest Start – Earliest Start
Activities with zero Total float are on the Critical Path

Experiment-3

Problem Statement: Define use cases and represent them in use-case document for all the stakeholders of the system to be automated

Procedure:

To start with, let's understand 'What is Use Case?' and later we will discuss 'What is Use Case Testing?'

A use case is a tool for defining the required user interaction. If you are trying to create a new application or make changes to an existing application, several discussions are made. One of the critical discussion you have to make is how you will represent the requirement for the software solution.

Business experts and developers must have a mutual understanding about the requirement, as it's very difficult to attain. Any standard method for structuring the communication between them will really be a boon. It will, in turn, reduce the miscommunications and here is the place where Use case comes into the picture.

1. Who uses 'Use Case' documents?

This documentation gives a complete overview of the distinct ways in which the user interacts with a system to achieve the goal. Better documentation can help to identify the requirement for a software system in a much easier way.

This documentation can be used by Software developers, software testers as well as Stakeholders.

Uses of the Documents:

1. Developers use the documents for implementing the code and designing it.
2. Testers use them for creating the test cases.
3. Business stakeholders use the document for understanding the software requirements.

1.1 Elements in Use Cases

Given below are the various elements:

1) Brief description: A brief description explaining the case.
2) Actor: Users that are involved in Use Cases Actions.
3) Precondition: Conditions to be satisfied before the case begins.
4) Basic Flow: 'Basic Flow' or 'Main Scenario' is the normal workflow in the system. It is the flow of transactions done by the Actors on accomplishing their goals. When the actors interact with the system, as it's the normal workflow, there won't be any error and the Actors will get the expected output.

5) Alternate flow: Apart from the normal workflow, a system can also have an 'Alternate workflow'. This is the less common interaction done by a user with the system.

6) Exception flow: The flow that prevents a user from achieving the goal.

7) Post Conditions: The conditions that need to be checked after the case is completed.

1.2 Representation

A case is often represented in a plain text or a diagram. Due to the simplicity of the use case diagram, it is considered to be optional by any organization

Use Case Example:

Here I will explain the case for 'Login' to a 'School Management System'.

Use Case Name	**Login**	
Use case Description	A user login to System to access the functionality of the system.	
Actors	Parents, Students, Teacher, Admin	
Pre-Condition	System must be connected to the network.	
Post -Condition	After a successful login a notification mail is sent to the User mail id	
Main Scenarios	**Serial No**	**Steps**
Actors/Users	1	Enter username Enter Password
	2	Validate Username and Password
	3	Allow access to System
Extensions	1a	Invalid Username System shows an error message
	2b	Invalid Password System shows an error message
	3c	Invalid Password for 4times Application closed

1.3 How to Write a Use Case?

The points summarized below will help you to write these:

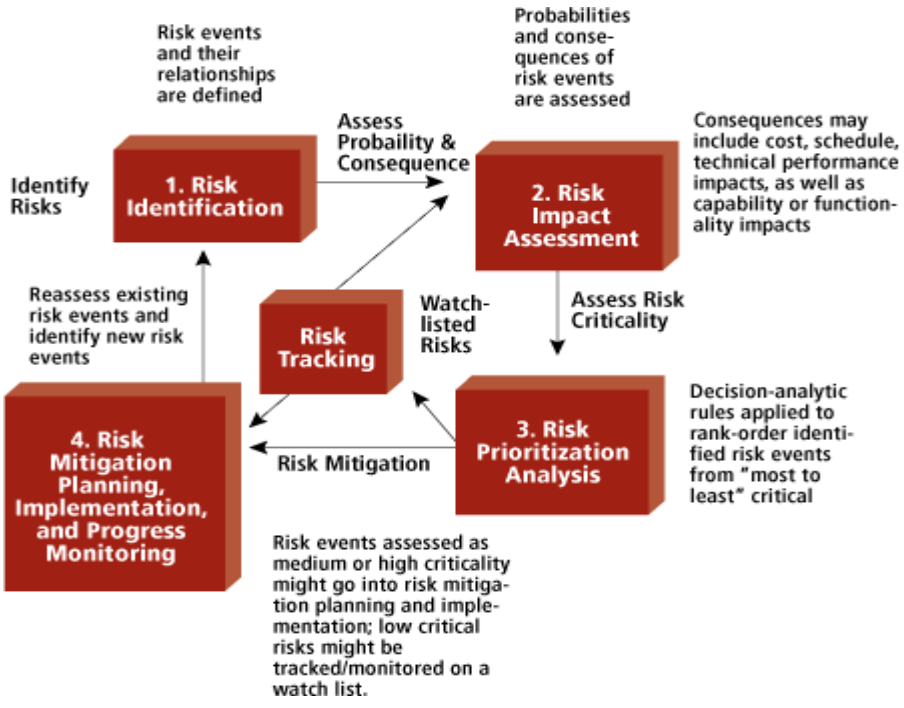
=> When we are trying to write a case, the first question that should raise is 'What's the primary use for the customer?' This question will make you write your cases from the User's perspective.
=> We must have obtained a template for these.
=> It must be productive, simple and strong. A strong Use Case can impress the audience even if they have minor mistakes.
=> We should number it.
=> We should write the Process Step in its Order.
=> Give proper name to the Scenarios, naming must be done according to the purpose.
=> This is an iterative process, which means when you write them for the first time it won't be perfect.
=> Identify the actors in the system. You may find a bunch of actors in the system.

Experiment-4

Problem Statement: Identify and analyze all the possible risks and its risk mitigation plan for the system to be automated

Procedure:

Definition: Risk mitigation planning is the process of developing options and actions to enhance opportunities and reduce threats to project objectives. Risk mitigation implementation is the process of executing risk mitigation actions. Risk mitigation progress monitoring includes tracking identified risks, identifying new risks, and evaluating risk process effectiveness throughout the project.



Example:

Evolution of Healthcare Enterprise Risk Management (ERM)

ERM encompasses eight risk domains:

1. Operational
2. Clinical & Patient Safety
3. Strategic
4. Financial
5. Human Capital
6. Legal & Regulatory
7. Technological
8. Environmental- and Infrastructure-Based Hazards.

Create a Healthcare Risk Management Plan

There are some fundamental components that belong in all healthcare risk management plans:

Education & Training:

Risk management plans need to detail employee training requirements which should include new employee orientation, ongoing and in-service training, annual review and competency validation, and event-specific training.

Patient & Family Grievances:

To promote patient satisfaction and reduce the likelihood of litigation, procedures for documenting and responding to patient and family complaints should be described in the Risk Management Plan. Response times, staff responsibilities, and prescribed actions need to be articulated and communicated.

Purpose, Goals, &Metrics:

Risk management plans should clearly define the purpose and benefits of the healthcare risk management plan. Specific goals to reduce liability claims, sentinel events, near misses, and the overall cost of the organization's risk should also be well-articulated. Additionally, reporting on quantifiable and actionable data should be detailed and mandated by the plan.

Communication Plan:

While it is critical that the healthcare risk management team promote open and spontaneous dialogue, information about how to communicate about risk and with whom should be provided in the healthcare risk management plan. Next steps and follow-up activities should be documented. It is essential as well that the plan detail reporting requirements to departments and C-Suite personnel. Furthermore, the plan should promote a safe, "no-blame" culture and should include anonymous reporting capabilities.

Contingency Plans:
Risk management plans also need to include contingency preparation for adverse system-wide failures and catastrophic situations such as malfunctioning EHR systems, security breaches, and cyber-attacks. The plan needs to include emergency preparedness for things like disease outbreaks, long-term power loss, and terror attacks or mass shootings.

Reporting Protocols:
Every healthcare organization must have a quick and easy-to-use, system for documenting, classifying, and tracking possible risks and adverse events. These systems must include protocols for mandatory reporting.

Response & Mitigation:
Plans for healthcare risk must also include collaborative systems for responding to reported risks and events including acute response, follow-up, reporting, and repeat failure prevention.

Experiment-5

Problem Statement: Diagnose any risk using Ishikawa Diagram (Can be called as Fish Bone Diagram or Cause & Effect Diagram)

Procedure:
A fishbone diagram is a visualization tool for categorizing the potential causes of a problem. This tool is used in order to identify a problem's root causes. Typically used for root cause analysis, a fishbone diagram combines the practice of brainstorming with a type of mind map template. It should be efficient as a test case technique to determine cause and effect.
A fishbone diagram is useful in product development and troubleshooting processes, typically used to focus a conversation around a problem. After the group has brainstormed all the possible causes for a problem, the facilitator helps the group to rate the potential causes according to their level of importance and diagram a hierarchy.

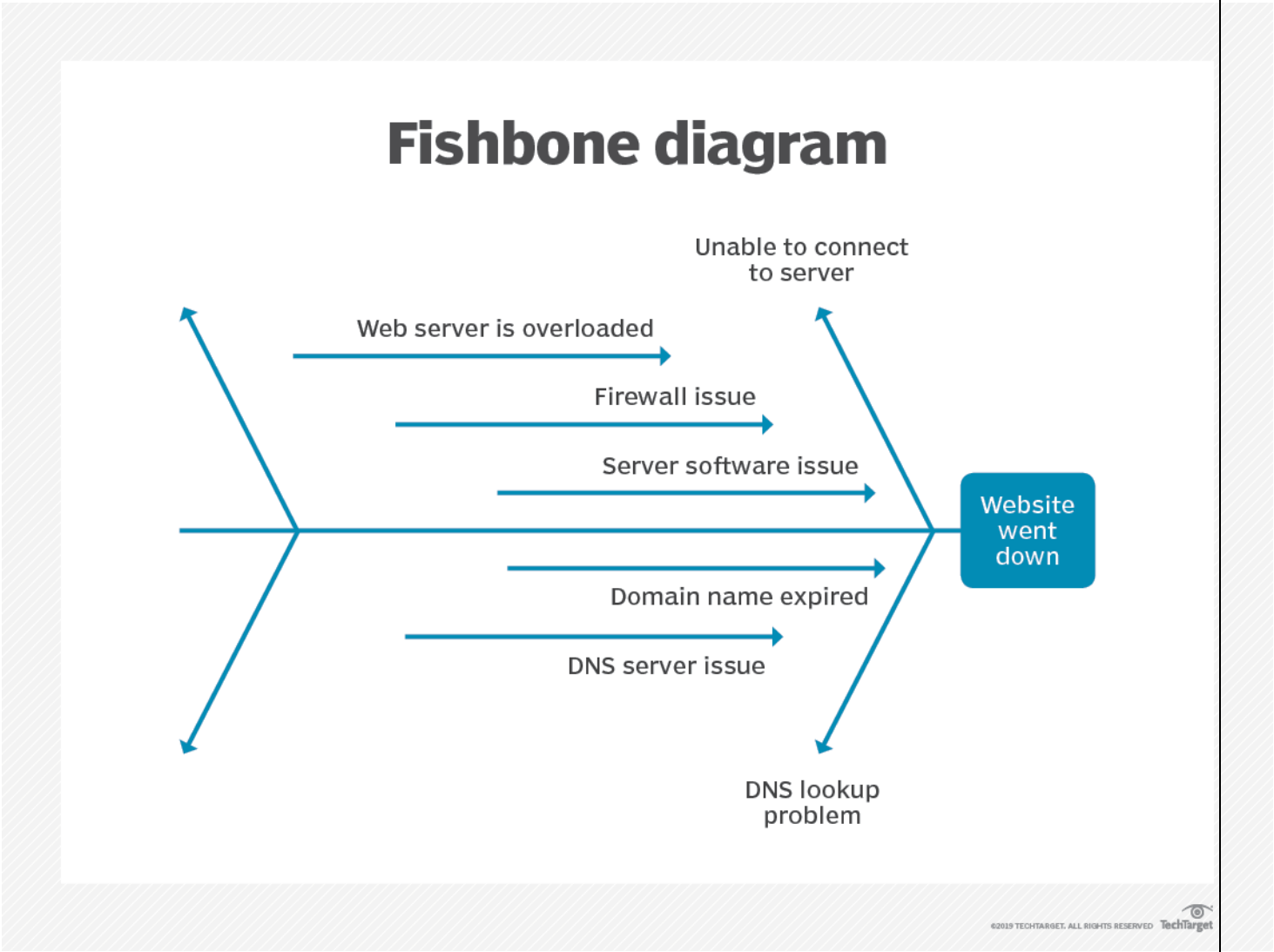
How to create a fishbone diagram

1. The head of the fish is created by listing the problem in a statement format and drawing a box around it. A horizontal arrow is then drawn across the page with an arrow pointing to the head. This acts as the backbone of the fish.
2. Then at least four overarching "causes" are identified that might contribute to the problem. Some generic categories to start with may include methods, skills, equipment, people, materials, environment or measurements. These causes are then drawn to branch off from the spine with arrows, making the first bones of the fish.

3. For each overarching cause, team members should brainstorm any supporting information that may contribute to it. This typically involves some sort of questioning methods, such as the 5 Why's or the 4P's (Policies, Procedures, People and Plant) to keep the conversation focused. These contributing factors are written down to branch off their corresponding cause.
4. This process of breaking down each cause is continued until the root causes of the problem have been identified. The team then analyzes the diagram until an outcome and next steps are agreed upon.

Example of a fishbone diagram

The following graphic is an example of a fishbone diagram with the problem "Website went down." Two of the overarching causes have been identified as "Unable to connect to server" and "DNS lookup problem," with further contributing factors branching off.



When to use a fishbone diagram

A few reasons a team might want to consider using a fishbone diagram are:

- To identify the possible causes of a problem.

- To help develop a product that addresses issues within current market offerings.
- To reveal bottlenecks or areas of weakness in a business process.
- To avoid reoccurring issues or employee burnout.
- To ensure that any corrective actions put into place will resolve the issue.

Experiment-6

Problem Statement:Define Complete Project plan for the system to be automated using Microsoft Project Tool

Procedure:

Microsoft Project Example – Let's create your first real project in a just few steps

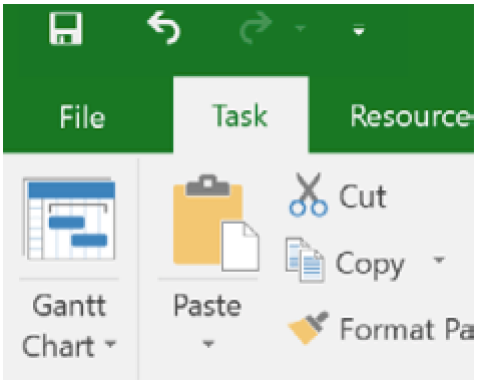
Creating a project plan in Microsoft Project isn't difficult at all.
This article will teach you how to create a simple project plan using a real project example.

Some basic configuration before you start

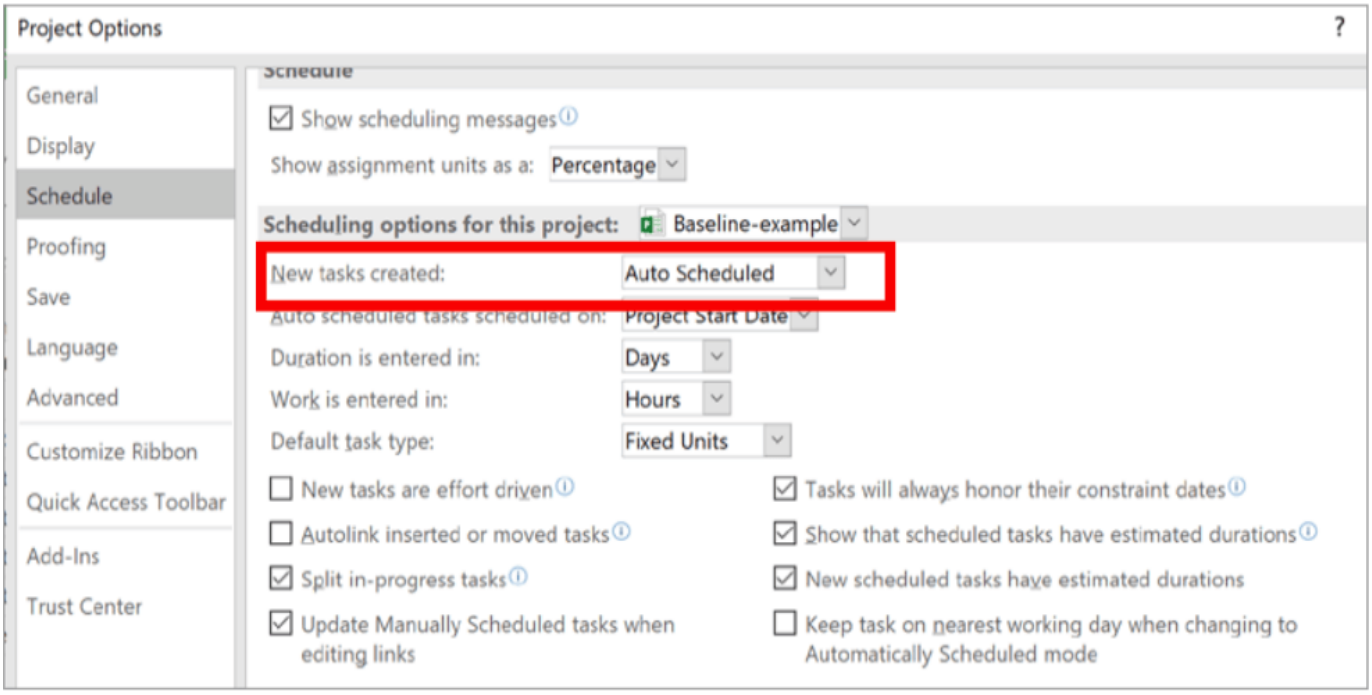
Before you create a schedule, you need to make two important changes in your settings:

Setting change 1: Make Auto Scheduling the default

o to File –> Options



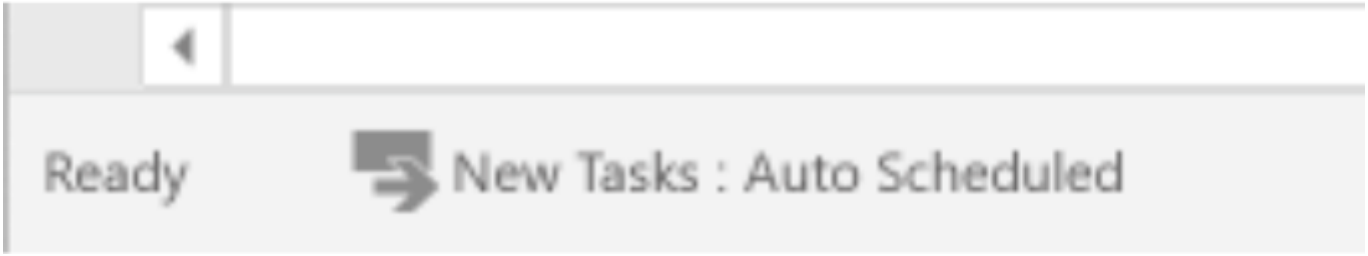
Make sure to set **Auto Schedule** for new tasks:



What does *Auto Scheduled* mean?

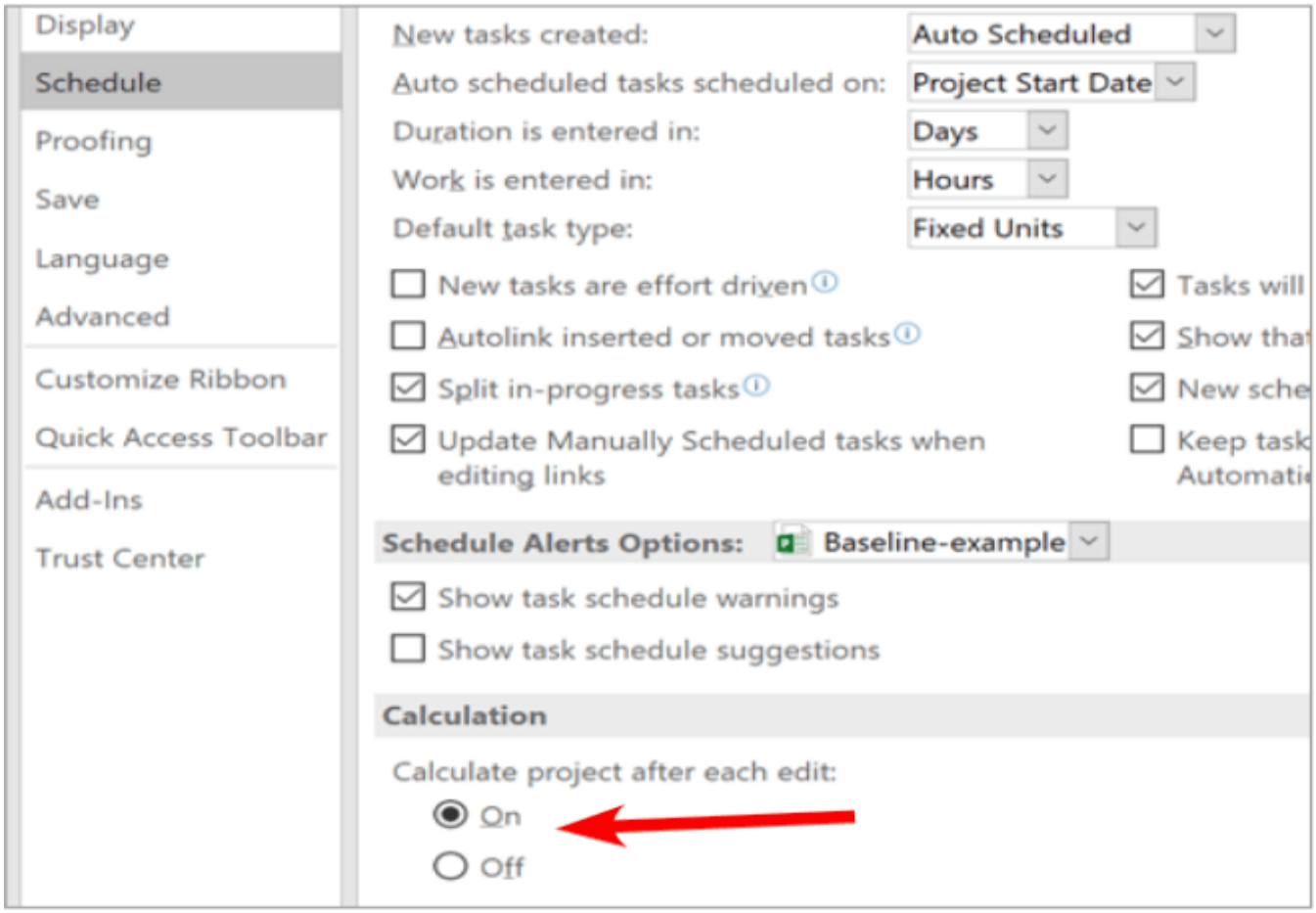
It means that new tasks will be scheduled automatically based on your project start date (or end date). More specifically, Project will determine the optimal start and end date for each task automatically, which is what we want (why would we use a computer-based scheduling tool if we would not want to automate the scheduling? Read more about manual vs. automatic scheduling in Project.)

Close the window. At the bottom left corner of the screen it should look like this:



Setting change 2: Enable immediate calculation

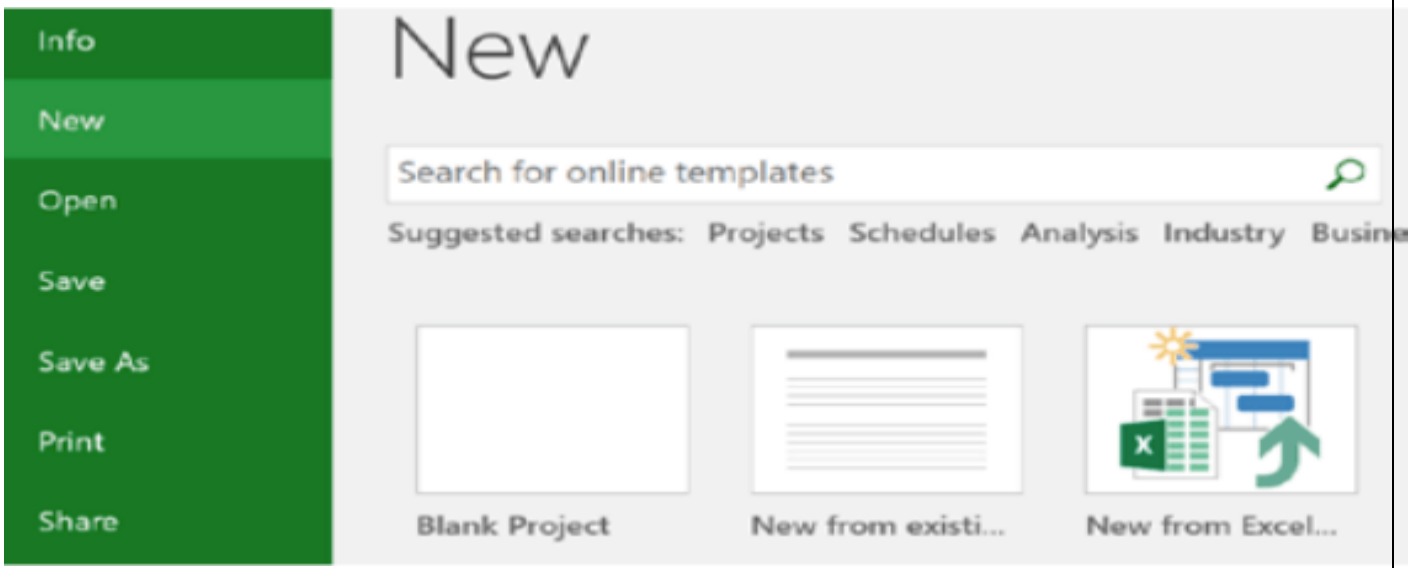
We want Microsoft Project to re-calculate the project schedule immediately after we make a change. This ensures the data you see is always up to date. Unless you run a mega-project, leave the setting enabled.



Now, let's schedule a simple project!

Our sample project: We are setting up our own business. We have picked a business idea and now we need to go from writing a business plan to a fully established business. For these steps, we are going to create a project plan.

Step 1: Create a new project

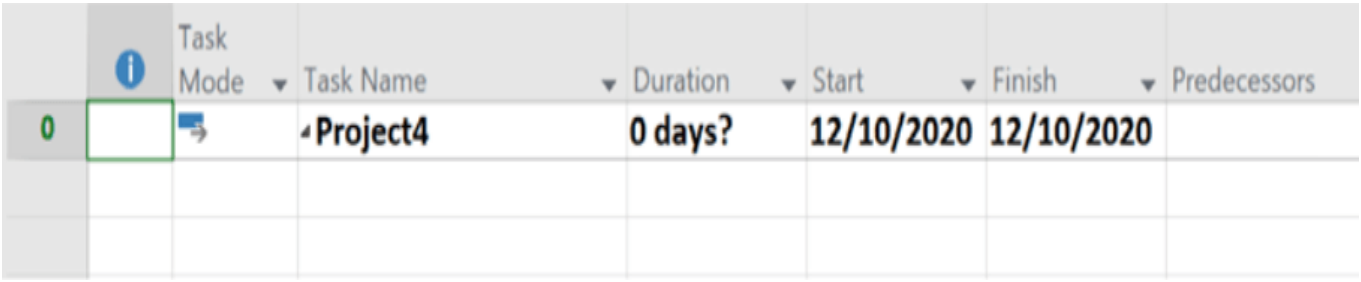


Choose *Blank Project.*

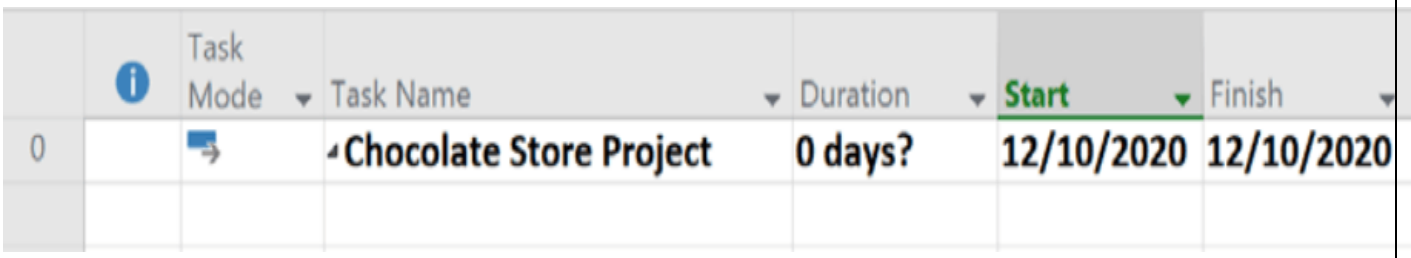
You will see a blank window.

First, let's create a **Project Summary Task**. This is like an overall "wrapper" task that contains our entire project.

Here's what you should see:



Give your project a suitable name:

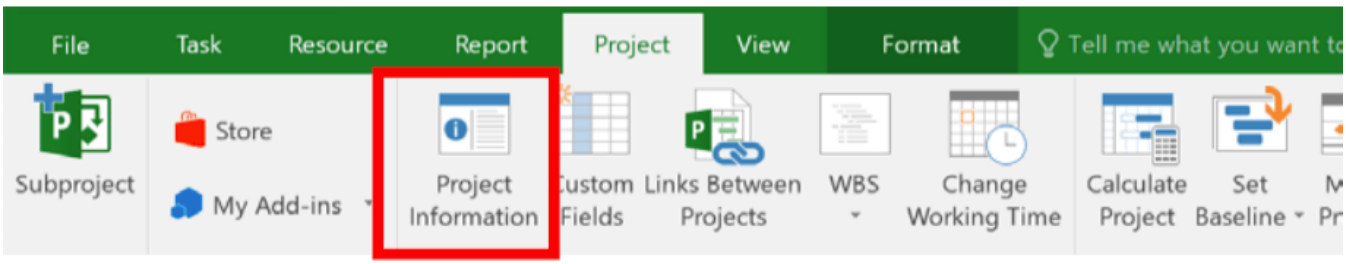


Don't worry about the duration and the dates – we'll take care of this later.

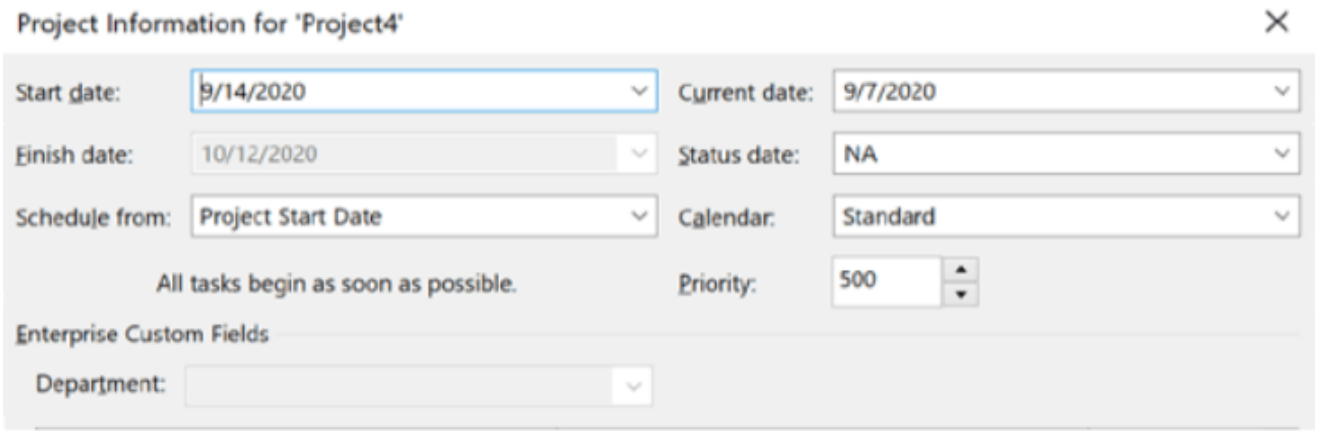
Step 2: Enter a project start date

You need to tell Project the date at which the project officially starts.

To set the project start date, open the Project tab and click *Project Information*:



Enter the project start date (in our example: 14th September 2020):



Note: You can decide whether you want Microsoft Project to schedule your project *forward* from a specific start date or *backward* from a desired end date. If you already have committed to a go live date and want to know by when you need to begin work, then chose Schedule from Project End Date to trigger the backward planning. For this example, we want to base our schedule on a start date of 14th September 2020.

Press OK after you've entered the project start date. You can see now the start date for our tasks is 14th September 2020.

tep 3: Enter the list of tasks

In this project, we need to accomplish the following tasks:

- **Create business plan**
- **Get business license**
- **Set up bank account**
- **Get funding**
- **Pick a business location**
- **Set up office equipment and furniture**
- **Hire team**
- **Run promotion**

If you look at the list, you notice that the tasks must be performed in a specific order. There are also some dependencies between the tasks.

For example, we can't get a bank account without having a business license. We also can't get funding (i.e. a bank loan) without having a business license. And of course we need the money to hire people for our store. So, everything is connected with each other.

Now let's enter those tasks into Microsoft Project.

Enter the tasks into the table next to the *Gantt view*:

	Task Mode	Task Name	Duration	Start	Finish	Prede
1						
2						
3		Create business plan	1 day?	Mon 9/7/20	Mon 9/7/20	
4		Get business license	1 day?	Mon 9/7/20	Mon 9/7/20	
5		Set up bank account	1 day?	Mon 9/7/20	Mon 9/7/20	
6		Get funding	1 day?	Mon 9/7/20	Mon 9/7/20	
7		Pick a business location	1 day?	Mon 9/7/20	Mon 9/7/20	
8		Set up office equipment a	1 day?	Mon 9/7/20	Mon 9/7/20	
9		Hire team	1 day?	Mon 9/7/20	Mon 9/7/20	
10		Run promotion	1 day?	Mon 9/7/20	Mon 9/7/20	

At this stage, Project doesn't have all the information it needs to create a schedule. It doesn't know how long each task is going to take. Therefore the *Duration* column has a question mark and the start and finish dates aren't correct yet.

Let's continue. You now need to "link" all tasks in the right order and **enter the estimated durations**.

Step 4: Enter task durations

Now, tell Project how long each task is going to take. What you enter here is the duration of the task, which is not the same as the effort. Duration is the total timespan until a task is finished. Effort (in Project, effort is called *Work*) is the amount of actual working time.

Enter the estimated duration in the duration column. **Tip:** you can either use the up/down ar- rows to change the values or enter for example "3d" to specify 3 days or "2w for 2 weeks of duration.

		Task Mode	Task Name	Duration	Start	Finish	Predeces
0			**Chocolate Store Project**	**21 days**	**9/14/2020**	**10/12/2020**	
1			Create business plan	5 days	9/14/2020	9/18/2020	
2			Get business license	1 day	9/14/2020	9/14/2020	
3			Set up bank account	1 day	9/14/2020	9/14/2020	
4			Get funding	21 days	9/14/2020	10/12/2020	
5			Pick a business location	5 days	9/14/2020	9/18/2020	
6			Set up office equipment and furniture	2 days	9/14/2020	9/15/2020	
7			Hire team	10 days	9/14/2020	9/25/2020	
8			Run promotion	4 days	9/14/2020	9/17/2020	

CHART

Experiment-7

Problem Statement:Define the Features, Vision, Business objectives, Business rules and stakeholders in the vision document

Procedure:

The Vision Document:

The Vision document is the Rational Unified Process artifact that captures all of the requirements information that we have been discussing in this chapter. As with all requirements documentation, its primary purpose is communication.

You write a Vision document to give the reader an overall understanding of the system to be developed by providing a self-contained overview of the system to be built and the motivations behind building it. To this end, it often contains extracts and summaries of other related artifacts, such as the business case and associated business models. It may also contain extracts from the system use-case model where this helps to provide a succinct and accessible overview of the system to be built.

The purpose of the Vision document is to capture the focus, stakeholder needs, goals and objectives, target markets, user environments, target platforms, and features of the product to be built. It communicates the fundamental "whys and what's" related to the project, and it is a gauge against which all future decisions should be validated.

The Vision document is the primary means of communication between the management, marketing, and project teams. It is read by all of the project stakeholders, including general managers, funding authorities, use-case modelers, and developers. It provides

- A high-level (sometimes contractual) basis for the more detailed technical requirements
- Input to the project-approval process (and therefore it is intimately related to the business case)
- A vehicle for eliciting initial customer feedback
- A means to establish the scope and priority of the product features

It is a document that gets "all parties working from the same book."

Because the Vision document is used and reviewed by a wide variety of involved personnel, the level of detail must be general enough for everyone to understand. However, enough detail must be available to provide the team with the information it needs to create a use-case model and supplementary specification.

The document contains the following sections:

- **Positioning:** This section summarizes the business case for the product and the problem or opportunity that the product is intended to address. Typically, the following areas should be addressed:

    - **The Business Opportunity:** A summary of business opportunity being met by the product
    - **The Problem Statement:** A solution-neutral summary of the problem being solved focusing on the impact of the problem and the benefits required of any successful solution
    - **Market Demographics:** A summary of the market forces that drive the product decisions.
    - **User Environment:** The user environment where the product could be applied.
- **Stakeholders and Users:** This section describes the stakeholders in, and users of, the product. The stakeholder roles and stakeholder types are defined in the project's Vision document—the actual stakeholder representatives are identified as part of the project plan just like any other resources involved in the project.
- **Key Stakeholder and User Needs:** This section describes the key needs that the stakeholders and users perceive the product should address. It does not describe their specific requests or their specific requirements, because these are captured in a separate stakeholder requests artifact. Instead, it provides the background and justification for why the requirements are needed.
- **Product Overview:** This section provides a high-level view of the capabilities, assumptions, dependencies (including interfaces to other applications and system configurations), and alternatives to the development of the product.
- **Features:** This section lists the features of the product. Features are the high-level capabilities (services or qualities) of the system that are necessary to deliver benefits to the users and satisfy the stakeholder and user needs. This is the most important, and consequently usually the longest, section of the Vision document.
- **Other Product Requirements:** This section lists any other high-level requirements that cannot readily be captured as product features. These include any constraints placed on the development of the product and any requirements the planned product places on its operating environment.

In many cases, a lot more work is put into uncovering the business opportunity and understanding the market demographics related to the proposed product than is reflected in the Vision document. This work is usually captured in-depth in business cases, business models, and market research documents. These documents are then summarized in the Vision document to ensure that they are reflected in the ongoing

evolution of the products specification.
We recommend that the Vision document be treated primarily as a report and that the stakeholder types, user types, stakeholder roles, needs, features, and other product requirements be managed using a requirements management tool. If the list of features is to be generated, it is recommended that they be presented in two sections:

- In-Scope features
- Deferred features

Do you really need to do all of this?
You are probably thinking that this all sounds like an awful lot of work, and you probably want to get started on the actual use-case modeling without producing reams and reams of additional documentation.
Well, projects are typically in one of four states when the use-case modeling activities are scheduled to commence:

- A formal Vision document has been produced.
- The information has already been captured but not consolidated into a single Vision document.
- There is a shared vision, but it has not been documented.
- There is no vision.

If your project is in one of the first two states, and the information is available to all the stakeholder representatives, then you are in a position to proceed at full speed with the construction and completion of the use-case model. If your project is in one of the last two states, then you should be very careful not to spend too much effort on the detailed use-case modeling activities. This does not mean that use-case modeling cannot be started—it simply means that any modeling you do must be undertaken in conjunction with other activities aimed at establishing a documented vision for the product. In fact, in many cases, undertaking some initial use-case modeling can act as a driver and facilitation device for the construction of the vision itself.
Our recommendation would be to always produce a Vision document for every project and to relate the information it contains to the use-case model to provide focus, context, and direction to the use-case modeling activities. For-mally relating the two sets of information also provides excellent validation of their contents and quality. If there is sufficient domain knowledge and agreement between the stakeholder representatives, then producing and reviewing the Vision document can be done very quickly. If there isn't, then there is no point in undertaking detailed use-case modeling; the resulting specifications would be ultimately worthless as they would not be a reflection of the product's true requirements.

Summary

Before embarking on any use-case modeling activities it is essential to establish a firm foundation upon which to build. The foundation has two dimen-sions, which must be

evolved in parallel with one another:

1. An understanding of the stakeholder and user community
2. The establishment of a shared vision for the product

Understanding the stakeholder community is essential as the stakeholders are the primary source of requirements. The following are the key to understanding the stakeholder community:

- **Stakeholder Types:** Definitions of all of the different types of stakeholder affected by the project and the product it produces.
- **User Types:** Definitions of characteristics and capabilities of the users of the system. The users are the people and things that will take on the roles defined by the actors in the use-case model.

For the use-case modeling activities to be successful, the stakeholders and users will need to be actively involved in them. The stakeholders and users directly involved in the project are known as stakeholder representatives. To ensure that the stakeholder representatives understand their commitment to the project, it is worthwhile to clearly define the "stakeholder roles" that they will be adopting. The stakeholder roles serve as a contract between the stakeholder representatives and the project, reflecting the responsibilities and expectations of both sides.

To establish a shared vision for the project, the following are essential:

- **The Problem Statement:** A solution-neutral summary of the problem being solved, focusing on the impact of the problem and the benefits required of any successful solution.
- **Stakeholder Needs:** The true "business requirements" of the stakeholders presented in a solution-neutral manner. These are the aspects of the problem that affect the individual stakeholders.
- **Features, Constraints, and Other High-Level Product Requirements:** A high-level definition of the system to be developed. These complement and provide a context for the use-case model and enable effective scope management.
- **Product Overview:** A summary of the other aspects of the product not directly captured by the high-level requirements.

The Vision document can be used to capture all of this information in a form that is accessible to all the stakeholders of the project.

The vision does not have to be complete before use-case modeling activities start; in fact, undertaking some initial use-case modeling can act as a driver and facilitation device for the construction of the vision itself, but if the vision is not established alongside the use-case model, then there is a strong possibility that it will not be a true reflection of the real requirements.

Experiment-8

Problem Statement:Define the functional and non-functional requirements of the system to be automated by using Use cases and document in SRS document

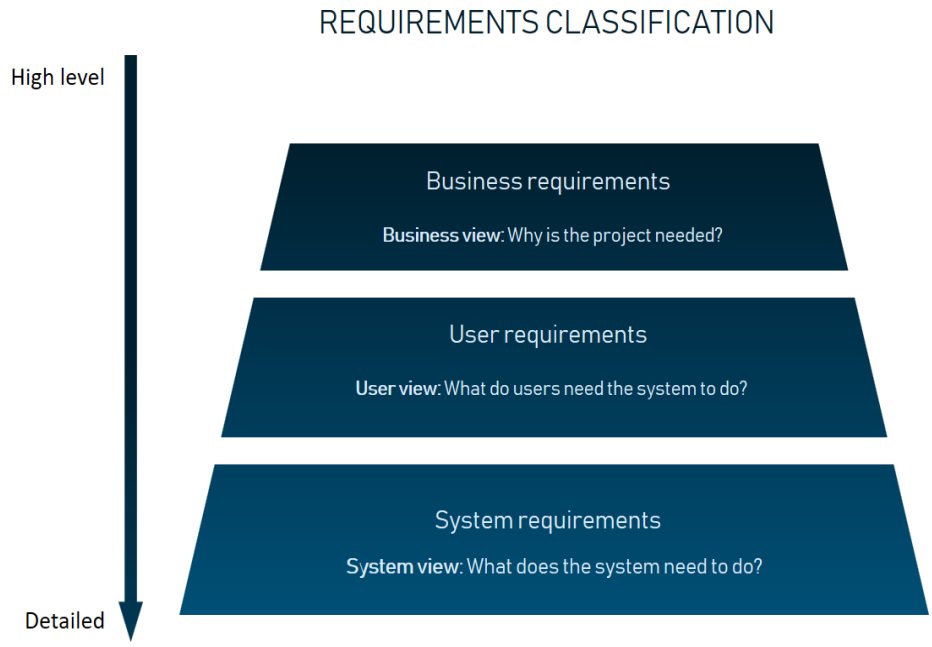
Procedure:

Clearly defined requirements are essential signs on the road that leads to a successful project. They establish a formal agreement between a client and a provider that they are both working to reach the same goal. High-quality, detailed requirements also help mitigate financial risks and keep the project on a schedule. According to the Business Analysis Body of Knowledge definition, requirements are a usable representation of a need.

Creating requirements is a complex task as it includes a set of processes such as elicitation, analysis, specification, validation, and management. In this article, we'll discuss the main types of requirements for software products and provide a number of recommendations for their use.

Classification of requirements

Prior to discussing how requirements are created, let's differentiate their types.



Business requirements. These include high-level statements of goals, objectives, and needs.

Stakeholder requirements. The needs of discrete stakeholder groups are also specified to define what they expect from a particular solution.

Solution requirements. Solution requirements describe the characteristics that a product must have to meet the needs of the stakeholders and the business itself.

- **Nonfunctional** requirements describe the general characteristics of a system. They are also known as *quality attributes*.
- **Functional** requirements describe how a product must behave, what its features and functions.

Transition requirements. An additional group of requirements defines what is needed from an organization to successfully move from its current state to its desired state with the new product.

Let's explore functional and nonfunctional requirements in greater detail.

Functional requirements and their specifications

Functional requirements are product features or functions that developers must implement to enable users to accomplish their tasks. So, it's important to make them clear both for the development team and the stakeholders. Generally, functional requirements describe system behavior under specific conditions. For instance:

A search feature allows a user to hunt among various invoices if they want to credit an issued invoice.

Here's another simple example: *As a guest, I want a sofa that I can sleep on overnight*.

Requirements are usually written in text, especially for Agile-driven projects. However, they may also be visuals. Here are the most common formats and documents:

- Software requirements specification document
- Use cases
- User stories
- Work Breakdown Structure (WBS) (functional decomposition)
- Prototypes
- Models and diagrams

Software requirements specification document

Functional and nonfunctional requirements can be formalized in the requirements specification (SRS) document. (To learn more about software documentation, read our article on that topic.) The SRS contains descriptions of functions and capabilities that the product must provide. The document also defines constraints and assumptions. The SRS can be a single document communicating functional requirements or it may accompany other software documentation like user stories and use cases.

We don't recommend composing SRS for the entire solution before the development kick-off, but you should document the requirements for every single feature before actually building it. Once you receive the initial user feedback, you can update the document.

SRS must include the following sections:

Purpose.Definitions, system overview, and background.

Overall description. Assumptions, constraints, business rules, and product vision.
Specific requirements. System attributes, functional requirements, database requirements.
It's essential to make the SRS readable for all stakeholders. You also should use templates with visual emphasis to structure the information and aid in understanding it. If you have requirements stored in some other document formats, link to them to allow readers to find the needed information.
Example: If you'd like to see an actual document, download this SRS example created at Michigan State University, which includes all points mentioned above in addition to presenting use cases to illustrate parts of the product.
Use cases
Use cases describe the interaction between the system and external users that leads to achieving particular goals.
Each use case includes three main elements:
Actors. These are the users outside the system that interact with the system.
System. The system is described by functional requirements that define an intended behavior of the product.
Goals. The purposes of the interaction between the users and the system are outlined as goals.
There are two formats to represent use cases:

- Use case specification structured in textual format
- Use case diagram

A **use case specification** represents the sequence of events along with other information that relates to this use case. A typical use case specification template includes the following information:

- Description
- Pre- and Post- interaction condition
- Basic interaction path
- Alternative path
- Exception path

Example:

Overview	
Title	[Title of the basic flow use case]
Description	[Short description of the basic flow]
Actors and Interfaces	[Identifies the Actors and Interfaces to components and services that participate in the use case]
Initial Status and Preconditions	[A pre-condition (of a use case) is the state of the system that must be present prior to a use case being performed]
Basic Flow	
STEP 1: … STEP 2: …	
Post Condition	
[A post-condition (of a use case) is a list of possible states the system can be in immediately after a use case has finished]	
Alternative Flow(s)	
[Alternative flows are described here if needed]	

Use case specification template

A **use case diagram** doesn't contain a lot of details. It shows a high-level overview of the relationships between actors, different use cases, and the system.

The use case diagram includes the following main elements:

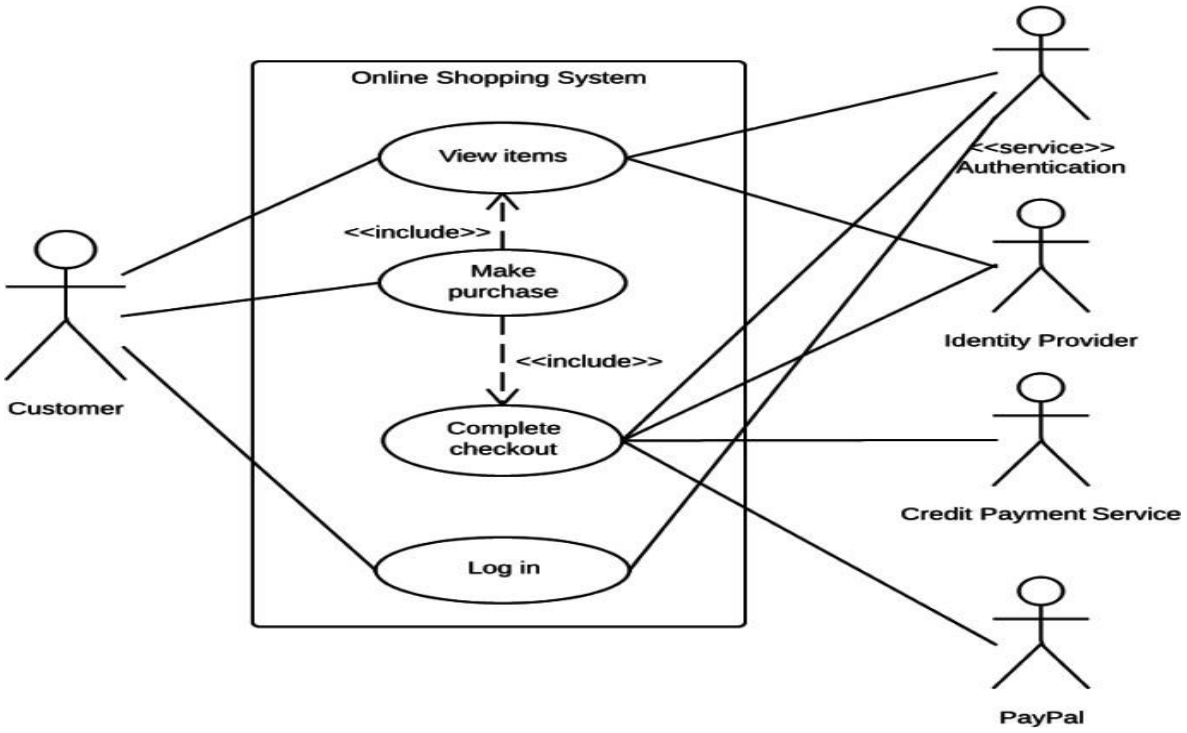
Use cases. Usually drawn with ovals, use cases represent different use scenarios that actors might have with the system (*log in, make a purchase, view items,* etc.)

System boundaries. Boundaries are outlined by the box that groups various use cases in a system.

Actors. These are the figures that depict external users (people or systems) that interact with the system.

Associations. Associations are drawn with lines showing different types of relationships between actors and use cases.

Example:



Use case diagram example

User stories

A user story is a documented description of a software feature seen from the end-user perspective. The user story describes what exactly the user wants the system to do. In Agile projects, user stories are organized in a *backlog*, which is an ordered list of product functions. Currently, user stories are considered to be the best format for backlog items.

A typical user story is written like this:

As a <type of user>, I want <some goal> so that <some reason>.

***Example**:*

As an admin, I want to add descriptions to products so that users can later view these descriptions and compare the products.

User stories must be accompanied by **acceptance criteria**. These are the conditions that the product must satisfy to be accepted by a user, stakeholders, or a product owner. Each user story must have at least one acceptance criterion. Effective acceptance criteria must be testable, concise, and completely understood by all team members and stakeholders. They can be written as checklists, plain text, or by using Given/When/Then format.

Example:

Here's an example of the acceptance criteria checklist for a user story describing a search feature:

- A search field is available on the top-bar.
- A search is started when the user clicks *Submit*.
- The default placeholder is a grey text *Type the name*.

- The placeholder disappears when the user starts typing.
- The search language is English.
- The user can type no more than 200 symbols.
- It doesn't support special symbols. If the user has typed a special symbol in the search input, it displays the warning massage: *Search input cannot contain special symbols*.

Finally, all user stories must fit the **INVEST quality model**:

- **I** – Independent
- **N** – Negotiable
- **V** – Valuable
- **E** – Estimable
- **S** – Small
- **T** – Testable

Independent. This means that you can schedule and implement each user story separately. This is very helpful if you implement continuous integration processes.
Negotiable. This means that all parties agree to prioritize negotiations over specification. This also means that details will be created constantly during development.
Valuable. A story must be valuable to the customer. You should ask yourself from the customer's perspective "why" you need to implement a given feature.
Estimable. A quality user story can be estimated. This will help a team schedule and prioritize the implementation. The bigger the story is, the harder it is to estimate it.
Small. Good user stories tend to be small enough to plan for short production releases. Small stories allow for more specific estimates.
Testable. If a story can be tested, it's clear enough and good enough. Tested stories mean that requirements are done and ready for use.

-

Experiment-9

Problem Statement:Define the following traceability matrices:

1. Use case Vs. Features

2. Functional requirements Vs.Usecases

Procedure:

The concept of Traceability Matrix is to be able to trace from top level requirements to implementation,andfrom top level requirements to test.

A traceability matrix is a table that traces a requirement to the tests that are needed to verify that the requirement is fulfilled. A good traceability matrix will provide backward and forward traceability, i.e. a requirement can be traced to a test and a test to a requirements. The matrix links higher level requirements, design specifications, test requirements, and code files. It acts as a map, providing the links necessary for determining where information is located. This is also known as Requirements Traceability Matrix or RTM.

This is mostly used for QA so that they can ensure that the customer gets what they requested. The Traceability matrix also helps the developer find out why some code was implemented the way it was, by being able to go from code to Requirements. If a test fails, it is possible to use the traceability matrix to see what requirements and code the test case relates to.

The goal of a matrix of this type is -

1. To make sure that the approved requirements are addressed/covered in all phases of development: From SRS to Development to Testing to Delivery.
2. That each document should be traceable: Written test cases should be traceable to its requirement specification. If there is new version then updated test cases should be traceable with that.

Traceability Matrix is used in entire software development life cycle phases:

1. Risk Analysis phase
2. Requirements Analysis and Specification phase
3. Design Analysis and Specification phase
4. Source Code Analysis, Unit Testing & Integration Testing phase
5. Validation – System Testing, Functional Testing phase

In this topic we will discuss:

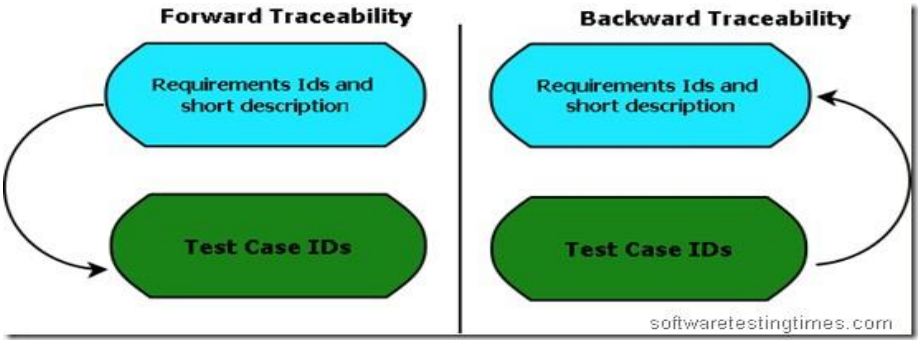
- What is Traceability Matrix from Software Testing perspective? (Point 5)
- Types of Traceability Matrix
- Disadvantages of not using Traceability Matrix
- Benefits of using Traceability Matrix in testing

- Step by step process of creating an effective Traceability Matrix from requirements. Sample formats of Traceability Matrix basic version to advanced version.

In Simple words - A requirements traceability matrix is a document that traces and maps user requirements [requirement Ids from requirement specification document] with the test case ids. Purpose is to make sure that all the requirements are covered in test cases so that while testing no functionality can be missed.
This document is prepared to make the clients satisfy that the coverage done is complete as end to end, this document consists of Requirement/Base line doc Ref No., Test case/Condition, and Defects/Bug id. Using this document the person can track the Requirement based on the Defect id.

Types of Traceability Matrix:

- Forward Traceability – Mapping of Requirements to Test cases
- Backward Traceability – Mapping of Test Cases to Requirements
- Bi-Directional Traceability - A Good Traceability matrix is the References from test cases to basis documentation and vice versa.



Why Bi-Directional Traceability is required?

Bi-Directional Traceability contains both Forward & Backward Traceability. Through Backward Traceability Matrix, we can see that test cases are mapped with which requirements.
This will help us in identifying if there are test cases that do not trace to any coverage item— in which case the test case is not required and should be removed (or maybe a specification like a requirement or two should be added!). This “backward” Traceability is also very helpful if you want to identify that a particular test case is covering how many requirements?
Through Forward Traceability – we can check that requirements are covered in which test cases? Whether is the requirements are coved in the test cases or not?
Forward Traceability Matrix ensures – We are building the Right Product.
Backward Traceability Matrix ensures – We the Building the Product Right.

Traceability matrix is the answer of the following questions of any Software Project:

- How is it feasible to ensure, for each phase of the SDLC, that I have correctly accounted for all the customer’s needs?

- How can I certify that the final software product meets the customer's needs? Now we can only make sure requirements are captured in the test cases by traceability matrix.

Disadvantages of not using Traceability Matrix [some possible (seen) impact]:

No traceability or Incomplete Traceability Results into:

1. Poor or unknown test coverage, more defects found in production
2. It will lead to miss some bugs in earlier test cycles which may arise in later test cycles. Then a lot of discussions arguments with other teams and managers before release.
3. Difficult project planning and tracking, misunderstandings between different teams over project dependencies, delays, etc

Benefits of using Traceability Matrix

- Make obvious to the client that the software is being developed as per the requirements.
- To make sure that all requirements included in the test cases
- To make sure that developers are not creating features that no one has requested
- Easy to identify the missing functionalities.
- If there is a change request for a requirement, then we can easily find out which test cases need to update.
- The completed system may have "Extra" functionality that may have not been specified in the design specification, resulting in wastage of manpower, time and effort.

Steps to create Traceability Matrix:

1. Make use of excel to create Traceability Matrix:
2. Define following columns:

Base Specification/Requirement ID (If any)
Requirement ID
Requirement description
TC 001
TC 002
TC 003.. So on.

3. Identify all the testable requirements in granular level from requirement document.

Typical requirements you need to capture are as follows:
Used cases (all the flows are captured)
Error Messages
Business rules
Functional rules
SRS
FRS

So on…
4. Identity all the test scenarios and test flows.
5. Map Requirement IDs to the test cases. Assume (as per below table), Test case “TC 001” is your one flow/scenario. Now in this scenario, Requirements SR-1.1 and SR-1.2 are covered. So, mark “x” for these requirements.
Now from below table you can conclude –
Requirement SR-1.1 is covered in TC 001
Requirement SR-1.2 is covered in TC 001
Requirement SR-1.5 is covered in TC 001, TC 003 [Now it is easy to identify, which test cases need to be updated if there is any change request].
TC 001 Covers SR-1.1, SR, 1.2 [we can easily identify that test cases covers which requirements].
TC 002 covers SR-1.3... So on...

Requirement ID	Requirement description	TC 001	TC 002	TC 003
SR-1.1	User should be able to do this	x		
SR-1.2	User should be able to do that	x		
SR-1.3	On clicking this, following message should appear		x	
SR-1.4			x	
SR-1.5		x		x
SR-1.6				x
SR-1.7			x	

Use Case
A use case describes the interactions between one of more Actors and the system in order to provide an observable result of value for the initiating actor.
The functionality of a system is defined by different use cases, each of which represents a specific goal (to obtain the observable result of value) for a particular actor.

In an automated teller machine shown in Figure 1, the Bank Customer can withdraw cash from an account, transfer funds between accounts, or deposit funds to an account. These correspond to specific goals that the actor has in using the system.

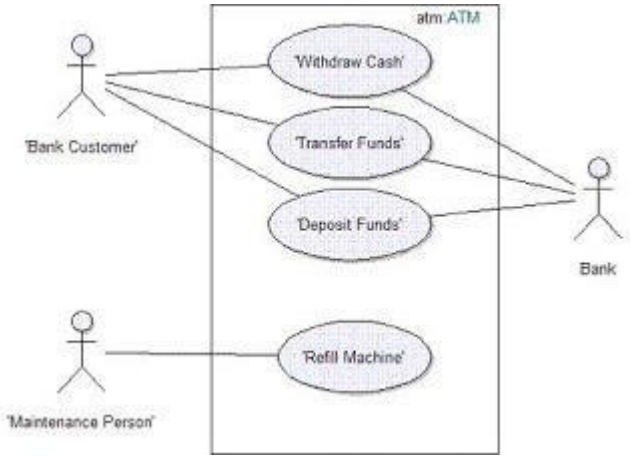


Figure 1: ATM Use-Case Example

Each use case is associated with a goal of one of the actors. The collection of use cases constitutes all the possible ways of using the system. You should be able to determine the goal of a use case simply by observing its name.

A use case describes the interactions between the actor(s) and the system in the form of a dialog between the actor(s) and the system, structured as follows:

1. The actor <<does something>>
2. The system <<does something in response>>
3. The actor <<does something else>>
4. The system ...

Each dialog of this form is called a "Flow of Events".

Because there are many flows of events possible for achieving the goal (for example, the flow may differ depending upon specific input from the actor), and there are situations in which the goal cannot be achieved (for example, a required network connection is currently unavailable), each use case will contain several flows, including one "Basic Flow of Events" and several "Alternative Flows".

The Basic Flow of Events specifies the interactions between the actor(s) and the system for the ideal case, where everything goes as planned, and the actor's goal (the observable result of value) is met. The basic flow represents the main capability provided by the system for this use case.

As the name implies, Alternative Flows specify alternative interactions associated with the same goal.

Closely related to use cases is the concept of a scenario. A scenario is a specific flow of events, for a specific set of inputs to the system, states of the system, and states of the system's environment. Scenarios are closely related to test cases.

Properties of Use Cases

Name

Each use case should have a name that indicates what is achieved by its interaction

with the actors. The name may have to be several words to be understood. Note: No two use cases can have the same name.

Brief Description

The brief description of the use case should reflect its purpose.

Flow of Events

Flow of Events - Contents

The flow of events should describe the use case's flow of events clearly enough for an outsider to easily understand. Remember, the flow of events should represent *what* the system does, not *how* the system is design to perform the required behavior.

Follow these guidelines for the contents of the flow of events:

· Describe how the use case starts and ends.

· Describe what data is exchanged between the actor and the use case.

· Do not describe the details of the user interface, unless it is necessary to understand the behavior of the system. Specifying user interface details too early will limit design options.

· Describe the flow of events, not only the functionality. To enforce this, start every action with "When the actor ... ".

· Describe only the events that belong to the use case, and not what happens in other use cases or outside of the system.

· Avoid vague terminology.

· Detail the flow of events. Specify what happens when, for each action.

Remember this text will be used to identify test cases.

If you have used certain terms in other use cases, be sure to use the exact same terms in this use case, and that the meaning of the terms is consistent. To manage common terms, put them in a glossary.

Flow of Events - Structure

The two main parts of the flow of events are **basic flow of events** and **alternative flows of events**. The basic flow of events should cover what "normally" happens when the use case is performed. The alternative flows of events cover behavior of optional or exceptional character in relation to the normal behavior, and also variations of the normal behavior. You can think of the alternative flows of events as detours from the basic flow of events, some of which will return to the basic flow of events and some of which will end the execution of the use case.

The straight arrow in Figure 2 represents the basic flow of events, and the curves represent alternative paths in relation to the normal. Some alternative paths return to the basic flow of events, whereas others end the use case.

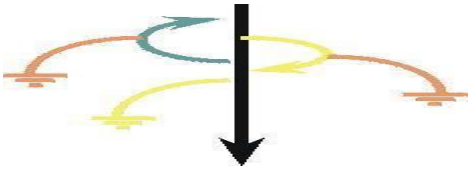


Figure 2: Typical structure of a use case flow of events

Both the basic and alternative flows should be further structured into steps or sub-flows. In doing this, your main goal should be readability of the text. A guideline is that a sub-flow should be a segment of behavior within the use case that has a clear purpose, and is "atomic" in the sense that you do either all or none of the actions described.

Special Requirements

In the Special Requirements of a use case, you describe all the requirements associated with the use case that are not covered by the flow of events. These are non-functional requirements that will influence the design. See also the discussion on non-functional requirements in Concept: Requirements.

Preconditions and Post-conditions

A **precondition** is the state of the system and its environment that is required before the use case can be started. Post-Conditions are the states the system can be in after the use case has ended. It can be helpful to use the concepts of **precondition** and **post-condition** to clarify how the flow of events starts and ends. However, only use them only if the audience for the description of the use case agrees that it is helpful. Figure 3 shows an example.

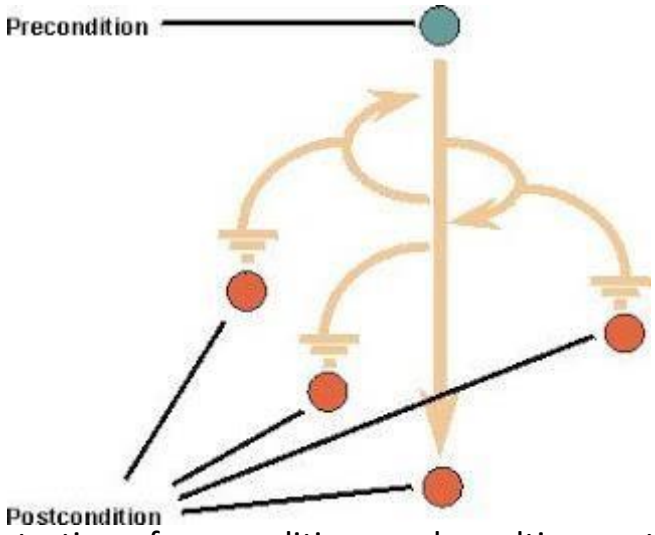


Figure 3: Illustration of preconditions and resulting post-conditions

Examples

A precondition for the use case Cash Withdrawal in the ATM machine: The customer has a personally issued card that fits in the card reader, has been issued a PIN number, and is registered with the banking system.

A post-condition for the use case Cash Withdrawal in the ATM machine: At the end of the use case, all account and transaction logs are balanced, communication with the banking system is reinitialized and the card is returned to the customer.

Experiment-10

Problem Statement:Estimate the effort using the following methods for the system to be automated:

1. Function point metric

2. Use case point metric

Procedure:

For the success of any project test estimation and proper execution is equally important as the development cycle. Sticking to the estimation is very important to build a good reputation with the client.

Experience plays a major role in estimating "Software Testing Efforts". Working on varied projects helps to prepare an accurate estimation of the testing cycle. Obviously one cannot just blindly put some number of days for any testing task. **Test estimation should be realistic and accurate.**

Brief Description of the Test Estimation Process

"Estimation is the process of finding an estimate, or approximation, which is a value that is usable for some purpose even if input data may be incomplete, uncertain, or unstable."

We all come across different tasks and duties and deadlines throughout our lives as professionals, now there are two approaches to find a solution to a problem.

A first approach is a reactive approach whereby we try to find a solution to the problem at hand only after it arrives.

In the second approach which can be called a Proactive Approach whereby we first prepare ourselves well before the problem arrives with our past experiences and then with our past experience, we try to find a solution to the challenge when it arrives.

Estimation can thus be considered as a technique that is applied when we take a proactive approach to the problem.

Thus Estimation can be used to predict how much effort with respect to time and cost would be required to complete a defined task.

Once the testing team is able to make an estimate of the problem at hand then it is easier for them to come up with a solution that would be optimum to the problem at hand.

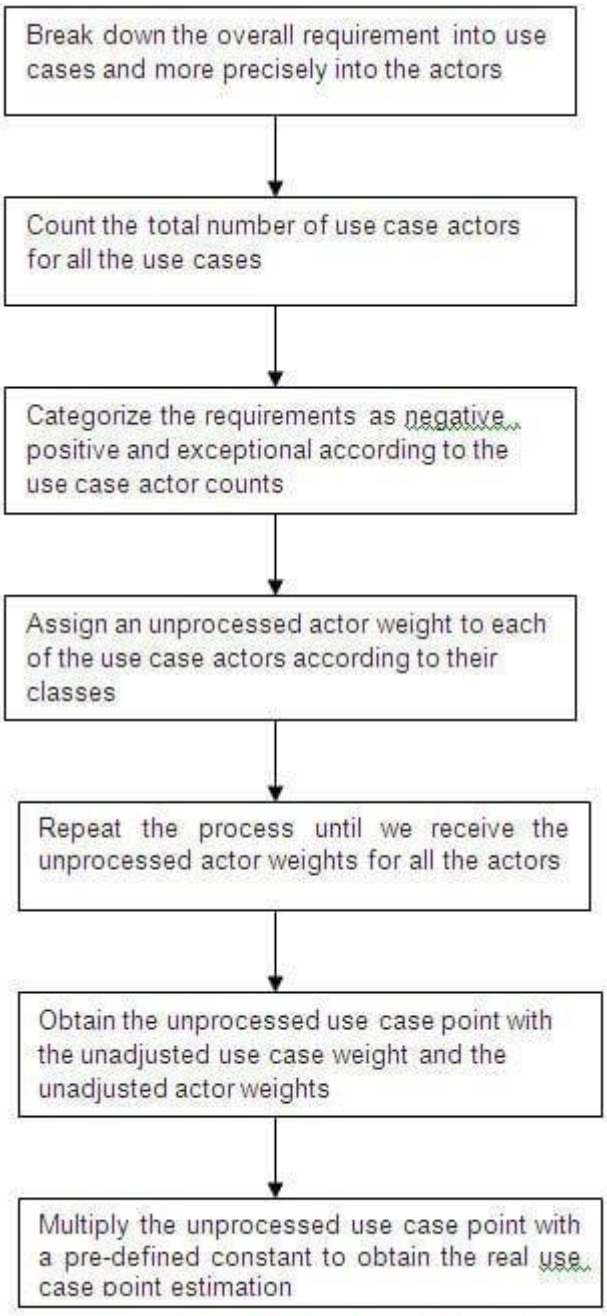
The practice of estimation can be defined then more formally as an approximate computation of the probable cost of a piece of work.

Test Estimation Examples

Use Case Point Estimation Method:

Use-Case Point Method is based on the use cases where we calculate the overall test estimation effort based on the use cases or the requirements.

Here is the detailed process of the Use case point estimation method:



An example of the same is that say in a particular requirement we have 5 use cases, use case 1, and use case 2… use case 5 respectively. Now let us consider that use case 1 consists of 6 actors, use case 2 consists of 15 actors, use cases 3, 4 and 5, 3, 4 and 5 actors respectively.

We consider any use case which involves the total number of actors as less than 5 as negative, any use case with the total number of actors is equal to or more than 5 and less than or equal to 10 as positive and any use case with more than 10 actors as exceptional.

We decide to assign 2 points to the exceptional use cases, 1 to the positive ones and -1 for the negative ones.

Thus we categorize the Use cases 1 and 5 as positive, use case 2 as exceptional and use

case 3, 4 as negative respectively based on our above-stated assumptions.
So the Unprocessed actor weights = Use case 1 = (total number of actors) 5 * 1(the assigned point) = 5. Similarly
Use case 2 = 15 * 2 = 30.
Repeating the process for rest of the use cases we receive the Unprocessed actor weights = 33
Unprocessed use case weight = total no. of use cases = 5
Unprocessed use case point = Unadjusted actor weights + Unadjusted use case weight = 33 + 5 = 38
Processed use case point = 38 * [0.65+ (0.01 * 50] = 26.7 or 28 Person Hours approximately

Function point metric:

A **Function Point** (FP) is a unit of measurement to express the amount of business functionality, an information system (as a product) provides to a user. FPs measure software size. They are widely accepted as an industry standard for functional sizing. For sizing software based on FP, several recognized standards and/or public specifications have come into existence. As of 2013, these are −
ISO Standards

- **COSMIC** − ISO/IEC 19761:2011 Software engineering. A functional size measurement method.
- **FiSMA** − ISO/IEC 29881:2008 Information technology - Software and systems engineering - FiSMA 1.1 functional size measurement method.
- **IFPUG** − ISO/IEC 20926:2009 Software and systems engineering - Software measurement - IFPUG functional size measurement method.
- **Mark-II** − ISO/IEC 20968:2002 Software engineering - Ml II Function Point Analysis - Counting Practices Manual.
- **NESMA** − ISO/IEC 24570:2005 Software engineering - NESMA function size measurement method version 2.1 - Definitions and counting guidelines for the application of Function Point Analysis.

Object Management Group Specification for Automated Function Point
Object Management Group (OMG), an open membership and not-for-profit computer industry standards consortium, has adopted the Automated Function Point (AFP) specification led by the Consortium for IT Software Quality. It provides a standard for automating FP counting according to the guidelines of the International Function Point User Group (IFPUG).
Function Point Analysis (FPA) technique quantifies the functions contained within software in terms that are meaningful to the software users. FPs consider the number of functions being developed based on the requirements specification.
Function Points (FP) Counting is governed by a standard set of rules, processes and

guidelines as defined by the International Function Point Users Group (IFPUG). These are published in Counting Practices Manual (CPM).

History of Function Point Analysis

The concept of Function Points was introduced by Alan Albrecht of IBM in 1979. In 1984, Albrecht refined the method. The first Function Point Guidelines were published in 1984. The International Function Point Users Group (IFPUG) is a US-based worldwide organization of Function Point Analysis metric software users. The **International Function Point Users Group (IFPUG)** is a non-profit, member-governed organization founded in 1986. IFPUG owns Function Point Analysis (FPA) as defined in ISO standard 20296:2009 which specifies the definitions, rules and steps for applying the IFPUG's functional size measurement (FSM) method. IFPUG maintains the Function Point Counting Practices Manual (CPM). CPM 2.0 was released in 1987, and since then there have been several iterations. CPM Release 4.3 was in 2010.

The CPM Release 4.3.1 with incorporated ISO editorial revisions was in 2010. The ISO Standard (IFPUG FSM) - Functional Size Measurement that is a part of CPM 4.3.1 is a technique for measuring software in terms of the functionality it delivers. The CPM is an internationally approved standard under ISO/IEC 14143-1 Information Technology – Software Measurement.

Elementary Process (EP)

Elementary Process is the smallest unit of functional user requirement that −

- Is meaningful to the user.
- Constitutes a complete transaction.
- Is self-contained and leaves the business of the application being counted in a consistent state.

Functions

There are two types of functions −

- Data Functions
- Transaction Functions

Data Functions

There are two types of data functions −

- Internal Logical Files
- External Interface Files

Data Functions are made up of internal and external resources that affect the system.

Internal Logical Files

Internal Logical File (ILF) is a user identifiable group of logically related data or control information that resides entirely within the application boundary. The primary intent of an ILF is to hold data maintained through one or more elementary processes of the application being counted. An ILF has the inherent meaning that it is internally

maintained, it has some logical structure and it is stored in a file. (Refer Figure 1)

External Interface Files

External Interface File (EIF) is a user identifiable group of logically related data or control information that is used by the application for reference purposes only. The data resides entirely outside the application boundary and is maintained in an ILF by another application. An EIF has the inherent meaning that it is externally maintained, an interface has to be developed to get the data from the file. (Refer Figure 1)

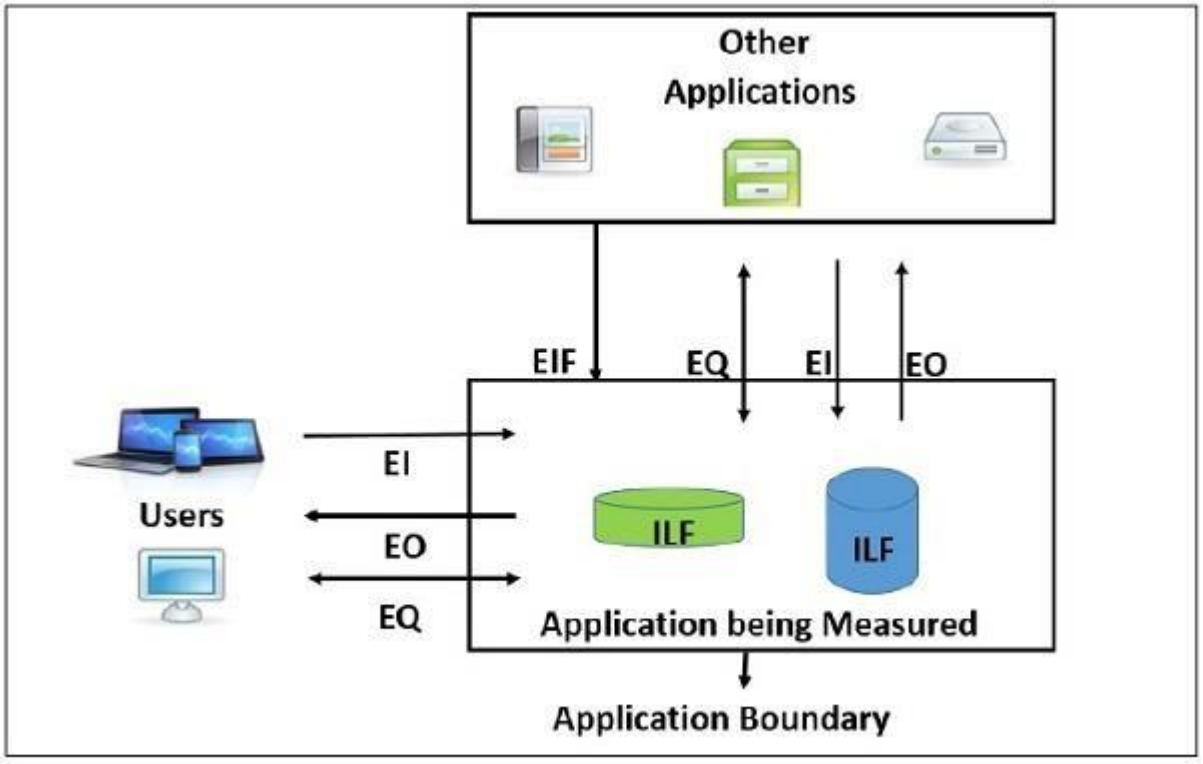


Figure 1: Application Boundary, Data Functions, Transaction Functions

Transaction Functions

There are three types of transaction functions.

- External Inputs
- External Outputs
- External Inquiries

Transaction functions are made up of the processes that are exchanged between the user, the external applications and the application being measured.

External Inputs

External Input (EI) is a transaction function in which Data goes “into” the application from outside the boundary to inside. This data is coming external to the application.

- Data may come from a data input screen or another application.
- An EI is how an application gets information.
- Data can be either control information or business information.
- Data may be used to maintain one or more Internal Logical Files.

- If the data is control information, it does not have to update an Internal Logical File. (Refer Figure 1)

External Outputs

External Output (EO) is a transaction function in which data comes "out" of the system. Additionally, an EO may update an ILF. The data creates reports or output files sent to other applications. (Refer Figure 1).

External Inquiries

External Inquiry (EQ) is a transaction function with both input and output components that result in data retrieval. (Refer Figure 1).

Experiment-11

Problem Statement: Develop a tool which can be used for quantification of all the non-functional requirements

Procedure:

Quantification of desirable properties of a system is an integral part of the software engineering. Most of requirement analysis methods which do not consider nonfunctional requirements lead to serious software development problems. The problems faced because of non-functional requirements omissions are more critical than the problems of functional requirements omissions. Therefore, it is necessary to measure non-functional requirement during software development process. Since software reliability is the major concern for quality system and considered as one of the most desirable quality attributes of the system, hence its quantitative measurement is an essential part of the development of a quality system. Estimation of software reliability becomes more important for those applications where risk is major consideration. Software reliability is the probability to perform failure free operation and produce correct output for a specified time under specified conditions. Once the system is installed, it is not possible to test system reliability in an operational environment because failure data collected in an uncontrolled tested environment may be limited. In this case, faults may not be necessarily removed as failures are identified. Therefore, failures are considered as one of the factors affecting the software. Failure data need to be properly collected and measured during software development process for the assessment of reliability of software systems. Software reliability is measured by collecting the historical data and various distribution curves. As failures are identified, faults are removed from the system to increase reliability. The fault tree uses fuzzy set theory for the quantification of uncertainty in order to make the system reliable. In the software development process, software is designed as an integration of sub-systems instead of a large system. The uncertainty of the overall system is found on the basis of uncertainties in the failure probability of sub-systems. Thus reliability requirements apply to the individual subsystems rather than the whole system. In recent years, various researchers developed different approaches

to analyze software quality attributes such as delta oriented slicing, feature wise measurement, fuzzy logic , scenarios and markov models and scenarios and Bayesian Belief network (BN).

Reliability is a key non-functional requirement, which can be divided into three Sub-NFRs (Availability and Recoverability). A. Availability Every system must be alive for service when it is requested by end-users. Availability of system indicates how reliable system is during operational hours. Therefore Availability is considered as one of the important metric used to assess the performance of a system, accounting for reliability of a system. It is defined as the ability to perform a required function under specified conditions at a given point of time or over a given time interval. It is often expressed as uptime and downtime. Uptime refers to the capability to perform the intended task in a stated environment and downtime refers to not being able to perform the intended task in a stated environment. In order to make system available, uptime of the system must be high and downtime of the system should be low. To reduce downtime, system should be properly maintained by constant monitoring of hardware & software and using anti-malicious software. To increase system availability during peak hours, dynamic Load balancing policies can be used to distribute I/O and CPU requests across all available paths to the storage device and servers respectively.

Recoverability

Recoverability is another important metric used to assess the reliability of a system. It is very important to know how often system fail to deliver expected level of service. This helps to measure the amount of data loss and downtime that a business can endure and decide how to recover from these failures. Good architecture provides recoverability in the time specified in a service-level agreement. To make system recoverable, backup of data should be taken at regular intervals. If mirroring of data is properly maintained, system will be more recoverable in case of any failure or disaster.

FUZZY SETS

Computer programming languages such as C, Java, and COBOL are best suited to develop such software systems whose behavior can be represented by mathematical model or logical reasoning. Since mathematical models do not work to develop software systems which require human judgment and decision making capabilities, Zadeh introduced the framework of Fuzzy sets to deal with a poorly defined concept in a coherent and structured way. Examples of poorly defined concepts suitable for the application of Fuzzy logic are semantic variables, such as high reliability, good performance, low maintenance, etc. The basis for proposing fuzzy logic was that human being relies on imprecise expression such as high, good or excellent. But software based systems work on Boolean logic. In this context, he emphasizes that human beings are desired to achieve the highest possible precision without paying attention to the imprecise character of quality. He proposed the theory of fuzzy sets in a mathematical way to represent vagueness in linguistics and can be considered as a generalization of classical set theory. In a classical set, an object either belongs to the set (member) or does not belong to the set (non-member). It means objects satisfy the

membership of the set precisely. Whereas in Fuzzy sets membership of the objects is approximate which helps to represent the real world system in more refined way. Fuzzy set removes the sharp boundaries that separate members and non-members in a group. In this case, the transition from a member to a non-member is gradual. It means that an object can belong to a fuzzy set either fully or partially. For example, categorizing whether room temperature is hot or cold is decided by considering various degree of membership. This example supports the concept of approximate membership. According to Zadeh, the membership of an object to a fuzzy set is represented by a continuous function defined on closed interval [0, 1] of real numbers. This type of membership incorporates various degrees of membership and also supports the concept of membership/non-membership of classical sets. Thus, classical sets are the collection of objects but Fuzzy sets are the collection of functions which defines the membership of the objects to the interval [0, 1]. Advantage of fuzzy logic is its ability to express the amount of ambiguity in human thinking and subjectivity. It is best suited to the problems which are concerned with continuous variables that are not easily divided into discrete variables. Fuzzy sets are the best choice for managing vague, imprecise, doubtful, contradicting and diverging opinions. IV. BAYESIAN NETWORKS Bayesian belief network represents compact networks of probabilities that capture the probabilistic relationships between variables, as well as historical information on their relationships. From another perspective, Bayesian belief network is a combination of Bayesian probability theory and the concept of conditional independence. It is also known as belief network, causal graph, causal network or probabilistic network. Bayesian belief network allows for clear graphical representation of cause and effect; and are effective for modeling scenarios where some prior information is already known but input data is uncertain, vague, conflicting or partially unavailable. Bayesian network is a way of representing joint probability distribution using Direct Acyclic Graph network structure given a set of variables □. Nodes of graph represent random uncertain variables and the arcs represent the Bayesian probabilistic relationships between these variables. It is a collection of conditional probability distributions where each variable □□ in the directed acyclic graph □ is denoted by a conditional distribution given its parent nodes □□□□□□ [23]. Bayesian network is also known as a network of nodes of influences based on reasoning. It uses Bayes theorem to express conditional probability between each alternative. It is a powerful tool for modeling causes and effects in systems and is sometimes described as a marriage between probability theory and graphical theory [24]. Its advantages are as follows: • There is no need of exact historical data or evidence to produce convincing results. • Despite of uncertainties in the input information, it provides effective output. • It represents the causes and effects relationship between the nodes in the form of arcs connecting nodes. • It helps in diagnosing current situation based on the historical relationship between nodes. Bayesian Networks represent joint probability distribution consisting of two components: a) A directed acyclic graph G whose vertices correspond to random

variables □ৎ, □ᘏ,ڼ.□ b) Conditional probability distribution of every variable □□ given its parents □□□ᕦ□□ᕗ. Graph G encodes the Markov assumption "Every variable □□ given its parents □□□ᕦ□□ᕗ is conditional independent of nondescendants nodes". Joint probability distribution satisfying Markov assumption is defined by Chain Rule as follows [22]: □ᕦ□ৎ, □ᘏ,ڼ□□,ᕗ≌ ∏ □ⓑ□□ዘ□□□ᕦ□□ᕗᕦ□□ 1ৎᕦ1ᕗ In discrete Bayesian Networks, the conditional probability distribution represents a multivariate discrete distribution that parameterizes all possible combination of discrete states of Vi parents. V. PROPOSED FUZZY-BAYESIAN NETWORK Probability helps to understand the randomness, whereas Fuzzy Sets deals with vagueness. But Zadeh [25] combined probability measures and fuzzy sets together to measure the probability of fuzzy event events in terms of crisp number. Proposed Fuzzy-Bayesian integrated approach combines the representation power of Fuzzy Set theory and the algorithmic strength of Bayesian Networks to measure non-functional requirements. Fuzzy Bayesian approach requires transformation of Bayesian network to fuzzy domain and corresponding fuzzy conditional probability distributions. To complete the transformation, we need to define the following: a) Creation of fuzzy variables of each continuous variable. b) Creation of fuzzy Conditional probability distributions for each continuous variable. A. Creation of fuzzy variables Assume X is a continuous variable ranging from value {ي {to {ي .{□It is represented by the Equation 2. □≌ ᕼي :ياي□ᕗᕦ2ᕗ where x is the actual value of the variable X which is defined on the closed interval [ي ,ي .[□A group of different values of x create continuous variable X. Thus x is the crisp value that represents the variable X, which is defined on the discourse X. Now, consider the fuzzy set Xf defined on X, to be ᕦ3ᕗ□□ א ي |ᕗيᕦ, □□ي□≌ □□ where يא □on the closed interval [0, 1] and □□ᕦيᕗ defines membership of ي to the Fuzzy Set □□. Zadeh [19] defined Fuzzy sets as the collection of functions which defines the membership of the objects to the interval [0, 1]. Therefore, various fuzzy states □□□ৎ, □□ᘏ ,□□ᘓ,...,□□□□ of variable □ can be represented as follows: □□□≌ □ ي□□□ ,ᕦيᕗ| ي א □ ,□0 □□□ᕦيᕗ 1 (4) where □□□ᕦيᕗ denotes the degree of membership of ي to the fuzzy state □□□. FUZZY-BAYESIAN NETWORK FOR RELIABILITY The application presented here is illustrating the applicability of our research methodology. Fuzzy-Bayesian Network for Reliability non-functional requirement is given in Figure 1. Figure 1: General Hybrid Bayesian Network for Reliability This network includes discrete functional and non-functional requirements which have continuous non-functional requirements as parents. There are total 9 nodes in the FuzzyBayesian network of the Reliability requirement. All nodes A, B, C, D, E, F, G, H and I are representing various functional and non-functional aspects for the satisfaction of Reliability non-functional requirement. Nodes D, E, F, H and I represent discrete functional requirements such as Mirroring, Backup Frequency, Load Balancing, Constant H/W & S/W checking and Use of anti-malicious S/W. Nodes A, B, C and G represent continuous non-functional requirements such as Reliability, Recoverability,

Availability and Software Maintenance.

Experiment-12

Problem Statement: Write C/C++/Java/Python program for classifying the various types of coupling.

Procedure:

Coupling refers to the usage of an object by another object. It can also be termed as collaboration. This dependency of one object on another object to get some task done can be classified into the following two types −

- **Tight coupling** - When an object creates the object to be used, then it is a tight coupling situation. As the main object creates the object itself, this object cannot be changed from outside world easily marked it as tightly coupled objects.
- **Loose coupling** - When an object gets the object to be used from the outside, then it is a loose coupling situation. As the main object is merely using the object, this object can be changed from the outside world easily marked it as loosely coupled objects.

Example - Tight Coupling

Tester.java

```
publicclassTester{
  publicstaticvoid main(Stringargs[]){
    A a=newA();

    //a.display() will print A and B
    //this implementation can not be changed dynamically
    //being tight coupling
    a.display();
  }
}

class A {
  B b;
  public A(){
```

```
    //b is tightly coupled to A
    b =newB();
  }

  publicvoid display(){
    System.out.println("A");
    b.display();
  }
}

class B {
  public B(){}
  publicvoid display(){
    System.out.println("B");
  }
}
```

Output

A

B

Example - Loose Coupling

Tester.java

```
importjava.io.IOException;

publicclassTester{
  publicstaticvoid main(Stringargs[])throwsIOException{
    Show b =newB();
    Show c =newC();

    A a=newA(b);
```

```
    //a.display() will print A and B
    a.display();

    A a1 =new A(c);
    //a.display() will print A and C
    a1.display();
  }
}

interfaceShow{
  publicvoid display();
}

class A {
  Show s;
  public A(Show s){
    //s is loosely coupled to A
    this.s= s;
  }

  publicvoid display(){
    System.out.println("A");
    s.display();
  }
}

class B implementsShow{
  public B(){}
  publicvoid display(){
    System.out.println("B");
  }
```

```
}

class C implementsShow{
  public C(){}
  publicvoid display(){
    System.out.println("C");
  }
}
```

Output

A

B

A

C

Experiment-13

Problem Statement: Write C/C++/Java/Python program for classifying the various types of coupling.

Procedure:

Cohesion in Java is the Object-Oriented principle most closely associated with making sure that a class is designed with a single, well-focused purpose. In object-oriented design, cohesion refers all to how a single class is designed.

The advantage of high cohesion is that such classes are much easier to maintain (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

Example: Suppose we have a class that multiplies two numbers, but the same class creates a pop-up window displaying the result. This is an example of a low cohesive class because the window and the multiplication operation don't have much in common. To make it high cohesive, we would have to create a class Display and a class Multiply. The Display will call Multiply's method to get the result and display it. This way to develop a high cohesive solution.

// Java program to illustrate

// high cohesive behavior

```
class Multiply
{
int a = 5;
int b = 5;
publicintmul(int a, int b)
{
this.a = a;
this.b = b;
return a * b;
}
}

class Display {
                        public static void main(String[] args)
                        {
                            Multiply m = new Multiply();
                            System.out.println(m.mul(5,   5));
                        }
}
```

Output

25

```
// Java program to illustrate
// high cohesive behavior

class Name {
        String name;
        public String getName(String name)
        {
                this.name = name;
                return name;
        }
}
```

```
class Age {
        int age;
        publicintgetAge(int age)
        {
                this.age = age;
                return age;
        }
}

class Number {
        intmobileno;
        publicintgetNumber(intmobileno)
        {
                this.mobileno = mobileno;
                returnmobileno;
        }
}

class Display {
        public static void main(String[] args)
        {
                Name n = new Name();
                System.out.println(n.getName("Geeksforgeeks"));
                Age a = new Age();
                System.out.println(a.getAge(10));
                Number no = new Number();
                System.out.println(no.getNumber(1234567891));
        }
```

}

Output
Geeksforgeeks

10

1234567891

Difference between high cohesion and low cohesion:

- High cohesion is when you have a class that does a well-defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
- High cohesion gives us better-maintaining facility and Low cohesion results in monolithic classes that are difficult to maintain, understand and reduce re-usability

Experiment-15

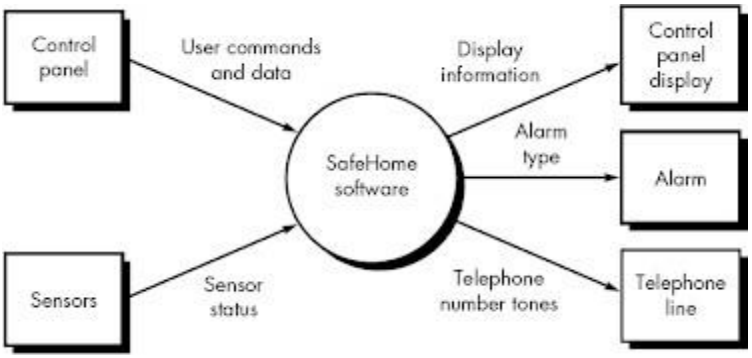
Problem Statement: Convert the DFD into appropriate architecture styles.

Procedure:

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. In this section transform mapping is described by applying design steps to an example system—a portion of the SafeHome security software.

An Example

The SafeHome security system is representative of many computer-based products and systems in use today. The product monitors the real world and reacts to changes that it encounters. It also interacts with a user through a series of typed inputs and alphanumeric displays. The level 0 data flow diagram for SafeHome, is shown in figure

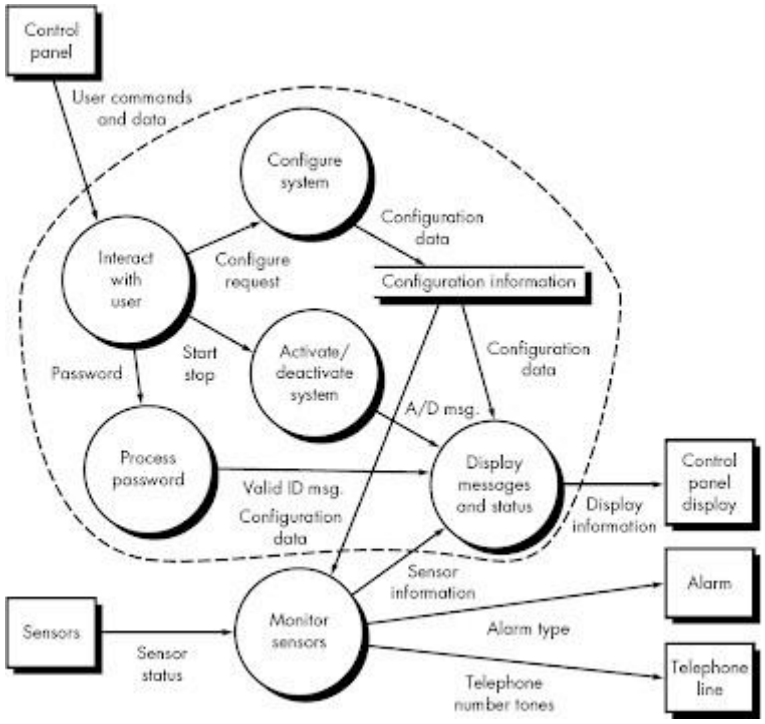


During requirements analysis, more detailed flow models would be created for SafeHome. In addition, control and process specifications, a data dictionary, and various behavioral models would also be created.

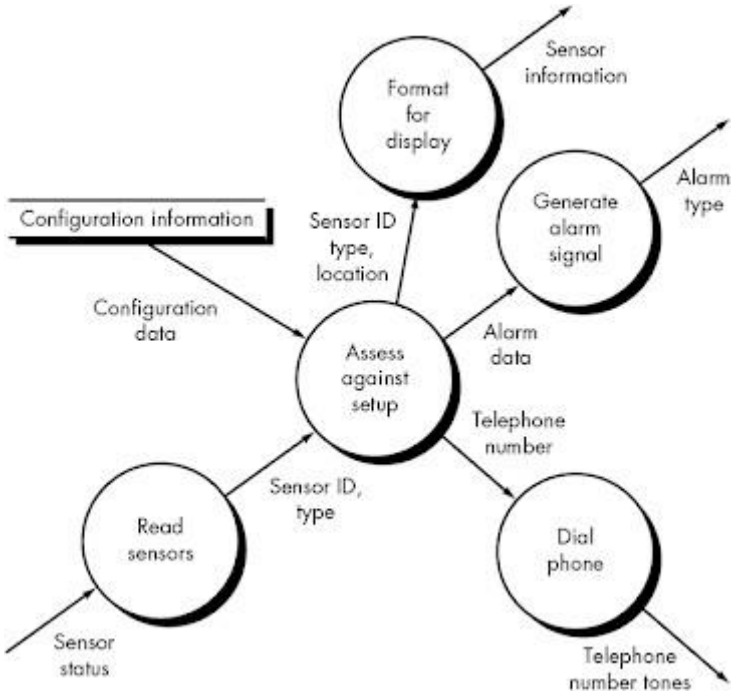
Design Steps

The preceding example will be used to illustrate each step in transform mapping. The steps begin with a re-evaluation of work done during requirements analysis and then move to the design of the software architecture.

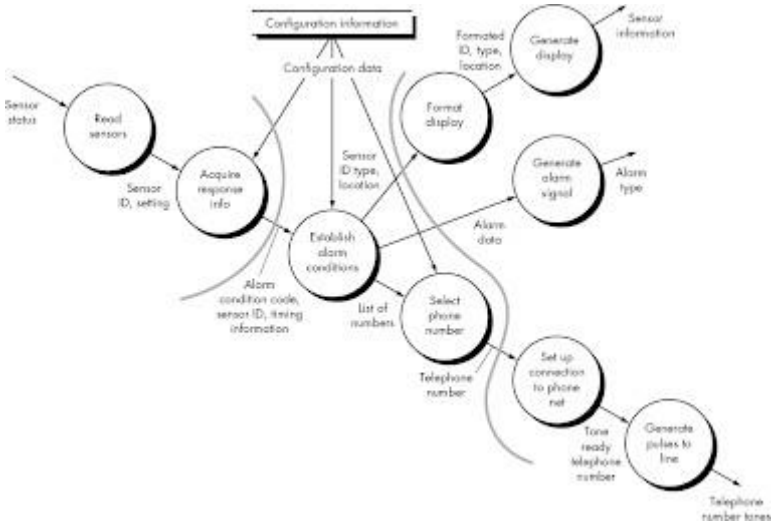
Step 1. Review the fundamental system model. The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the System Specification and the Software Requirements Specification. Both documents describe information flow and structure at the software interface. Figure 1 and 2 depict level 0 and level 1 data flow for the SafeHome software.



Step 2. Review and refine data flow diagrams for the software. Information obtained from analysis models contained in the Software Requirements Specification is refined to produce greater detail. For example, the level 2 DFD for monitor sensors is examined, and a level 3 data flow diagram is derived . At level 3, each transform in the data flow diagram exhibits relatively high cohesion. That is, the process implied by a transform performs a single, distinct function that can be implemented as a module9 in the SafeHome software. Therefore, the DFD in figure contains sufficient detail for a "first cut" at the design of architecture for the monitor sensors subsystem, and we proceedwithoutfurtherrefinement.



Step 3. Determine whether the DFD has transform or transaction flow characteristics. In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic is encountered, a different design mapping is recommended. In this step, the designer selects global (softwarewide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These subflows can be used to refine program architecture derived from a global characteristic described previously. For now, we focus our attention only on the monitor sensors subsystem data flow depicted in figure.



Evaluating the DFD , we see data entering the software along one incoming path and

exiting along three outgoing paths. No distinct transaction center is implied (although the transform establishes alarm conditions that could be perceived as such). Therefore, an overall transform characteristic will be assumed for information flow.

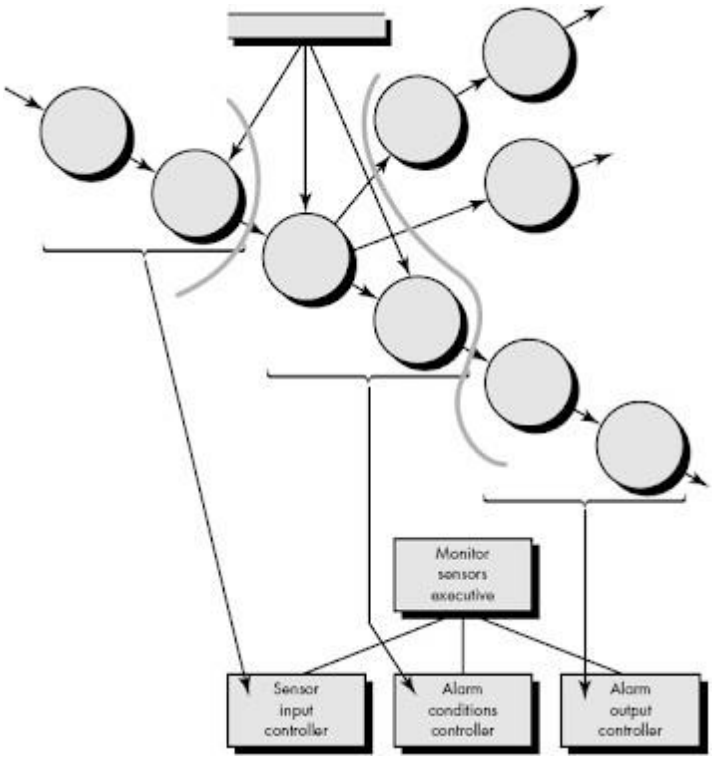
Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in the above figure. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g, an incoming flow boundary separating read sensors and acquire response info could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

Step 5. Perform "first-level factoring." Program structure represents a top-down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.

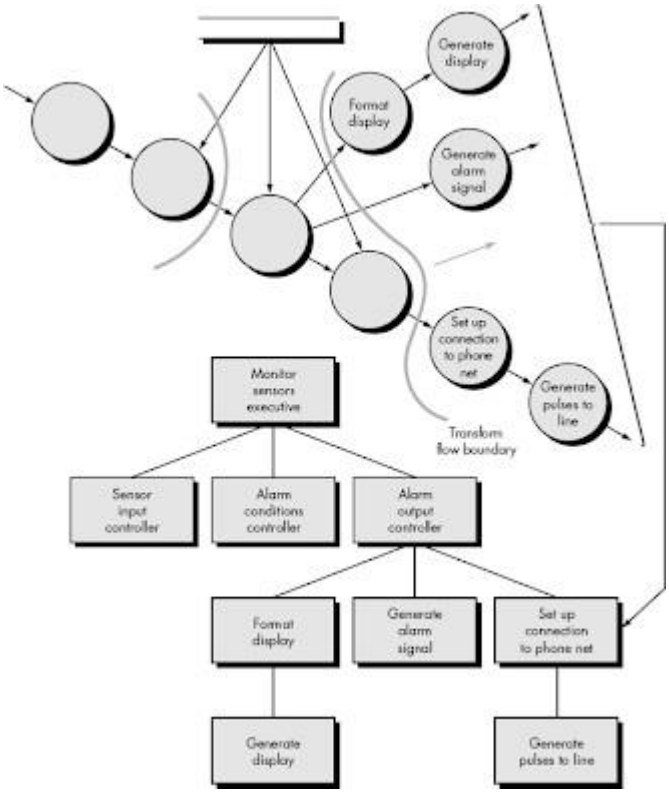
When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the monitor sensors subsystem is illustrated in figure below. A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:

- An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.
- A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).
- An outgoing information processing controller, called alarm output controller, coordinates production of output information.



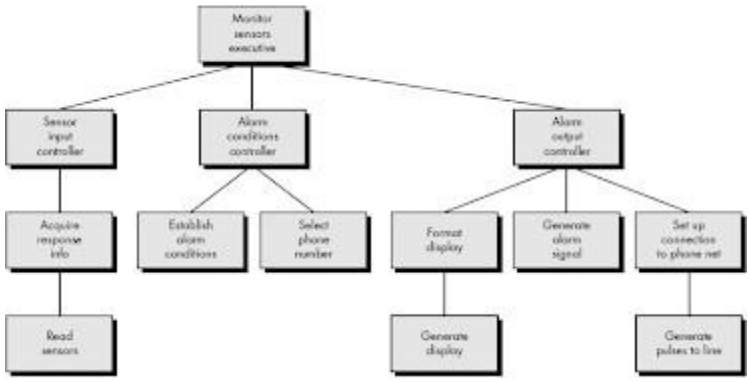
Although a three-pronged structure is implied by figure complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good coupling and cohesion characteristics.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second-level factoring for the SafeHome data flow is illustrated in figure.



Although the figure illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one module (recalling potential problems with cohesion) or a single bubble may be expanded to two or more modules. Practical considerations and measures of design quality dictate the outcome of secondlevel factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of monitor sensors subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in figure.



The modules mapped in the preceding manner and shown in figure represent an initial design of software architecture. Although modules are named in a manner that implies function, a brief processing narrative (adapted from the PSPEC created during analysis modeling) should be written for each. The narrative describes

- Information that passes into and out of the module (an interface description).
- Information that is retained by a module, such as data stored in a local data structure.
- A procedural narrative that indicates major decision points and tasks.
- A brief discussion of restrictions and special features (e.g., file I/O, hardwaredependent characteristics, special timing requirements).

The narrative serves as a first-generation Design Specification. However, further refinement and additions occur regularly during this period of design.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of module independence . Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refinements are dictated by the analysis and assessment methods described briefly , as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a module that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter. Many modifications can be made to the first iteration architecture developed for the SafeHome monitor sensors subsystem. Among many possibilities,

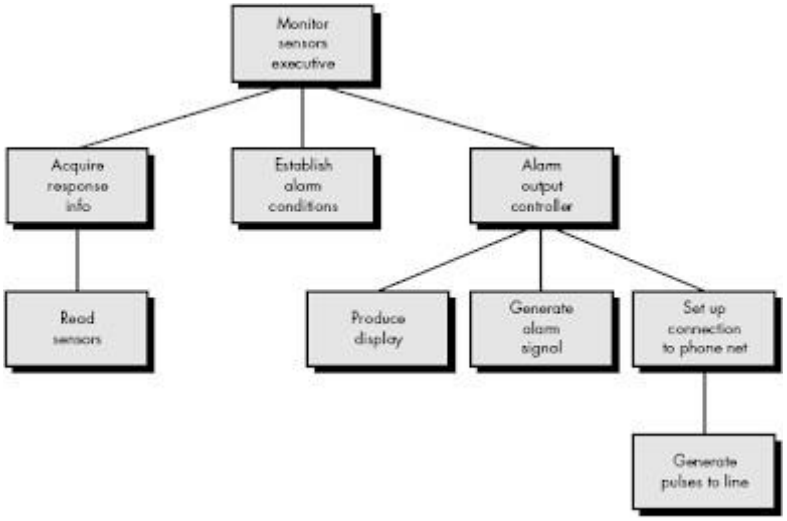
1. The incoming controller can be removed because it is unnecessary when a single

incoming flow path is to be managed.

2. The substructure generated from the transform flow can be imploded into the module establish alarm conditions (which will now include the processing implied by select phone number). The transform controller will not be needed and the small decrease in cohesion is tolerable.

3. The modules format display and generate display can be imploded (we assume that display formatting is quite simple) into a new module called produce display.

The refined software structure for the monitor sensors subsystem is shown in figure.



The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

Experiment-17

Problem Statement: Define the design activities along with necessary artifacts using Design Document.

Procedure:

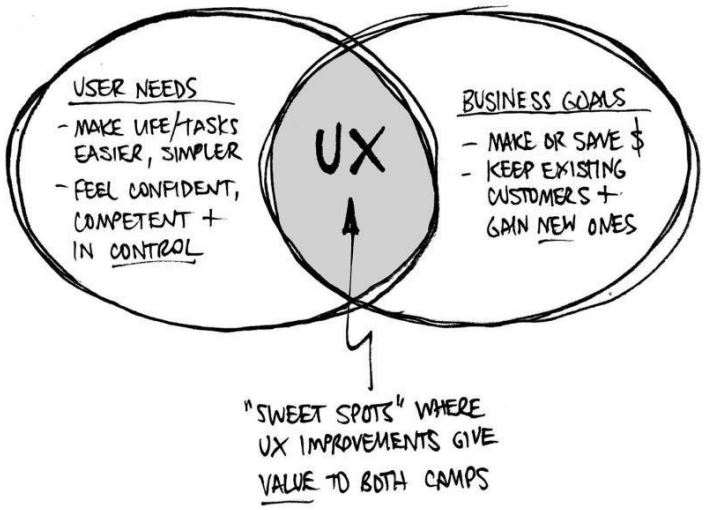
What is design documentation?

Design documentation is a collection of documents and resources that covers all aspects of your product design. Documentation should include information about users, product features, and project deadlines; all essential implementation details; and design decisions that your team and stakeholders have agreed on.

Why invest in documentation design?

Clarify project requirements

Gaining stakeholder approval to begin implementing a design is one of the most important steps in the design process. You need to be on the same page with stakeholders to gain this approval. Proper documentation makes it easier to achieve this goal. How? Documentation helps you organize and deliver your thoughts to stakeholders, which in turn helps them understand how your design decisions will satisfy the user needs and their own business objectives.



Designers need to find a sweet spot between business goals and user needs. Image

credit UX Booth.

Streamline design implementation

By documenting a design, you also aid in the implementation of it. Product design is a collaborative process, and in many cases, multiple people work on the project. It's not always possible to share implementation details verbally (for example, when you work with remote teams). Thus, the design documents act as a single source of truth for everyone who is involved in product development and can rally your team around a specific goal.

Motivate your team

Good documentation tells a high-level story about the product and gets team members excited about the vision. It answers the questions, "How do we want to build this?" and, importantly, "Why do we want to build this?"

A list of essential docs

While documentation can vary from project to project, the following docs will be relevant to all. This information can be included in a single document or separated into multiple documents. Which approach you take will depend on the complexity of your project.

- **Project overview** – This document contains a high-level overview of the design and the goals the design team wants to accomplish. By reading this document, anyone should be able to understand the purpose of a project.
- **Product requirements** – This document covers the business and technical requirements of the design. It should be shared with stakeholders before starting the design to ensure that both types of requirements are satisfied. It's also worth including in this doc information about constraints and assumptions because they will influence the design decisions.
- **Project deliverables** – This document provides information about the design artifacts established during the wireframing and prototyping phases (e.g., lo-fi wireframes,

mock-ups, hi-fi prototypes) that will be provided as deliverables once implementation has been completed.

- **Target audience information** – This document lists relevant information about your audience, from user personas to data from user research. This information will help your team understand who your users are and what good design means to them (via their functional and aesthetic preferences). The doc serves as a reference for designers when sharing their rationale behind individual design decisions.
- **User journeys** – This document outlines the path a user may take to reach their goal when using a product.
- **Design guidelines** – This document describes the components and specifications required to build the solution.
- **Style guides** – This document lists a set of standards for the stylization of design. Styles, colors, and typefaces are essential pieces of this guide.
- **Project scope and implementation plan** – This document describes the roles and flow of cross-team collaboration. The implementation plan documents the requirements necessary to complete the implementation of the design. For simple projects, it might be a high-level overview of the steps required to complete the implementation. For complex projects, it can include a project timeline with information about the time required to complete each of the steps.
- **Design validation and user testing** – This document provides an overview of the practices to be executed during the product design cycle, as well as steps to be taken after product release to verify that the product satisfies user needs.
- **Operational instructions** – This document provides detailed instructions on how to perform common operational tasks after the design is implemented. For example, it can provide step-by-step instructions on how to roll out a new version of an app in the production environment.

Properly documenting design

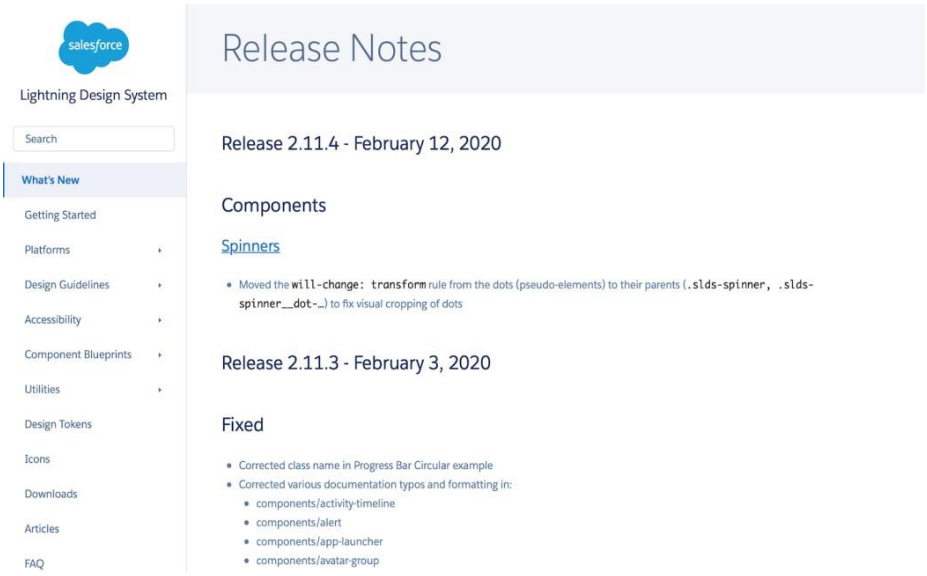
Though there's no single way to conduct design documentation, and it varies by product team, there are a few general recommendations that can benefit every project.

Make documentation usable for the target audience

It's possible to identify three large groups of users for documentation: product team members, stakeholders, and end users. Every group has its own needs, and it's important to consider this fact when working on your docs. Both the content of and the format for documentation should be adapted to suit your target audience.

Provide up-to-date documentation

Introduce a version control framework to keep your documentation up-to-date and therefore minimize the risk of incorrect design decisions. UX managers should validate the documentation at least once a month.



Release notes for Salesforce's Lightning Design System feature the release date. Image credit Salesforce.

Work on design documentation incrementally

Documentation design isn't a one-and-done activity. In many cases, it's impossible to create all the docs in one attempt. Thus, product teams should work on documentation as they go through the project. Documentation should be a "living" project that is constantly updated as you work on the design. Product teams should invest time in creating a flexible, accessible structure—anyone from a team should be able to update documentation rather effortlessly.

Test documentation

Documentation is a by-product of your product design, and like other products, it should be tested with users. Ensure that users know how to use it and find the documentation valuable. You can also introduce a simple feedback loop, such as an online response form, so your users can record their reactions and help you continuously improve your documentation.

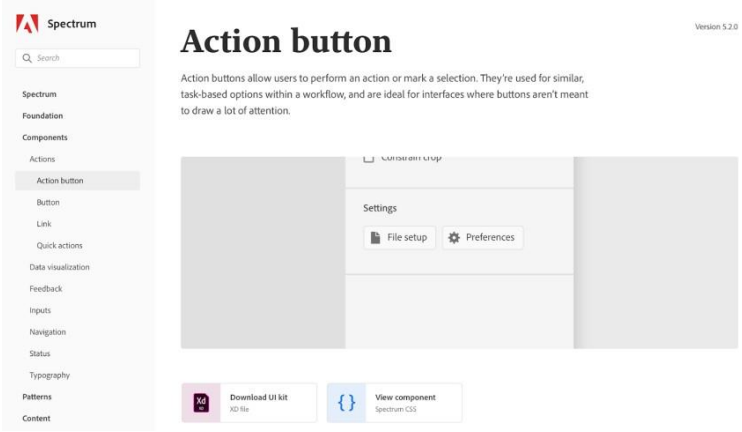
Avoid jargon

Every field has its own special language. When used in an appropriate context, this special language helps you communicate precisely with specialists that have the required expertise. But when you are uncertain about the expertise of your target audience, minimize the use of technical language in your documents.

The best documentation is the kind that your target audience can easily understand. It's important to learn what's appropriate for your audience and leave out jargon if it can be replaced by more familiar terms. Try reading the text out loud and evaluating it from the perspective of your documentation readers. Note any terms that might cause confusion and replace them with clearer terms.

Create easy access

Static paper-based documentation is quickly becoming a thing of the past. Modern documentation should be provided as an online resource. This format not only makes it easier for users to access the documentation, but it also simplifies the procedure for updates. Prioritize sections with information, and make sure search works fine. The structure you choose should follow the pattern that users follow when browsing the documentation.



the Adobe Spectrum design system uses an easy-to-follow structure that makes documentation readable and its organization intuitive. Image credit Adobe.

Provide visual or code samples in the doc

It's much easier to use information when you can match it with an actual design. To create contextual hierarchies and improve comprehension, documentation should include visual design and code snippets, not just plain words. Visual design or code samples make it easier for users to translate the information into design decisions.

Adobe Spectrum pairs visual examples with a text description. By doing so, it simplifies user comprehension. Image credit Adobe.

Update documentation automatically

If some part of the design goes undocumented, it doesn't exist. If elements of the design system go undocumented, you run the risk of duplicating elements. Try to keep documentation up-to-date with your product's code by automating documentation. Rules and systems should be in place for documentation to be updated as soon as developers introduce a change in the front-end design. This includes both visual references and code samples.

Find patterns in existing docs and turn them into templates

Once you have created the documentation for a few projects, review the docs and try to identify common aspects of all the projects. Define templates for the standard parts to aid in the creation of design documentation. Templates will also serve as a foundation for building out design documents for your future projects.

Conclusion

Creating design documentation is an important step in the project design process and has a direct impact on the outcome. The best design documentation gives a product team a framework for making design decisions. No matter how tight your deadlines for creating a design, you should never overlook the documentation process.

Experiment-18

Problem Statement: Reverse Engineer any object-oriented code to an appropriate class and object diagrams.

Procedure:

Introduction

In software evolution and maintenance, the ultimate, most reliable description of a system is its source code. Reverse engineering aims at extracting abstract, goal-oriented views from the code, to summarize relevant properties of program computations.

Reverse Engineering of Object Oriented Code presents a unifying framework for the analysis of object oriented code. Using Unified Modeling Language (UML) to represent the extracted design diagrams, the book explains how to recover them from object oriented code, thereby enabling developers to better comprehend their product and evaluate the impact of changes to it. Furthermore, it describes the algorithms involved in recovering views and demonstrates some of the techniques that can be employed for their visualization. The presentation is fully self-contained.

Topics and Features:

*Provides unique, in-depth exposition of the core concepts, principles, and methods behind reverse engineering object oriented code

*Explains the techniques and algorithms through numerous examples of object oriented code, the leading programming paradigm

*Focuses on fully automated design recovery, and deals with static and dynamic source-code analysis algorithms

*Explores code-centered analysis to obtain design diagrams aligned with the implementation

*Describes structural and behavioral views to offer a multi-perspective assessment of the system being analyzed

*Reports the analysis results in UML, the standard language for representing design diagrams in object oriented program development

This new state-of-the-art volume covers core methodologies for reverse engineering object oriented code, allowing for improved control in future code maintenance and modification. It is a significant resource for researchers and software engineers in the areas of reverse engineering, code analysis, object oriented programming, and UML. In addition, it will be invaluable as the reference book for advanced courses in these areas.

Experiment-19

Problem Statement: Test a piece of code which executes a specific functionality in the code to be tested and asserts a certain behavior or state using Junit.

Procedure:

JUnit is a popular unit-testing framework in the Java ecosystem. JUnit 5 added many new features based on the Java 8 version of the language.

Configuration for using JUnit 5

To use JUnit 5 you have to make the libraries available for your test code. Jump to the section which is relevant to you, for example read the Maven part, if you are using Maven as build system.

How to define a test in JUnit?

A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*. To mark a method as a test method, annotate it with the `@Test` annotation. This method executes the code under test.

The following code defines a minimal test class with one minimal test method.

```
packagecom.vogella.junit.first;

importstaticorg.junit.jupiter.api.Assertions.assertTrue;

importorg.junit.jupiter.api.Test;

classAClassWithOneJUnitTest{

@Test
voiddemoTestMethod(){
assertTrue(true);
}
}
```

You can use *assert* methods, provided by JUnit or another assert framework, to check an expected result versus the actual result. Such statement are called *asserts* or *assert statements*.

Assert statements typically allow to define messages which are shown if the test fails. You should provide here meaningful messages to make it easier for the user to identify and fix the problem. This is especially true if someone looks at the problem, who did not write the code under test or the test code.

Example for developing a JUnit 5 test for another class

The following example defines a Java class and defines software tests for it.

Assume you have the following class which you want to test.

```
packagecom.vogella.junit5;

publicclassCalculator{

publicintmultiply(inta,intb){
returna*b;
}
}
```

A test class for the above class could look like the following.

```
packagecom.vogella.junit5;

importstaticorg.junit.jupiter.api.Assertions.assertEquals;
```

```
importorg.junit.jupiter.api.BeforeEach;
importorg.junit.jupiter.api.DisplayName;
importorg.junit.jupiter.api.RepeatedTest;
importorg.junit.jupiter.api.Test;

classCalculatorTest{

Calculatorcalculator;

@BeforeEach
voidsetUp(){
calculator=newCalculator();
}

@Test
@DisplayName("Simple multiplication should work")
voidtestMultiply(){
assertEquals(20,calculator.multiply(4,5),
"Regular multiplication should work");
}

@RepeatedTest(5)
@DisplayName("Ensure correct handling of zero")
voidtestMultiplyWithZero(){
assertEquals(0,calculator.multiply(0,5),"Multiple with zero should be zero");
assertEquals(0,calculator.multiply(5,0),"Multiple with zero should be zero");
}
}
```

The method annotated with @BeforeEach runs before each test

A method annotated with @Test defines a test method

@DisplayName can be used to define the name of the test which is displayed to the user

This is an assert statement which validates that expected and actual value is the same, if not the message at the end of the method is shown

@RepeatedTest defines that this test method will be executed multiple times, in this example 5 times

JUnit test class naming conventions

Build tools like Maven use a pattern to decide if a class is a test classes or not. The following is the list of classes Maven considers automatically during its build:

```
**/Test*.java
**/*Test.java
**/*Tests.java
**/*TestCase.java
```

includes all of its subdirectories and all Java filenames that start with `Test`.

includes all of its subdirectories and all Java filenames that end with `Test`.

includes all of its subdirectories and all Java filenames that end with `Tests`.

includes all of its subdirectories and all Java filenames that end with `TestCase`.

Therefore, it is common practice to use the *Test* or *Tests* suffix at the end of test classes names.

Where should the test be located?

Typical, unit tests are created in a separate source folder to keep the test code separate from the real code. The standard convention from the Maven and Gradle build tools is to use:

- src/main/java - for Java classes
- src/test/java - for test classes

Static imports and unit testing

JUnit 5 allows to use static imports for its assertStatements to make the test code short and easy to read. Static imports are a Java feature that allows fields and methods defined in a class as `public static` to be used without specifying the class in which the field is defined.

JUnit assert statements are typically defined as `public static` to allow the developer to write short test statements. The following snippet demonstrates an assert statement with and without static imports.

```
// without static imports you have to write the following statement
importorg.junit.jupiter.api.Assertions;
// more code
Assert.assertEquals("10 x 5 must be 50",50,tester.multiply(10,5));
```

```
// alternatively define assertEquals as static import
importstaticorg.junit.jupiter.api.Assertions.assertEquals;
// more code
// use assertEquals directly because of the static import
assertEquals(calculator.multiply(4,5),20,"Regular multiplication should work");
```

2. Assertions and assumptions

JUnit 5 comes with multiple assert statements, which allows you to test your code under test. Simple assert statements like the following allow to check for true, false or equality. All of them are static methods from the `org.junit.jupiter.api.Assertions.*` package.

Assert statement	Example
assertEquals	assertEquals(4, calculator.multiply(2, 2),"optional failure message");
assertTrue	assertTrue('a' < 'b', () → "optional failure message");
assertFalse	assertFalse('a' > 'b', () → "optional failure message");
assertNotNull	assertNotNull(yourObject, "optional failure message");
assertNull	assertNull(yourObject, "optional failure message");

Messages can be created via lambda expressions, to avoid the overhead in case the construction of the message is expensive.

```
assertTrue('a'<'b',()->"Assertion messages can be lazily evaluated -- "
+"to avoid constructing complex messages unnecessarily.");
```

Experiment-20

Problem Statement: Test the percentage of code to be tested by unit test using any code coverage tools

Procedure:

Code coverage is a metric that can help you understand how much of your source is tested. It's a very useful metric that can help you assess the quality of your test suite, and we will see here how you can get started with your projects.

How is code coverage calculated?

Code coverage tools will use one or more criteria to determine how your code was exercised or not during the execution of your test suite. The common metrics that you might see mentioned in your coverage reports include:

- **Function coverage:** how many of the functions defined have been called.
- **Statement coverage:** how many of the statements in the program have been executed.
- **Branches coverage**: how many of the branches of the control structures (if statements for instance) have been executed.
- **Condition coverage:** how many of the boolean sub-expressions have been tested for a true and a false value.
- **Line coverage:** how many of lines of source code have been tested.

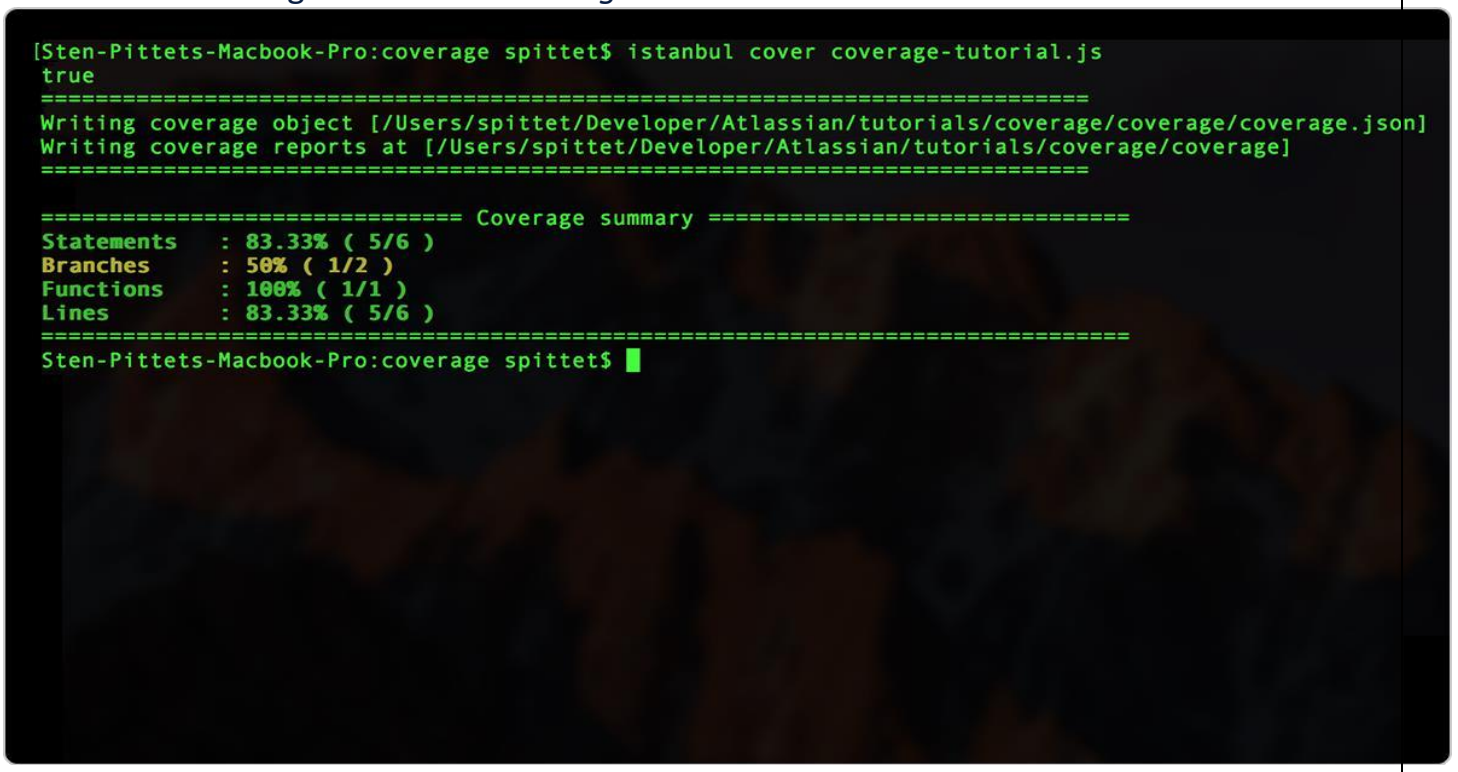
These metrics are usually represented as the number of items actually tested, the items found in your code, and a coverage percentage (items tested / items found).

These metrics are related, but distinct. In the trivial script below, we have a Javascript function checking whether or not an argument is a multiple of 10. We'll use that *function* later to check whether or not 100 is a multiple of 10. It'll help understand the difference between the function coverage and branch coverage.

coverage-tutorial.js

```
function isMultipleOf10(x) {  if (x % 10 == 0)    return true;  else   return false; } cons
ole.log(isMultipleOf10(100));
```

We can use the coverage tool istanbul to see how much of our code is executed when we run this script. After running the coverage tool we get a coverage report showing our coverage metrics. We can see that while our *Function Coverage* is 100%, our *Branch Coverage* is only 50%. We can also see that the isntanbul code coverage tool isn't calculating a *Condition Coverage* metric.

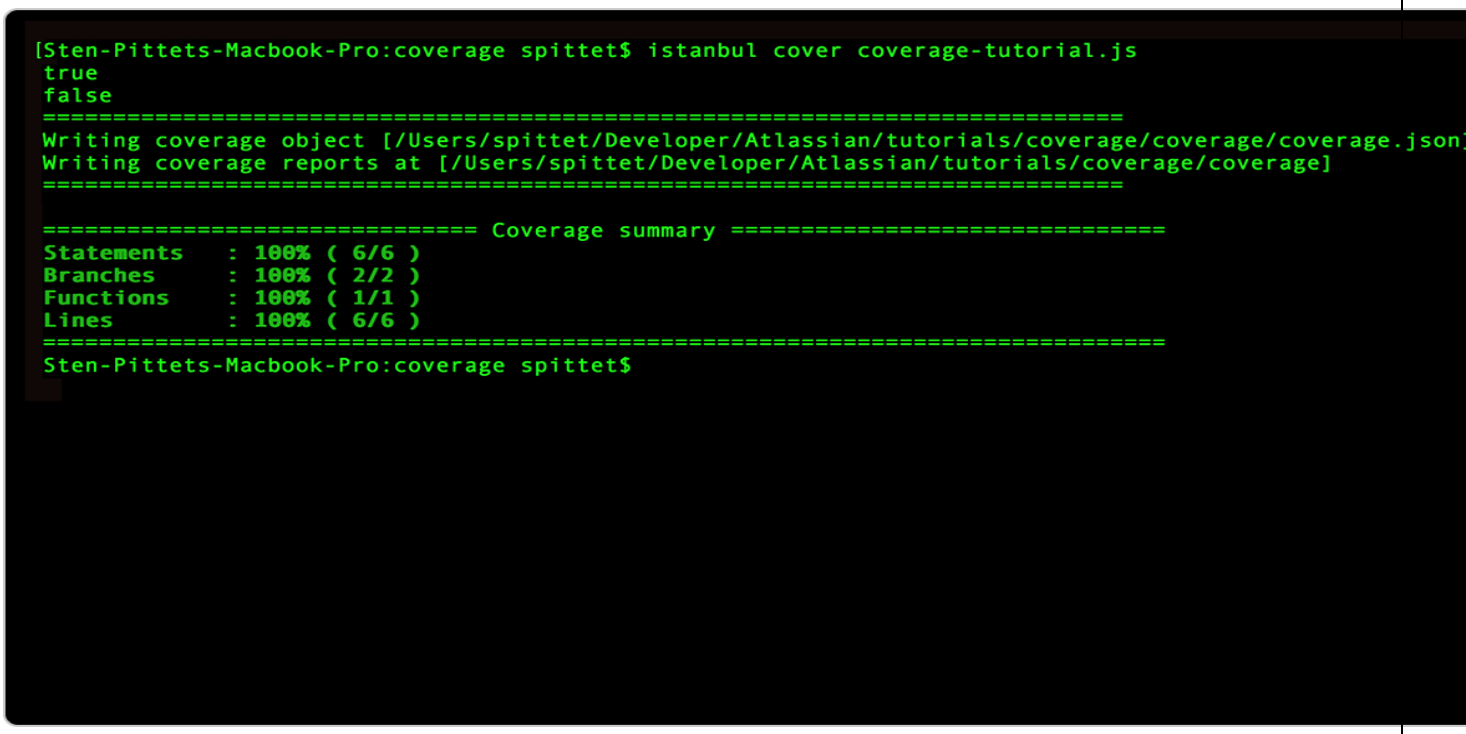


This is because when we run our script, the else statement has not been executed. If we wanted to get 100% coverage, we could simply add another line, essentially another test, to make sure that all branches of the if statement is used.

coverage-tutorial.js

```
function isMultipleOf10(x) { if (x % 10 == 0) return true; else return false; } cons
ole.log(isMultipleOf10(100)); console.log(isMultipleOf10(34)); // This will make our cod
e execute the "return false;" statement.
```

A second run of our coverage tool will now show that 100% of the source is covered thanks to our two console.log() statements at the bottom.



In this example, we were just logging results in the terminal but the same principal applies when you run your test suite. Your code coverage tool will monitor the execution of your test suite and tell you how much of the statements, branches, functions and lines were run as part of your tests.

Code coverage is a metric that can help you understand how much of your source is tested. It's a very useful metric that can help you assess the quality of your test suite, and we will see here how you can get started with your projects.

How is code coverage calculated?

Code coverage tools will use one or more criteria to determine how your code was exercised or not during the execution of your test suite. The common metrics that you might see mentioned in your coverage reports include:

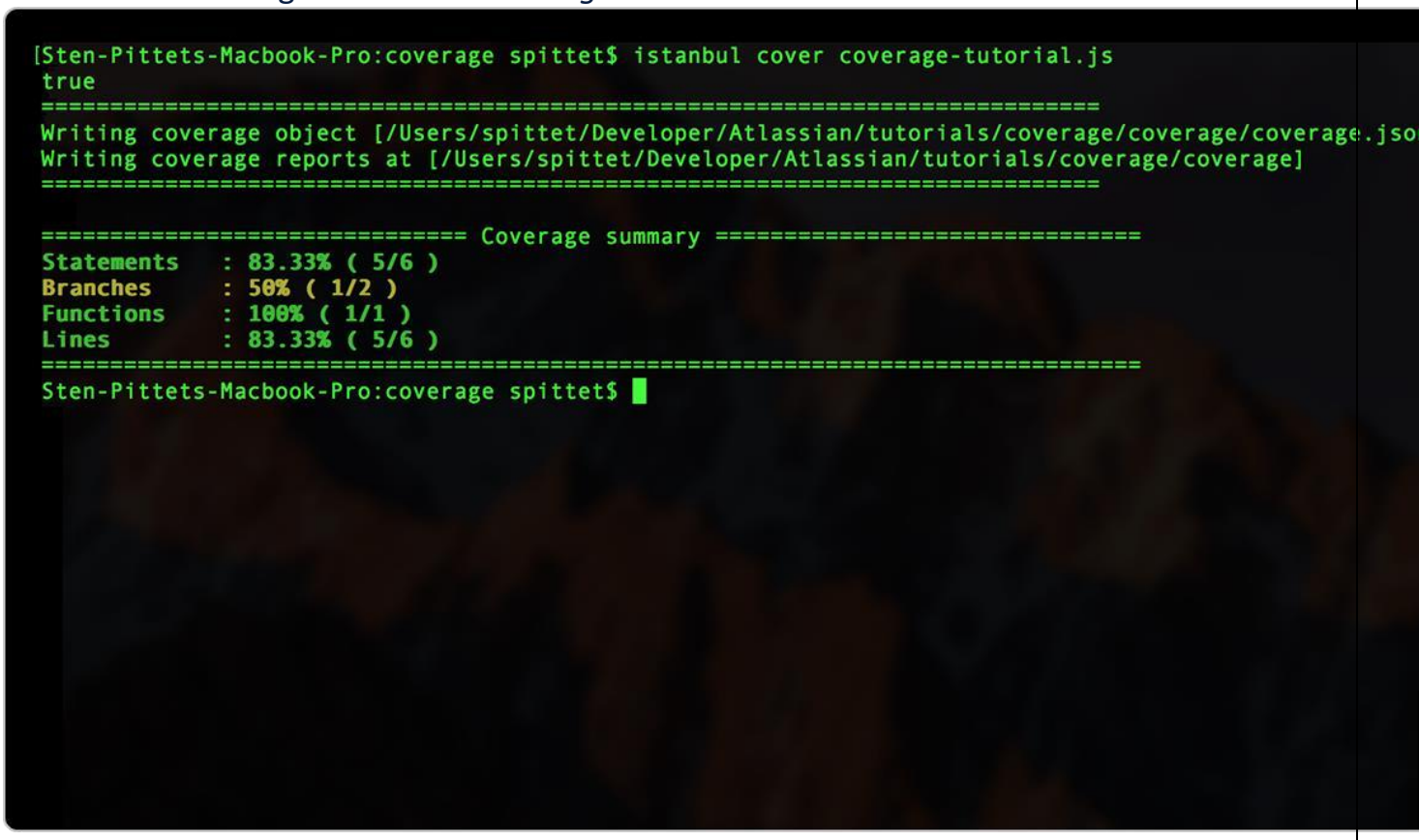
- **Function coverage:** how many of the functions defined have been called.
- **Statement coverage:** how many of the statements in the program have been executed.
- **Branches coverage**: how many of the branches of the control structures (if statements for instance) have been executed.
- **Condition coverage:** how many of the boolean sub-expressions have been tested for a true and a false value.
- **Line coverage:** how many of lines of source code have been tested.
  These metrics are usually represented as the number of items actually tested, the items found in your code, and a coverage percentage (items tested / items found).

These metrics are related, but distinct. In the trivial script below, we have a Javascript function checking whether or not an argument is a multiple of 10. We'll use that *function* later to check whether or not 100 is a multiple of 10. It'll help understand the difference between the function coverage and branch coverage.

coverage-tutorial.js

```
function isMultipleOf10(x) {  if (x % 10 == 0)    return true;  else   return false; } cons
ole.log(isMultipleOf10(100));
```

We can use the coverage tool istanbul to see how much of our code is executed when we run this script. After running the coverage tool we get a coverage report showing our coverage metrics. We can see that while our *Function Coverage* is 100%, our *Branch Coverage* is only 50%. We can also see that the isntanbul code coverage tool isn't calculating a *Condition Coverage* metric.



This is because when we run our script, the else statement has not been executed. If we wanted to get 100% coverage, we could simply add another line, essentially another test, to make sure that all branches of the if statement is used.

coverage-tutorial.js

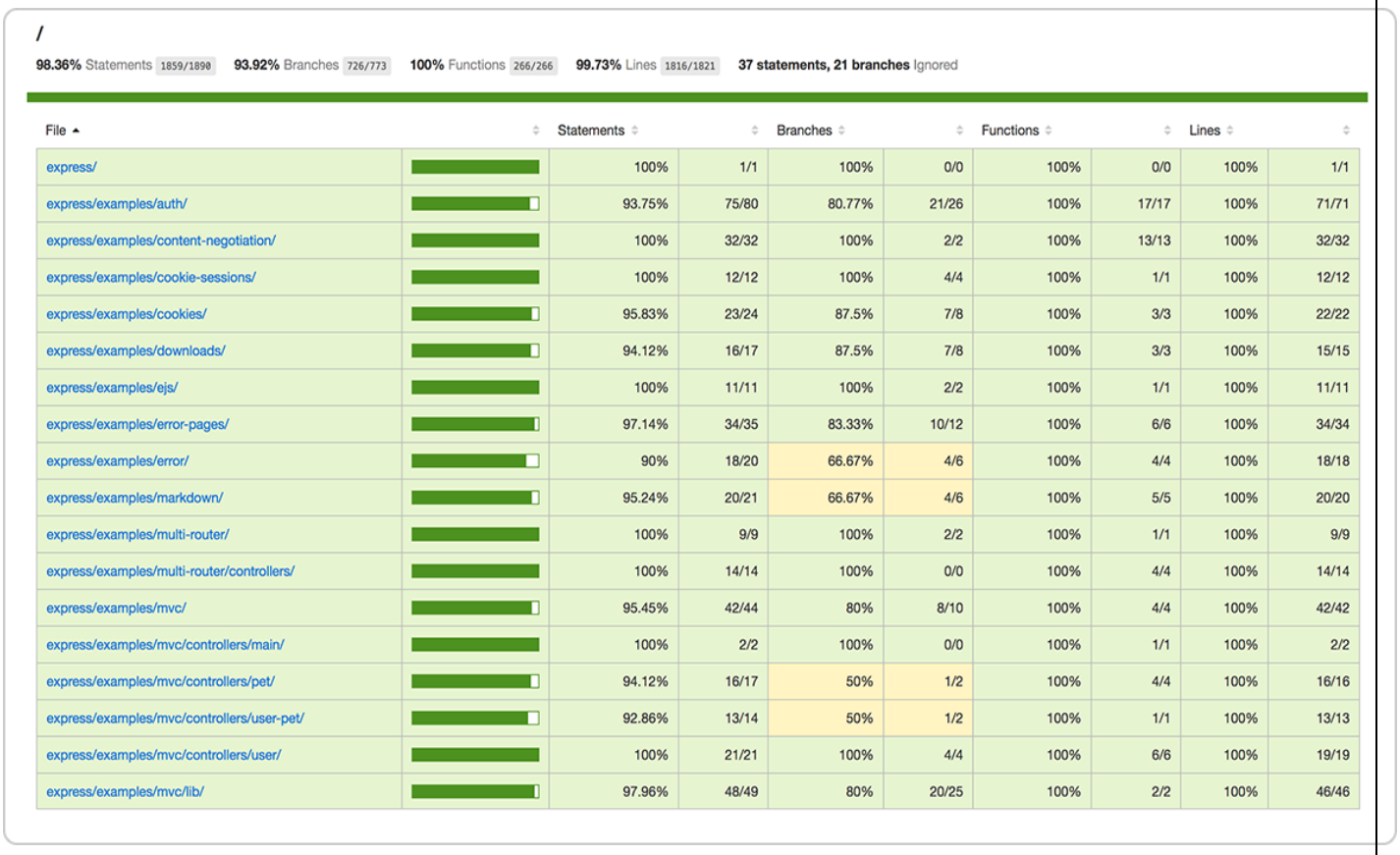
```
function isMultipleOf10(x) { if (x % 10 == 0) return true; else return false; } cons
ole.log(isMultipleOf10(100)); console.log(isMultipleOf10(34)); // This will make our cod
e execute the "return false;" statement.
```

A second run of our coverage tool will now show that 100% of the source is covered thanks to our two console.log() statements at the bottom.

```
[Sten-Pittets-Macbook-Pro:coverage spittet$ istanbul cover coverage-tutorial.js
 true
 false
 ================================================================================
 Writing coverage object [/Users/spittet/Developer/Atlassian/tutorials/coverage/coverage/coverage.json]
 Writing coverage reports at [/Users/spittet/Developer/Atlassian/tutorials/coverage/coverage]
 ================================================================================

 ============================== Coverage summary ==============================
 Statements   : 100% ( 6/6 )
 Branches     : 100% ( 2/2 )
 Functions    : 100% ( 1/1 )
 Lines        : 100% ( 6/6 )
 ==============================================================================
 Sten-Pittets-Macbook-Pro:coverage spittet$
```

In this example, we were just logging results in the terminal but the same principal applies when you run your test suite. Your code coverage tool will monitor the execution of your test suite and tell you how much of the statements, branches, functions and lines were run as part of your tests.



/

98.36% Statements 1859/1890 **93.92%** Branches 726/773 **100%** Functions 266/266 **99.73%** Lines 1816/1821 **37 statements, 21 branches** Ignored

File		Statements		Branches		Functions		Lines	
express/		100%	1/1	100%	0/0	100%	0/0	100%	1/1
express/examples/auth/		93.75%	75/80	80.77%	21/26	100%	17/17	100%	71/71
express/examples/content-negotiation/		100%	32/32	100%	2/2	100%	13/13	100%	32/32
express/examples/cookie-sessions/		100%	12/12	100%	4/4	100%	1/1	100%	12/12
express/examples/cookies/		95.83%	23/24	87.5%	7/8	100%	3/3	100%	22/22
express/examples/downloads/		94.12%	16/17	87.5%	7/8	100%	3/3	100%	15/15
express/examples/ejs/		100%	11/11	100%	2/2	100%	1/1	100%	11/11
express/examples/error-pages/		97.14%	34/35	83.33%	10/12	100%	6/6	100%	34/34
express/examples/error/		90%	18/20	66.67%	4/6	100%	4/4	100%	18/18
express/examples/markdown/		95.24%	20/21	66.67%	4/6	100%	5/5	100%	20/20
express/examples/multi-router/		100%	9/9	100%	2/2	100%	1/1	100%	9/9
express/examples/multi-router/controllers/		100%	14/14	100%	0/0	100%	4/4	100%	14/14
express/examples/mvc/		95.45%	42/44	80%	8/10	100%	4/4	100%	42/42
express/examples/mvc/controllers/main/		100%	2/2	100%	0/0	100%	1/1	100%	2/2
express/examples/mvc/controllers/pet/		94.12%	16/17	50%	1/2	100%	4/4	100%	16/16
express/examples/mvc/controllers/user-pet/		92.86%	13/14	50%	1/2	100%	1/1	100%	13/13
express/examples/mvc/controllers/user/		100%	21/21	100%	4/4	100%	6/6	100%	19/19
express/examples/mvc/lib/		97.96%	48/49	80%	20/25	100%	2/2	100%	46/46

Getting started with code coverage

Find the right tool for your project

You might find several options to create coverage reports depending on the language(s) you use. Some of the popular tools are listed below:

- Java: Atlassian Clover, Cobertura, JaCoCo
- Javascript: istanbul, Blanket.js
- PHP: PHPUnit
- Python: Coverage.py
- Ruby: SimpleCov

Some tools like istanbul will output the results straight into your terminal while others can generate a full HTML report that lets you explore which part of the code are lacking coverage.

What percentage of coverage should you aim for?

There's no silver bullet in code coverage, and a high percentage of coverage could still be problematic if critical parts of the application are not being tested, or if the existing tests are not robust enough to properly capture failures upfront. With that being said it is generally accepted that 80% coverage is a good goal to aim for. Trying to reach a higher coverage might turn out to be costly, while not necessary producing enough benefit.

The first time you run your coverage tool you might find that you have a fairly low percentage of coverage. If you're just getting started with testing it's a normal situation to be in and you shouldn't feel the pressure to reach 80% coverage right away. The reason is that rushing into a coverage goal might push your team to write tests that are hitting every line of the code instead of writing tests that are based on the business requirements of your application.

For instance, in the example above we reached 100% coverage by testing if 100 and 34 were multiples of 10. But what if we called our function with a letter instead of a number? Should we get a true/false result? Or should we get an exception? It is important that you give time to your team to think about testing from a user perspective and not just by looking at lines of code. Code coverage will not tell you if you're missing things in your source.

Focus on unit testing first

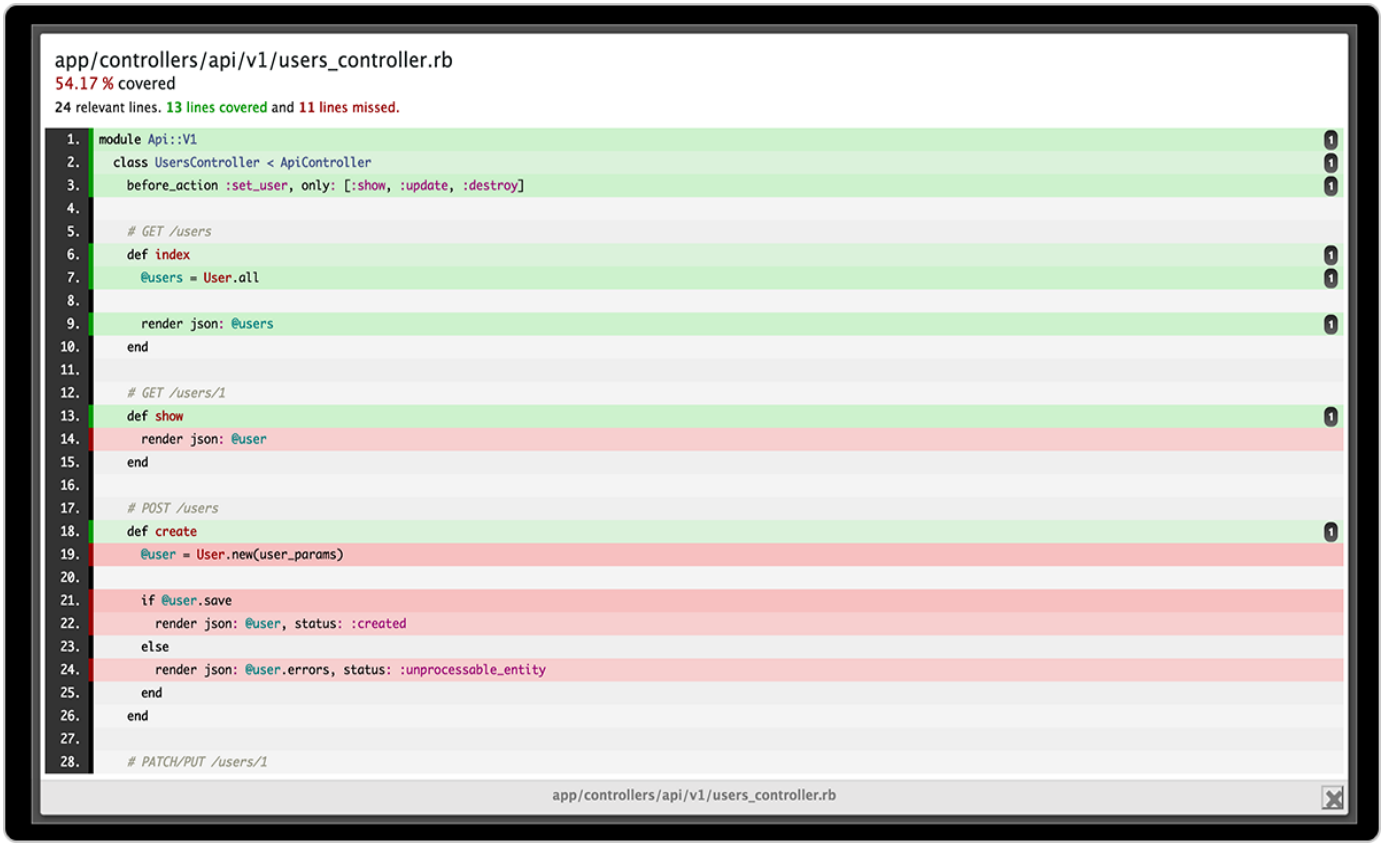
Unit tests consist in making sure that the individual methods of the classes and components used by your application are working. They're generally cheap to implement and fast to run and give you an overall assurance that the basis of the platform is solid. A simple way to increase quickly your code coverage is to start by

adding unit tests as, by definition, they should help you make sure that your test suite is reaching all lines of code.

Use coverage reports to identify critical misses in testing

Soon you'll have so many tests in your code that it will be impossible for you to know what part of the application is checked during the execution of your test suite. You'll know what breaks when you get a red build, but it'll be hard for you to understand what components have passed the tests.

This is where the coverage reports can provide actionable guidance for your team. Most tools will allow you to dig into the coverage reports to see the actual items that weren't covered by tests and then use that to identify critical parts of your application that still need to be tested.



Make code coverage part of your continuous integration flow when you're ready

When you've established your continuous integration (CI) workflow you can start failing the tests if you don't reach a high enough percentage of coverage. Of course, as we said it earlier, it would be unreasonable to set the failure threshold too high, and 90% coverage is likely to cause your build to fail a lot. If your goal is 80% coverage, you might consider setting a failure threshold at 70% as a safety net for your CI culture.

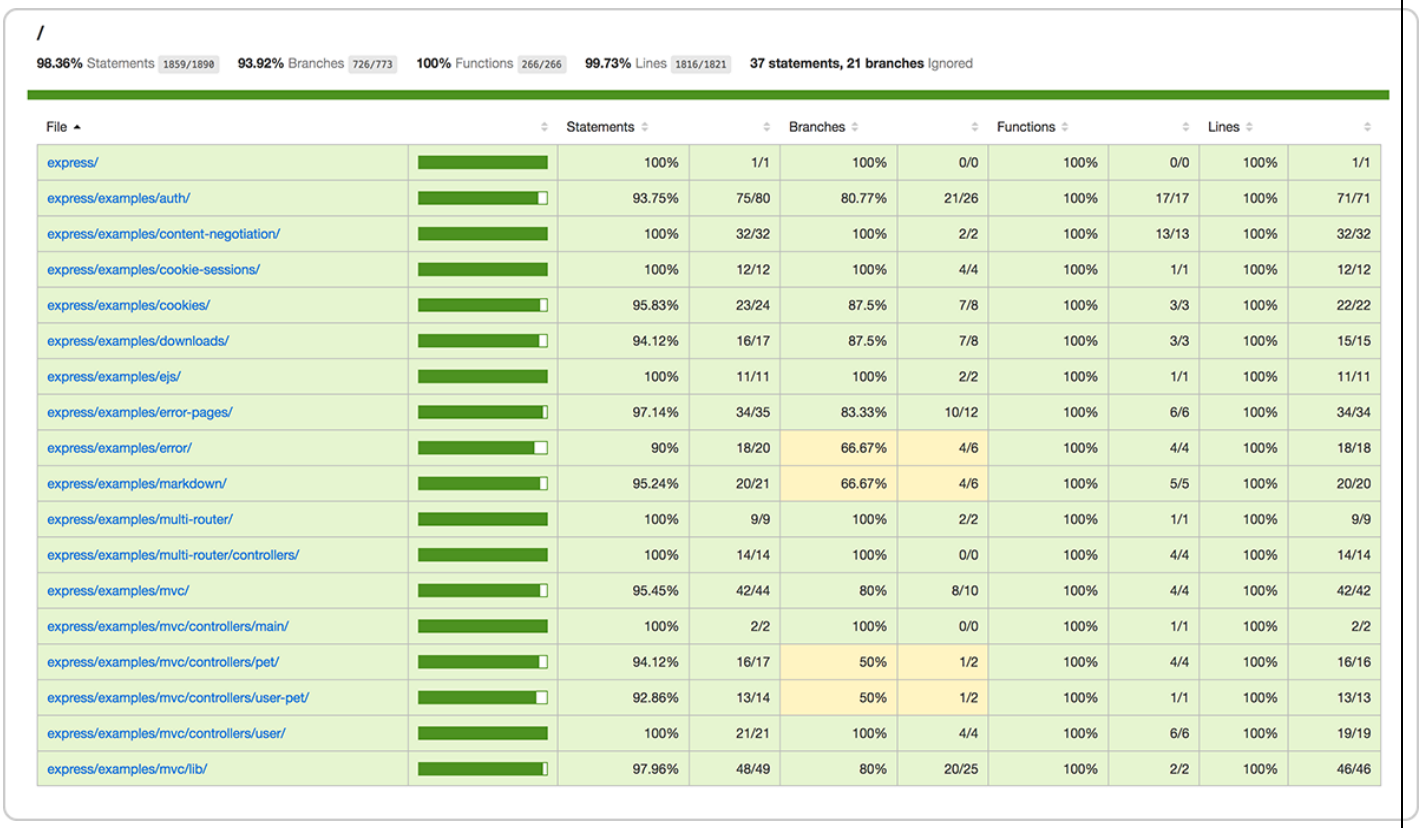
Once again, be careful to avoid sending the wrong message as pressuring your team to

reach good coverage might lead to bad testing practices.

Good coverage does not equal good tests

Getting a great testing culture starts by getting your team to understand how the application is supposed to behave when someone uses it properly, but also when someone tries to break it. Code coverage tools can help you understand where you should focus your attention next, but they won't tell you if your existing tests are robust enough for unexpected behaviors.

Achieving great coverage is an excellent goal, but it should be paired with having a robust test suite that can ensure that individual classes are not broken as well as verify the integrity of the system.



/

98.36% Statements 1859/1890 **93.92%** Branches 726/773 **100%** Functions 266/266 **99.73%** Lines 1816/1821 **37 statements, 21 branches** Ignored

File		Statements		Branches		Functions		Lines	
express/		100%	1/1	100%	0/0	100%	0/0	100%	1/1
express/examples/auth/		93.75%	75/80	80.77%	21/26	100%	17/17	100%	71/71
express/examples/content-negotiation/		100%	32/32	100%	2/2	100%	13/13	100%	32/32
express/examples/cookie-sessions/		100%	12/12	100%	4/4	100%	1/1	100%	12/12
express/examples/cookies/		95.83%	23/24	87.5%	7/8	100%	3/3	100%	22/22
express/examples/downloads/		94.12%	16/17	87.5%	7/8	100%	3/3	100%	15/15
express/examples/ejs/		100%	11/11	100%	2/2	100%	1/1	100%	11/11
express/examples/error-pages/		97.14%	34/35	83.33%	10/12	100%	6/6	100%	34/34
express/examples/error/		90%	18/20	66.67%	4/6	100%	4/4	100%	18/18
express/examples/markdown/		95.24%	20/21	66.67%	4/6	100%	5/5	100%	20/20
express/examples/multi-router/		100%	9/9	100%	2/2	100%	1/1	100%	9/9
express/examples/multi-router/controllers/		100%	14/14	100%	0/0	100%	4/4	100%	14/14
express/examples/mvc/		95.45%	42/44	80%	8/10	100%	4/4	100%	42/42
express/examples/mvc/controllers/main/		100%	2/2	100%	0/0	100%	1/1	100%	2/2
express/examples/mvc/controllers/pet/		94.12%	16/17	50%	1/2	100%	4/4	100%	16/16
express/examples/mvc/controllers/user-pet/		92.86%	13/14	50%	1/2	100%	1/1	100%	13/13
express/examples/mvc/controllers/user/		100%	21/21	100%	4/4	100%	6/6	100%	19/19
express/examples/mvc/lib/		97.96%	48/49	80%	20/25	100%	2/2	100%	46/46

Getting started with code coverage

Find the right tool for your project

You might find several options to create coverage reports depending on the language(s) you use. Some of the popular tools are listed below:

- Java: Atlassian Clover, Cobertura, JaCoCo
- Javascript: istanbul, Blanket.js
- PHP: PHPUnit

- Python: Coverage.py
- Ruby: SimpleCov

Some tools like istanbul will output the results straight into your terminal while others can generate a full HTML report that lets you explore which part of the code are lacking coverage.

What percentage of coverage should you aim for?

There's no silver bullet in code coverage, and a high percentage of coverage could still be problematic if critical parts of the application are not being tested, or if the existing tests are not robust enough to properly capture failures upfront. With that being said it is generally accepted that 80% coverage is a good goal to aim for. Trying to reach a higher coverage might turn out to be costly, while not necessary producing enough benefit.

The first time you run your coverage tool you might find that you have a fairly low percentage of coverage. If you're just getting started with testing it's a normal situation to be in and you shouldn't feel the pressure to reach 80% coverage right away. The reason is that rushing into a coverage goal might push your team to write tests that are hitting every line of the code instead of writing tests that are based on the business requirements of your application.

For instance, in the example above we reached 100% coverage by testing if 100 and 34 were multiples of 10. But what if we called our function with a letter instead of a number? Should we get a true/false result? Or should we get an exception? It is important that you give time to your team to think about testing from a user perspective and not just by looking at lines of code. Code coverage will not tell you if you're missing things in your source.

Focus on unit testing first

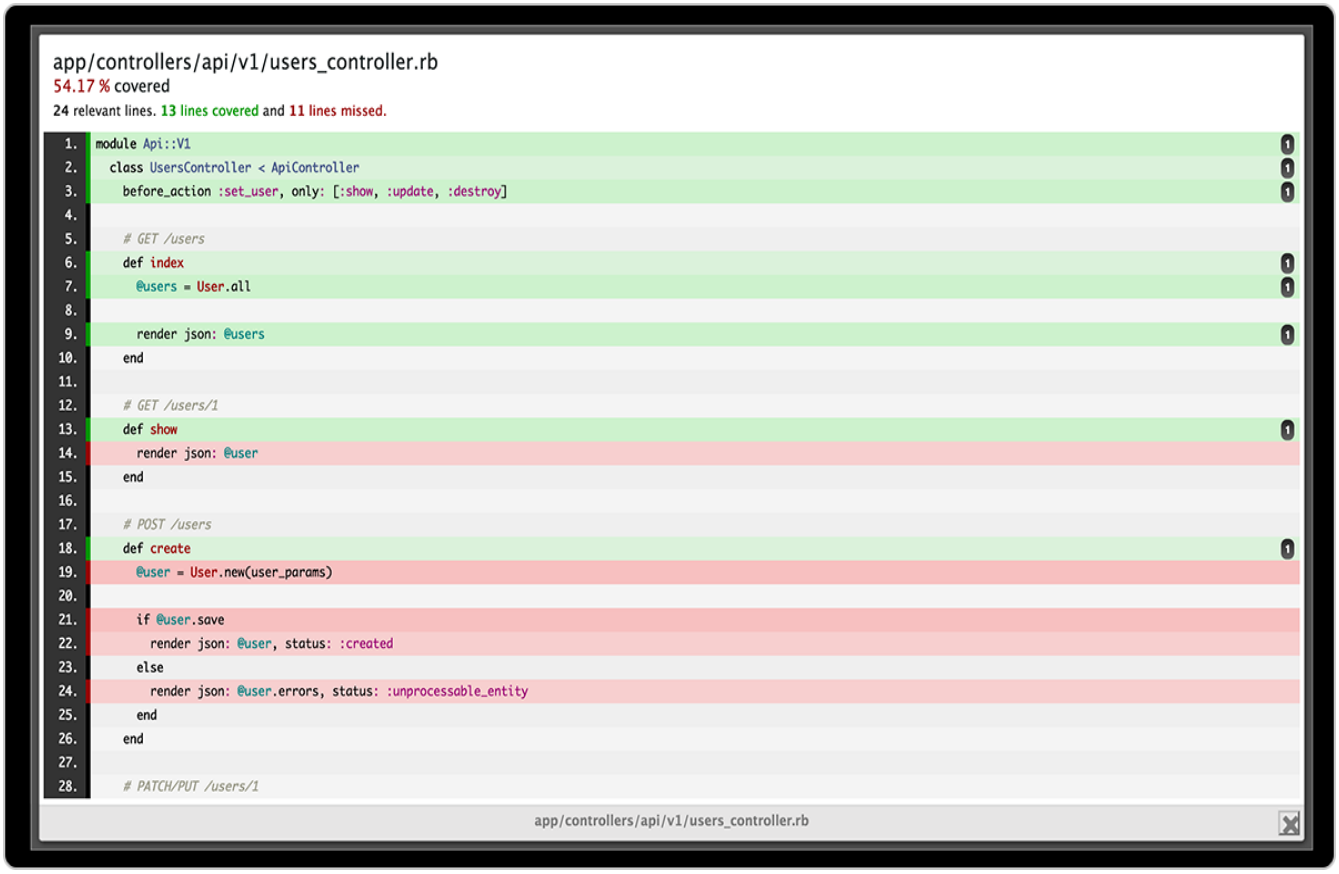
Unit tests consist in making sure that the individual methods of the classes and components used by your application are working. They're generally cheap to implement and fast to run and give you an overall assurance that the basis of the platform is solid. A simple way to increase quickly your code coverage is to start by adding unit tests as, by definition, they should help you make sure that your test suite is reaching all lines of code.

Use coverage reports to identify critical misses in testing

Soon you'll have so many tests in your code that it will be impossible for you to know what part of the application is checked during the execution of your test suite. You'll

know what breaks when you get a red build, but it'll be hard for you to understand what components have passed the tests.

This is where the coverage reports can provide actionable guidance for your team. Most tools will allow you to dig into the coverage reports to see the actual items that weren't covered by tests and then use that to identify critical parts of your application that still need to be tested.



Make code coverage part of your continuous integration flow when you're ready

When you've established your continuous integration (CI) workflow you can start failing the tests if you don't reach a high enough percentage of coverage. Of course, as we said it earlier, it would be unreasonable to set the failure threshold too high, and 90% coverage is likely to cause your build to fail a lot. If your goal is 80% coverage, you might consider setting a failure threshold at 70% as a safety net for your CI culture.

Once again, be careful to avoid sending the wrong message as pressuring your team to reach good coverage might lead to bad testing practices.

Good coverage does not equal good tests

Getting a great testing culture starts by getting your team to understand how the application is supposed to behave when someone uses it properly, but also when someone tries to break it. Code coverage tools can help you understand where you should focus your attention next, but they won't tell you if your existing tests are robust enough for unexpected behaviors.

Achieving great coverage is an excellent goal, but it should be paired with having a robust test suite that can ensure that individual classes are not broken as well as verify the integrity of the system.

Experiment-21

Problem Statement: Define an appropriate metrics for at least 3 quality attributes for any software application of your interest.

Procedure:

Quality Attributes

Quality may be defined from different perspectives. Now let's see how one can measure the Quality Attributes of a product or application.

The following factors are used to measure Software Development Quality. Each attribute can be used to measure product performance. These attributes can be used for Quality assurance as well as Quality control.

Quality Assurance activities are oriented towards the prevention of the introduction of defects and **Quality Control activities** are aimed at detecting defects in products and services.

1) Reliability

Measure if the product is reliable enough to sustain in any condition. Should give the correct results consistently. Product reliability is measured in terms of working of the project under different working environments and different conditions.

2) Maintainability

Different versions of the product should be easy to maintain. For development, it should be easy to add code to the existing system, should be easy to upgrade for new features and new technologies from time to time.

Maintenance should be cost-effective and easy. The system is easy to maintain and correct defects or make a change in the software.

3) Usability

This can be measured in terms of ease of use. The application should be user-friendly. It should be easy to learn. Navigation should be simple.

The system must be:

- Easy to use for input preparation, operation, and interpretation of the output.
- Provide consistent user interface standards and conventions with our other frequently used systems.
- Easy for new or infrequent users to learn to use the system.

4) Portability

This can be measured in terms of Costing issues related to porting, Technical issues related to porting, and Behavioral issues related to porting.

5) Correctness

The application should be correct in terms of its functionality, calculations used internally and the navigation should be correct. This means that the application should adhere to functional requirements.

6) Efficiency

It is one of the major system quality attributes. It is measured in terms of time required to complete any task given to the system. **For example,** the system should utilize processor capacity, disk space, and memory efficiently.
If the system is using all the available resources then the user will get degraded performance failing the system for efficiency. If the system is not efficient, then it cannot be used in real-time applications.

Recommended Reading =>> What is Efficiency Testing?

7) Integrity or Security

Integrity comes with security. System integrity or security should be sufficient to prevent unauthorized access to system functions, prevent information loss, ensure that the software is protected from virus infection, and protect the privacy of data entered into the system.

8) Testability

The system should be easy to test and find defects. If required, it should be easy to divide into different modules for testing.

9) Flexibility

Should be flexible enough to modify. Adaptable to other products with which it needs interaction. Should be easy to interface with other standard 3rd party components.

10) Reusability

Software reuse is a good cost-efficient and time-saving development method. Different

code library classes should be generic enough to be easily used in different application modules. Divide the application into different modules so that modules can be reused across the application.

Recommended reading =>> Cost of Quality and Cost of Poor Quality

11) Interoperability

Interoperability of one system to another should be easy for the product to exchange data or services with other systems. Different system modules should work on different operating system platforms, different databases, and protocol conditions.

Conclusion

By applying the above quality attributes standards we can determine whether the system meets the requirements of quality or not.

As specified above all these attributes are applied to QA and QC process so that both the tester as well as the customer can find the quality of the application or system.

Experiment-22

Problem Statement: Define a complete call graph for any C/C++ code. (Note: The student may use any tool that generate call graph for source code)

Procedure:

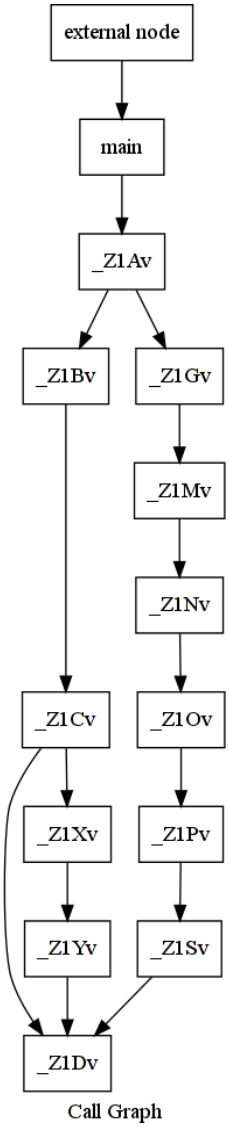
```
path1: A -> B -> C -> D
path2: A -> B -> X -> Y -> D
path3: A -> G -> M -> N -> O -> P -> S -> D
...
path n: ...
staticvoidD(){ }
staticvoidY(){ D(); }
staticvoidX(){ Y(); }
staticvoidC(){ D(); X(); }
staticvoidB(){ C(); }
staticvoidS(){ D(); }
staticvoidP(){ S(); }
staticvoidO(){ P(); }
staticvoidN(){ O(); }
staticvoidM(){ N(); }
staticvoidG(){ M(); }
staticvoidA(){ B(); G(); }
```

```
intmain(){
A();
}
```

Then

```
$ clang++ -S -emit-llvm main1.cpp -o - | opt -analyze -dot-callgraph
$ dot -Tpng -ocallgraph.png callgraph.dot
```

Yields some shiny picture (there is an "external node", because main has external linkage and might be called from outside that translation unit too):



You may want to postprocess this with c++filt, so that you can get the unmangled names of the functions and classes involved. Like in the following

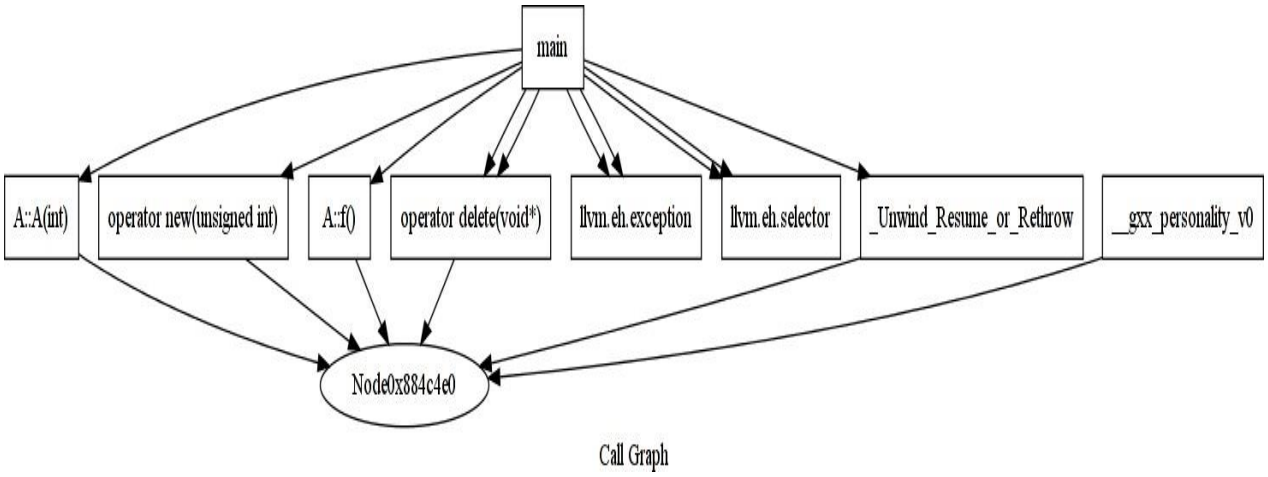
```
#include<vector>
```

```
structA {
A(int);
voidf(); // not defined, prevents inlining it!
};

intmain(){
std::vector<A> v;
v.push_back(42);
v[0].f();
}

$ clang++ -S -emit-llvm main1.cpp -o - |
opt -analyze -std-link-opts -dot-callgraph
$ cat callgraph.dot |
c++filt |
sed's,>,\\>,g; s,-\\>,->,g; s,<,\\<,g' |
gawk'/external node/{id=$1} $1 != id' |
dot -Tpng -ocallgraph.png
```

Yields this beauty (oh my, the size without optimizations turned on was too big!)



Call Graph

That mystical unnamed function, Node0x884c4e0, is a placeholder assumed to be called by any function whose definition is not known.