UNIT-03

Coding Standard & Guideline

UNIT-03/LECTURE-01

Programming Standards and Procedures: [RGPV/June 2014(7)]

- Standards for you
  - methods of code documentation
- Standards for others
  - Integrators, maintainers, testers
  - Prologue documentation
  - Automated tools to identify dependencies
- Matching design with implementation
  - Low coupling, high cohesion, well-defined interfaces
- Allow flexibility to be creative and evolve product’s details in stages
- Flexibility does not preclude standards

Programming Guidelines

Control Structures

- Make the code easy to read
- Build the program from modular blocks
- Make the code not too specific, and not too general
- Use parameter names and comments to exhibit coupling among components
- Make the dependency among components visible

Example of Control Structures

- Control skips around among the program’s statements

```
benefit = minimum;
if (age < 75) goto A;
benefit = maximum;
goto C;
if (AGE < 65) goto B;
if (AGE < 55) goto C;
```

```
A:      if (AGE < 65) goto B;
                benefit = benefit * 1.5 + bonus;
                goto C;
B:              if (age < 55) goto C;
                benefit = benefit * 1.5;
C:      next statement
```

- Rearrange the code

```
if (age < 55) benefit = minimum;
elseif (AGE < 65) benefit = minimum + bonus;
elseif (AGE < 75) benefit = minimum * 1.5 + bonus;
else benefit = maximum;
```

Programming Guidelines

Algorithms

- Common objective and concern: performance (speed)
- Efficiency may have hidden costs
  - cost to write the code faster
  - cost to test the code
  - cost to understand the code
  - cost to modify the code

Programming Guidelines

Data Structures

- Several techniques that used the structure of data to organize the program
  - keeping the program simple
  - using a data structure to determine a program structure

Programming Guidelines

Keep the Program Simple

Example: Determining Federal Income Tax

1. For the first $10,000 of income, the tax is 10%
2. For the next $10,000 of income above $10,000, the tax is 12 percent
3. For the next $10,000 of income above $20,000, the tax is 15 percent

4. For the next $10,000 of income above $30,000, the tax is 18 percent
5. For any income above $40,000, the tax is 20 percent

```
tax = 0.
if (taxable_income == 0) goto EXIT;
if (taxable_income > 10000) tax = tax + 1000;
else{
        tax = tax + .10*taxable_income;
        goto EXIT;
}
if (taxable_income > 20000) tax = tax + 1200;
else{
        tax = tax + .12*(taxable_income-10000):
        goto EXIT;
}
if (taxable_income > 30000) tax = tax + 1500;
else{
        tax = tax + .15*(taxable_income-20000);
        goto EXIT;
}
if (taxable_income < 40000){
        tax = tax + .18*(taxable_income-30000);
        goto EXIT;
}
else
        tax = tax + 1800. + .20*(taxable_income-40000);
```

- Define a tax table for each “bracket” of tax liability

Bracket	Base	Percent
0	0	10
10,000	1000	12
20,000	2200	15
30,000	3700	18
40,000	55000	20

-

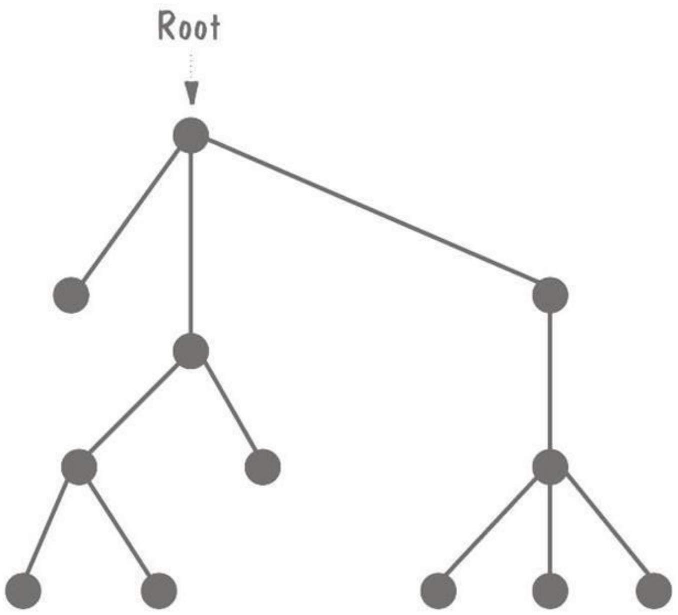
- Simplified algorithm

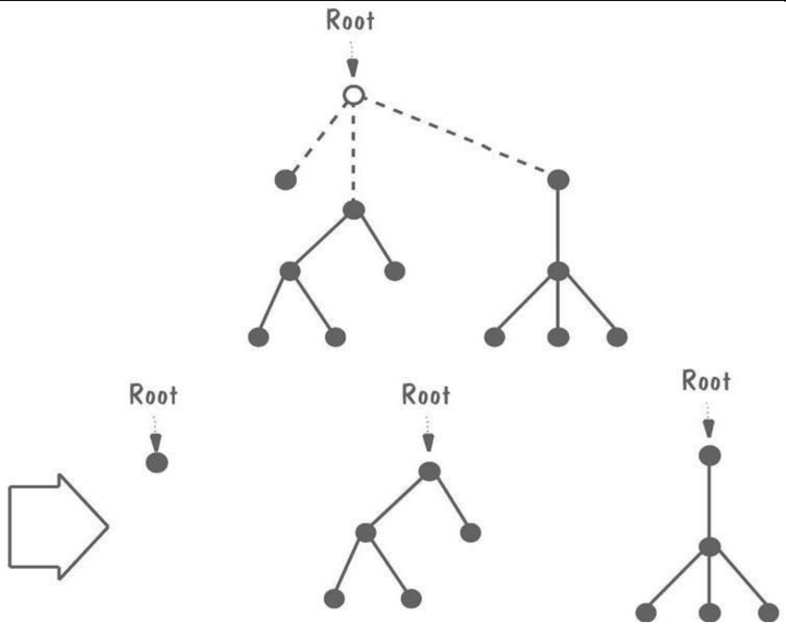
```
        for (int i=2; level=1; i <= 5; i++)
                if (taxable_icome > bracket[i])
                        level = level + 1;
  tax= base[level]+percent[level] * (taxable_income - bracket[level]);
```

Programming Guidelines

Data Structures Example: Rooted Tree

- Recursive data structure
- Graph composed of nodes and lines
  - Exactly one node as root
  - If the lines emanating from the root are erased, the resulting graph is a rooted tree




Root
Root
Root
Root

UNIT-03/LECTURE-02

Programming Guidelines

General Guidelines to Preserve Quality: [RGPV/Jun 2012,2011(10)]

- Localize input and output
- Employ pseudocode
- Revise and rewrite, rather than patch
- Reuse
  - Producer reuse: create components designed to be reused in future applications
  - Consumer reuse: reuse components initially developed for other projects

Programming Guidelines

Consumer Reuse

- Four key characteristics to check about components to reuse
  - does the component perform the function or provide the data needed?
  - is it less modification than building the component from scratch?
  - is the component well-documented?
  - is there a complete record of the component’s test and revision history?

Programming Guidelines

Producer Reuse

- **Several issues to keep in mind**
  - make the components general
  - separate dependencies (to isolate sections likely to change)
  - keep the component interface general and well-defined
  - include information about any faults found and fixed
  - use clear naming conventions
  - document data structures and algorithms
  - keep the communication and error-handling sections separate and easy to modify
- **Reuse Council**
  - Created inventory of components
  - Formed matrix with the features of all past and planned projects
  - Met every week to make component selections, inspect design documentation,

and monitor levels of reuse

Documentation

- Internal documentation
    - header comment block
    - meaningful variable names and statement labels
    - other program comments
    - format to enhance understanding
    - document data (data dictionary)
- External documentation
    - describe the problem
    - describe the algorithm
- What is the component called
- Who wrote the component
- Where the component fits in the general system design
- When the component was written and revised
- Why the component exists
- How the component uses its data structures, algorithms, and control

The Programming Process

Programming as Problem-Solving

- Polya's (1957) four distinct stages of finding a good solution
    - understanding the problem
    - devising plan
    - carrying out the plan
    - looking back
- Two types of participants
    - customers: who define the features using stories, describe detailed tests and assign priorities
    - programmers: who implement the stories

UNIT-03/LECTURE-03

Software Requirement Specification: [RGPV/June2014,2012(7,10)]

There are many good definitions of System and Software Requirements Specifications that will provide us a good basis upon which we can both define a great specification and help us identify deficiencies in our past efforts. There is also a lot of great stuff on the web about writing good specifications. The problem is not lack of knowledge about how to create a correctly formatted specification or even what should go into the specification. The problem is that we don't follow the definitions out there.

We have to keep in mind that the goal is not to create great specifications but to create great products and great software. Can you create a great product without a great specification? Absolutely! You can also make your first million through the lottery – but why take your chances? Systems and software these days are so complex that to embark on the design before knowing what you are going to build is foolish and risky.

The IEEE (www.ieee.org) is an excellent source for definitions of System and Software Specifications. As designers of real-time, embedded system software, we use IEEE STD 830-1998 as the basis for all of our Software Specifications unless specifically requested by our clients. Essential to having a great Software Specification is having a great System Specification. The equivalent IEEE standard for that is IEEE STD 1233-1998. However, for most purposes in smaller systems, the same templates can be used for both.

What are the benefits of a Great SRS?

The IEEE 830 standard defines the benefits of a good SRS:

Establish the basis for agreement between the customers and the suppliers on what the software product is to do. The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs. [NOTE: We use it as the basis of our contract with our clients all the time].

Reduce the development effort. The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

Provide a basis for estimating costs and schedules. The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates. [NOTE: Again, we use the SRS as the basis for our fixed price estimates]

Provide a baseline for validation and verification. Organizations can develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured. [NOTE: We use the SRS to create the Test Plan].

Facilitate transfer.The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

Serve as a basis for enhancement. Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation. [NOTE: This is often a major pitfall – when the SRS is not continually updated with changes]

What should the SRS address?

Again from the IEEE standard:

The basic issues that the SRS writer(s) shall address are the following:

a) Functionality. What is the software supposed to do?
b) External interfaces. How does the software interact with people, the system's hardware, other hardware, and other software?
c) Performance. What is the speed, availability, response time, recovery time of various software functions, etc.?
d) Attributes. What is the portability, correctness, maintainability, security, etc. considerations?
e) Design constraints imposed on an implementation. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

What are the characteristics of a great SRS?

Again from the IEEE standard:

An SRS should be

a) Correct
b) Unambiguous
c) Complete
d) Consistent
e) Ranked for importance and/or stability
f) Verifiable
g) Modifiable
h) Traceable

Correct - This is like motherhood and apple pie. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the specification up to date when you find things that are not correct.

Unambiguous - An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.

Complete - A simple judge of this is that is should be all that is needed by the software designers to create the software.

Consistent - The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.

Ranked for Importance - Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful provide this information in the SRS.

Verifiable - Don't put in requirements like - "It should provide the user a fast response." Another of my favorites is - "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."

Modifiable - Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.

Traceable - Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement?

What is the difference between a System Specification and a Software Specification?

Very often we find that companies do not understand the difference between a System specification and a Software Specification. Important issues are not defined up front and Mechanical, Electronic and Software designers do not really know what their requirements are.

The following is a high level list of requirements that should be addressed in a System Specification:

- Define the functions of the system
- Define the Hardware / Software Functional Partitioning
- Define the Performance Specification
- Define the Hardware / Software Performance Partitioning
- Define Safety Requirements
- Define the User Interface (A good user’s manual is often an overlooked part of the

System specification. Many of our customers haven't even considered that this is the right time to write the user's manual.)

- Provide Installation Drawings/Instructions.
- Provide Interface Control Drawings (ICD's, External I/O)

One job of the System specification is to define the full functionality of the system. In many systems we work on, some functionality is performed in hardware and some in software. It is the job of the System specification to define the full functionality and like the performance requirements, to set in motion the trade-offs and preliminary design studies to allocate these functions to the different disciplines (mechanical, electrical, software).

Another function of the System specification is to specify performance. For example, if the System is required to move a mechanism to a particular position accurate to a repeatability of ± 1 millimeter that is a System's requirement. Some portion of that repeatability specification will belong to the mechanical hardware, some to the servo amplifier and electronics and some to the software. It is the job of the System specification to provide that requirement and to set in motion the partitioning between mechanical hardware, electronics, and software. Very often the System specification will leave this partitioning until later when you learn more about the system and certain factors are traded off (For example, if we do this in software we would need to run the processor clock at 40 mHz. However, if we did this function in hardware, we could run the processor clock at 12 mHz). [This implies that a certain level of research or even prototyping and benchmarking needs to be done to create a System spec. I think it is useful to say that explicitly.]

However, for all practical purposes, most of the systems we are involved with in small to medium size companies, combine the software and the systems documents. This is done primarily because most of the complexity is in the software. When the hardware is used to meet a functional requirement, it often is something that the software wants to be well documented. Very often, the software is called upon to meet the system requirement with the hardware you have. Very often, there is not a systems department to drive the project and the software engineers become the systems engineers. For small projects, this is workable even if not ideal. In this case, the specification should make clear which requirements are software, which are hardware, and which are mechanical.

What is the difference between a design requirement and software requirement? [RGPV/June 2011(10)

In short, the SRS should not include any design requirements. However, this is a difficult discipline. For example, because of the partitioning and the particular RTOS you are using, and the particular hardware you are using, you may require that no task use more than 1 ms of processing prior to releasing control back to the RTOS. Although that may be a true requirement and it involves software and should be tested – it is truly a design requirement and should be included in the Software Design Document or in the Source code.

Consider the target audience for each specification to identify what goes into what documents.

Marketing/Product Management

Creates a product specification and gives it to Systems. It should define everything Systems needs to specify the product

Systems

Creates a System Specification and gives it to Systems/Software and Mechanical and Electrical Design.

Systems/Software

Creates a Software Specification and gives it to Software. It should define everything Software needs to develop the software.

Thus, the SRS should define everything explicitly or (preferably) by reference that software needs to develop the software. References should include the version number of the target document. Also, consider using master document tools which allow you to include other documents and easily access the full requirements.

Is this do-able? Won’t we miss our deadlines if we take the time to do this?

This is a great question. There is no question that there is balance in this process. We have seen companies and individuals go overboard on documenting software that doesn’t need to be documented, such as a temporary utility. We have also seen customers kill good products by spending too much time specifying it.

However, the bigger problem is at the other end of the spectrum. We have found that taking the time up front pays dividends down stream. If you don’t have time to specify it up front, you probably don’t have the time to do the project.

Here are some of our guidelines:

- Spend time specifying and documenting well software that you plan to keep.

- Keep documentation to a minimum when the software will only be used for a short time or has a limited number of users.

- Have separate individuals write the specifications (not the individual who will write the code).

- The person to write the specification should have good communication skills.

- Pretty diagrams can help but often tables and charts are easier to maintain and can communicate the same requirements.

- Take your time with complicated requirements. Vagueness in those areas will come back to bite you later.

- Conversely, watch out for over-documenting those functions that are well understood by many people but for which you can create some great requirements.

- Keep the SRS up to date as you make changes.

- Approximately 20-25% of the project time should be allocated to requirements definition.

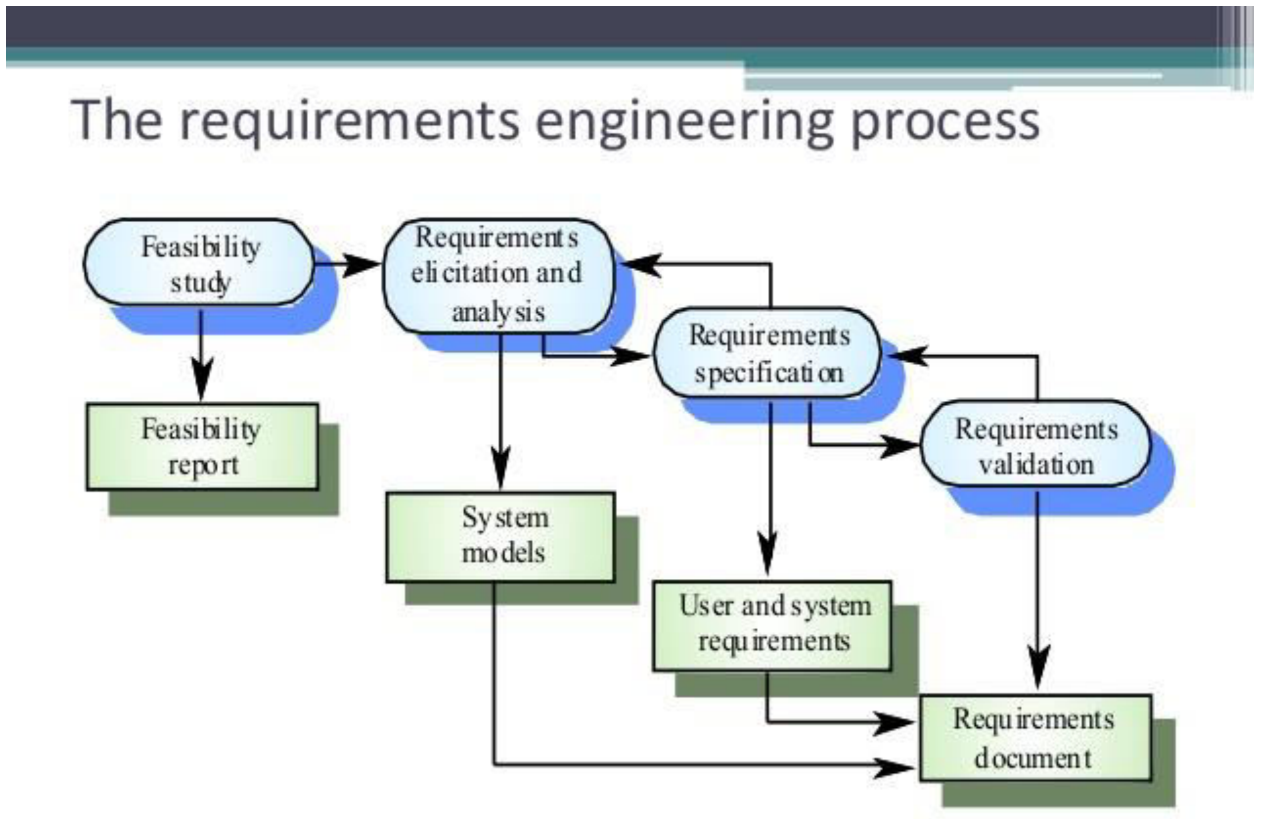
- Keep 5% of the project time for updating the requirements after the design has begun.

- Test the requirements document by using it as the basis for writing the test plan.

Feasibility Study: [RGPV/June 2012(10)]

Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practical and workable in the software. Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study. The objective of the feasibility study is to establish the reasons for developing the software that is acceptable to users, adaptable to change and conformable to established standards. Various other objectives of feasibility study are listed below.

- To analyze whether the software will meet organizational requirements
- To determine whether the software can be implemented using the current technology and within the specified budget and schedule
- To determine whether the software can be integrated with other existing software.


The requirements engineering process
Feasibility study
Requirements elicitation and analysis
Requirements specification
Requirements validation
Feasibility report
System models
User and system requirements
Requirements document

Feasibility Study Process

Feasibility study comprises the following steps.

Information assessment: Identifies information about whether the system helps in achieving the objectives of the organization. It also verifies that the system can be implemented using new technology and within the budget and whether the system can be integrated with the existing system.
Information collection: Specifies the sources from where information about software can be obtained. Generally, these sources include users (who will operate the software), organization (where the software will be used), and the software development team (which understands user requirements and knows how to fulfil them in software).
Report writing: Uses a feasibility report, which is the conclusion of the feasibility study by the

software development team. It includes the recommendations whether the software development should continue. This report may also include information about changes in the software scope, budget, and schedule and suggestions of any requirements in the system.

General information: Describes the purpose and scope of feasibility study. It also describes system overview, project references, acronyms and abbreviations, and points of contact to be used. **System overview** provides description about the name of the organization responsible for the software development, system name or title, system category, operational status, and so on. **Project references** provide a list of the references used to prepare this document such as documents relating to the project or previously developed documents that are related to the project. **Acronyms and abbreviations** provide a list of the terms that are used in this document along with their meanings. **Points of contact** provide a list of points of organizational contact with users for information and coordination. For example, users require assistance to solve problems (such as troubleshooting) and collect information such as contact number, e-mail address, and so on.

Management summary: Provides the following information.

Environment: Identifies the individuals responsible for software development. It provides information about input and output requirements, processing requirements of the software and the interaction of the software with other software. It also identifies system security requirements and the system's processing requirements

Current functional procedures: Describes the current functional procedures of the existing system, whether automated or manual. It also includes the data-flow of the current system and the number of team members required to operate and maintain the software.

Functional objective: Provides information about functions of the system such as new services, increased capacity, and so on.

Performance objective: Provides information about performance objectives such as reduced staff and equipment costs, increased processing speeds of software, and improved controls.

. **Assumptions and constraints:** Provides information about assumptions and constraints such as operational life of the proposed software, financial constraints, changing hardware, software and operating environment, and availability of information and sources.

. **Methodology:** Describes the methods that are applied to evaluate the proposed software in order to reach a feasible alternative. These methods include survey, modelling, benchmarking, etc.

. **Evaluation criteria:** Identifies criteria such as cost, priority, development time, and ease of system use, which are applicable for the development process to determine the most suitable system option.

. **Recommendation:** Describes a recommendation for the proposed system. This includes the delays and acceptable risks.

. **Proposed software:** Describes the overall concept of the system as well as the procedure to be used to meet user requirements. **In** addition, it provides information about improvements, time and resource costs, and impacts. Improvements are performed to enhance the functionality and performance of the existing software. Time and resource costs include the costs associated with software development from its requirements to its maintenance and staff training. Impacts describe the possibility of future happenings and include various types of impacts as listed below.

. **Equipment impacts:** Determine new equipment requirements and changes to be made in the currently available equipment requirements.

. **Software impacts:** Specify any additions or modifications required in the existing software and supporting software to adapt to the proposed software.

. **Organizational impacts:** Describe any changes in organization, staff and skills requirement.

. **Operational impacts:** Describe effects on operations such as user-operating procedures, data processing, data entry procedures, and so on.

. **Developmental impacts:** Specify developmental impacts such as resources required to develop databases, resources required to develop and test the software, and specific activities to be performed by users during software development.

. **Security impacts:** Describe security factors that may influence the development, design, and continued operation of the proposed software.

. **Alternative systems:** Provide description of alternative systems, which are considered in a feasibility study. This also describes the reasons for choosing a particular alternative system to develop the proposed software and the reason for rejecting alternative systems.

Types of Feasibility Study

- **Technical Feasibility** - Does the company have the technological resources to undertake the project? Are the processes and procedures conducive to project success?

- **Schedule Feasibility** - Does the company currently have the time resources to undertake the project? Can the project be completed in the available time?

- **Economic Feasibility** - Given the financial resources of the company, is the project something that can be completed? The economic feasibility study is more commonly called the cost/benefit analysis.

- **Cultural Feasibility** - What will be the impact on both local and general cultures? What sort of environmental implications does the feasibility study have?

- **Legal/Ethical Feasibility** - What are the legal implications of the project? What sort of ethical considerations are there? You need to make sure that any project undertaken will meet all legal and ethical requirements before the project is on the table.

- **Resource Feasibility** - Do you have enough resources, what resources will be required, what facilities will be required for the project, etc.

- **Operational Feasibility** - This measures how well your company will be able to solve problems and take advantage of opportunities that are presented during the course of the project

- **Marketing Feasibility** - Will anyone want the product once its done? What is the target demographic? Should there be a test run? Is there enough buzz that can be created for the product?
- **Real Estate Feasibility** - What kind of land or property will be required to undertake the project? What is the market like? What are the zoning laws? How will the business impact the area?
- **Comprehensive Feasibility** - This takes a look at the various aspects involved in the project - marketing, real estate, cultural, economic, etc. When undertaking a new business venture, this is the most common type of feasibility study performed.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain SRS. What are the good characteristics of SRS?	June 2014,2012	7,10
Q.2	What do you mean by feasibility study? Explain different type of feasibilities.	June 2012	10
Q.3	What is the difference between a design requirement and software requirement?	June 2011	10

UNIT-03/LECTURE-04

Software Design: [RGPV/June 2014(7)]

A software design creates meaningful engineering representation (or model) of some software product that is to be built. Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model. A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, and behaviour) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system. Each design product is reviewed for quality (i.e. identify and correct errors, inconsistencies, or omissions, whether better alternatives exist, and whether the design model can be implemented within the project constraints) before moving to the next phase of software development.

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product
- Design principles establish and overriding philosophy that guides the designer as the work is performed
- Design concepts must be understood before the mechanics of design practice are applied
- Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight)
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve

Software Engineering Design

- Data/Class design - created by transforming the analysis model class-based elements (class diagrams, analysis packages, CRC models, collaboration diagrams) into classes and data structures required to implement the software
- Architectural design - defines the relationships among the major structural elements of the software, it is derived from the class-based elements and flow-oriented elements (data flow diagrams, control flow diagrams, processing narratives) of the analysis model
- Interface design - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements (use-case text, use-case diagrams, activity diagrams, swim lane diagrams), flow-oriented elements, and behavioural elements (state diagrams, sequence diagrams)
- Component-level design - created by transforming the structural elements defined by the software architecture into a procedural description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioural elements

Generic Design Task Set

1. Examine information domain model and design appropriate data structures for data objects and their attributes
2. Select an architectural pattern appropriate to the software based on the analysis model
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture
    - Be certain each subsystem is functionally cohesive

- Design subsystem interfaces
- Allocate analysis class or functions to subsystems

4. Create a set of design classes
   - Translate analysis class into design class
   - Check each class against design criteria and consider inheritance issues
   - Define methods and messages for each design class
   - Evaluate and select design patterns for each design class or subsystem after considering alternatives
   - Revise design classes and revise as needed

5. Design any interface required with external systems or devices

6. Design user interface
   - Review task analyses
   - Specify action sequences based on user scenarios
   - Define interface objects and control mechanisms
   - Review interface design and revise as needed

7. Conduct component level design
   - Specify algorithms at low level of detail
   - Refine interface of each component
   - Define component level data structures
   - Review components and correct all errors uncovered

8. Develop deployment model

Design Concepts

- Abstraction – allows designers to focus on solving a problem without being concerned about irrelevant lower level details (*procedural abstraction* - named sequence of events and data abstraction – named collection of data objects)
- Software Architecture – overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
  - Structural models – architecture as organized collection of components
  - Framework models – attempt to identify repeatable architectural patterns
  - Dynamic models – indicate how program structure changes as a function of external events
  - Process models – focus on the design of the business or technical process that system must accommodate
  - Functional models – used to represent system functional hierarchy
- Design Patterns – description of a design structure that solves a particular design problem within a specific context and its impact when applied
- Separation of concerns – any complex problem is solvable by subdividing it into pieces that can be solved independently
- Modularity - the degree to which software can be understood by examining its components independently of one another
- Information Hiding – information (data and procedure) contained within a module is inaccessible to modules that have no need for such information

- Functional Independence – achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models
  - Cohesion - qualitative indication of the degree to which a module focuses on just one thing
  - Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world
- Refinement – process of elaboration where the designer provides successively more detail for each design component
- Aspects – a representation of a cross-cutting concern that must be accommodated as refinement and modularization occur
- Refactoring – process of changing a software system in such a way internal structure is improved without altering the external behaviour or code design

Design Classes

- Refine analysis classes by providing detail needed to implement the classes and implement a software infrastructure the support the business solution
- Five types of design classes can be developed to support the design architecture
  - user interface classes – abstractions needed for human-computer interaction (HCI)
  - business domain classes – refinements of earlier analysis classes
  - process classes – implement lower level business abstractions
  - persistent classes – data stores that persist beyond software execution
  - System classes – implement software management and control functions

Design Class Characteristics

- Complete (includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
- Primitiveness – each class method focuses on providing one service
- High cohesion – small, focused, single-minded classes
- Low coupling – class collaboration kept to minimum

Design Model

- Process dimension – indicates design model evolution as design tasks are executed during software process
  - Architecture elements
  - Interface elements
  - Component-level elements
  - Deployment-level elements
- Abstraction dimension – represents level of detail as each analysis model element is transformed into a design equivalent and refined
  - High level (analysis model elements)
  - Low level (design model elements)
- Many UML diagrams used in the design model are refinements of diagrams created in the analysis model (more implementation specific detail is provided)

- Design patterns may be applied at any point in the design process

Data Design

- High level model depicting user's view of the data or information
- Design of data structures and operators is essential to creation of high-quality applications
- Translation of data model into database is critical to achieving system business objectives
- Reorganizing databases into a data warehouse enables data mining or knowledge discovery that can impact success of business itself

Architectural Design

- Provides an overall view of the software product
- Derived from
  - Information about the application domain relevant to software
  - Relationships and collaborations among specific analysis model elements
  - Availability of architectural patterns and styles
- Usually depicted as a set of interconnected systems that are often derived from the analysis packages with in the requirements model

Interface Design

- Interface is a set of operations that describes the externally observable behaviour of a class and provides access to its public operations
- Important elements
  - User interface (UI)
  - External interfaces to other systems
  - Internal interfaces between various design components
- Modelled using UML communication diagrams (called collaboration diagrams in UML 1.x)

Component-Level Design

- Describes the internal detail of each software component
- Defines
  - Data structures for all local data objects
  - Algorithmic detail for all component processing functions
  - Interface that allows access to all component operations
- Modelled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

Deployment-Level Design

- Indicates how software functionality and subsystems will be allocated within the physical computing environment.
- Modelled using UML deployment diagrams.
- Descriptor form deployment diagrams show the computing environment but does not indicate configuration details.
- Instance form deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design.

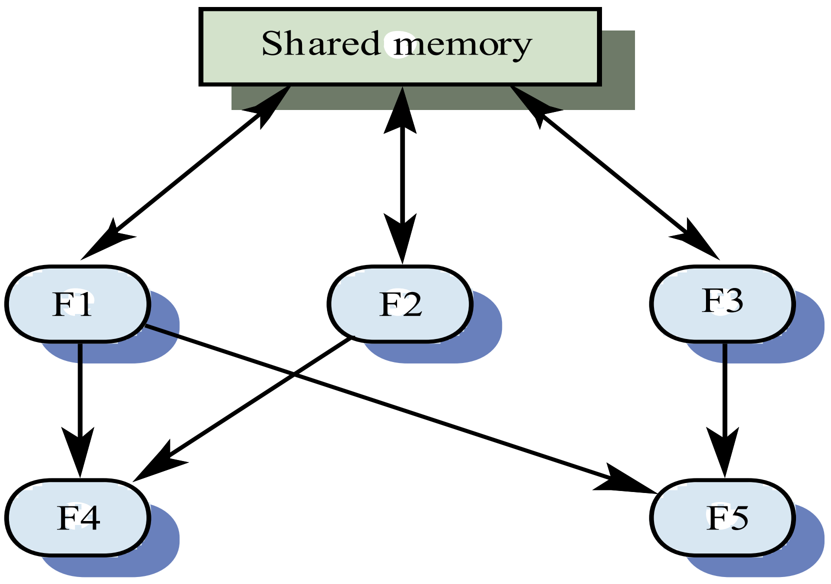
S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Describe the design process in software development. What are the characteristics & criteria for design?	June 2014	7

UNIT-03/LECTURE-05

Function-oriented design[RGPV/June 2013(7)]

- Design with functional units which transform inputs to outputs
- Practiced informally since programming began
- Thousands of systems have been developed using this approach
- Supported directly by most programming languages
- Most design methods are functional in their approach

A function-oriented view of design



Functional design process

- Data-flow design
  - Model the data processing in the system using data-flow diagrams
- Structural decomposition
  - Model how functions are decomposed to sub-functions using graphical structure charts
- Detailed design
  - The entities in the design and their interfaces are described in detail. These may be recorded in a data dictionary and the design expressed using a PDL

Data flow diagrams[RGPV/June 2012(10)]

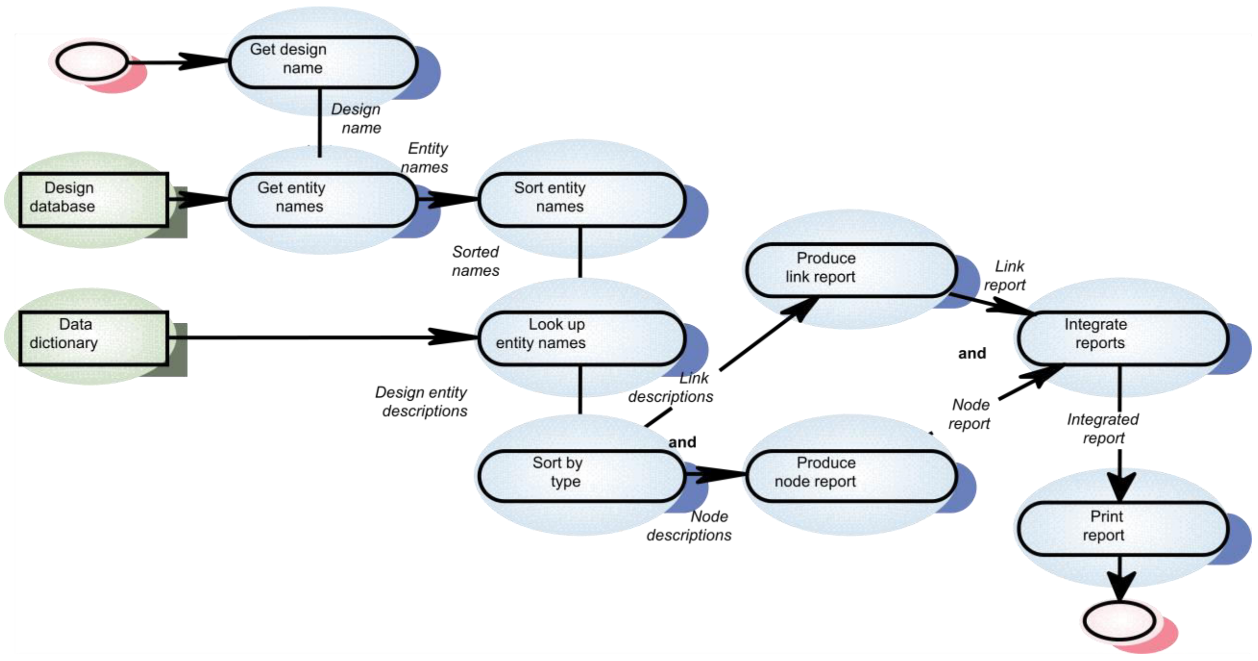
- Show how an input data item is functionally transformed by a system into an output data item.

- Are an integral part of many design methods
- May be translated into either a sequential or parallel design. In a sequential design, processing elements are functions or procedures; in a parallel design, processing elements are tasks or processes

DFD notation

- Rounded rectangle – function or transform
- Rectangle – data store
- Circles – user interactions with the system
- Arrows – show direction of data flow
- keywords and/ or. Used to link data flows

Design report generator



Structural decomposition

- Structural decomposition is concerned with developing a model of the design which shows the dynamic structure i.e. function calls
- This is not necessarily the same as the static composition structure
- The aim of the designer should be to derive design units which are highly cohesive and loosely coupled

- In essence, a data flow diagram is converted to a structure chart

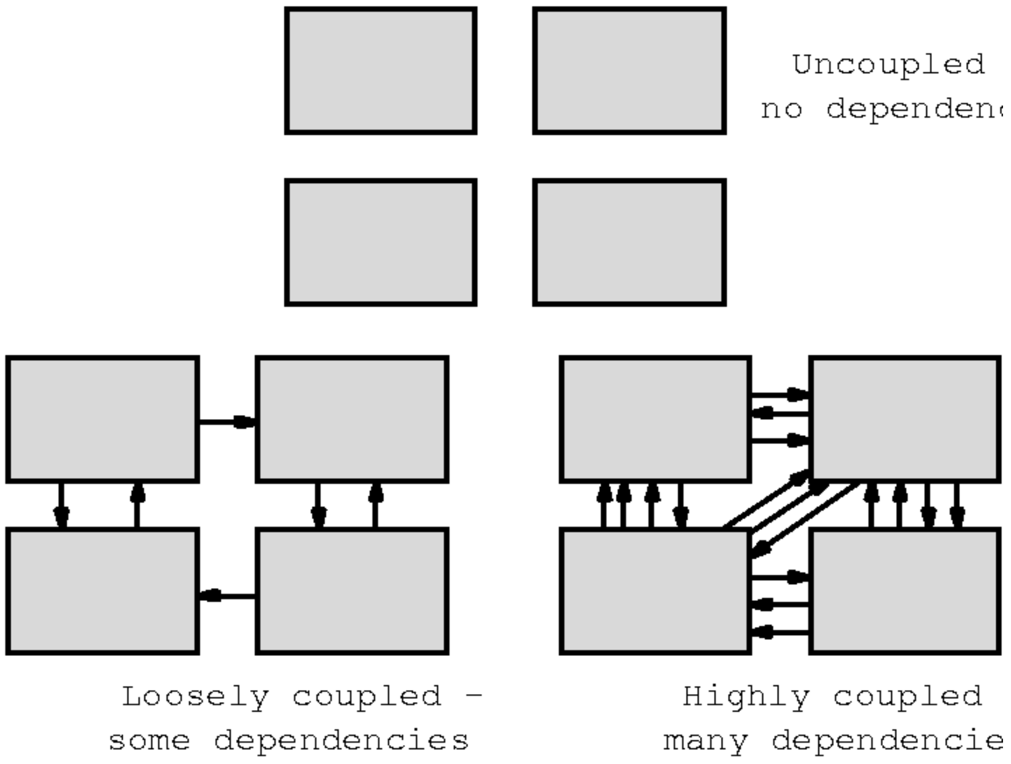
Decomposition guidelines

- For business applications, the top-level structure chart may have three functions namely input, process and output
- Data validation functions should be subordinate to an input function
- Coordination and control should be the responsibility of functions near the top of the hierarchy
- Each node in the structure chart should have between two and seven subordinates
- The aim of the design process is to identify loosely coupled, highly cohesive functions. Each function should therefore do one thing and one thing only

– Cohesion – the degree to which a module performs one and only one function.

– Coupling – the degree to which a module is connected to other modules in the system.

Coupling and cohesion[RGPV/June 2014(7)]



Process steps

- Identify system processing transformations
    - Transformations in the DFD which are concerned with processing rather than input/output activities. Group under a single function in the structure chart
- Identify input transformations
    - Transformations concerned with reading, validating and formatting inputs. Group under the input function
- Identify output transformations
    - Transformations concerned with formatting and writing output. Group under the output function

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain extensions of DFD for real time systems.	June 2011	10
Q.2	Differentiate between data structure design & object oriented design.	June 2013	7
Q.3	Discuss the impact of cohesion, coupling, fan-in, fan-out & factoring June 2013in design phase	June 2014	7

UNIT-03/LECTURE-06

Object Oriented Design: [RGPV/June 2012(10)]

The Object Oriented Design converts the Object Oriented Analysis model into a design model. This serves an outline for software construction. Object Oriented Design supports following object oriented concepts such as Abstraction, Information Hiding, Functional Independence, and Modularity. Design is the initial step in moving towards from the problem domain to the solution domain. A detailed design includes specification of all the classes with its attributes, detailed interface. The purpose of design is to specify a working solution that can be easily translated into a programming language code.

The Object Oriented Design is classified into

- Architectural Design
- Detailed Design

Architectural Design

Architectural design divides the system into different sub systems known as packages. Then the dependency, relationship and communication between the packages are also identified. Package diagram is use to represent architectural design using UML.

Detailed Design

It describes the detailed description of the classes that is all the attributes (Variables and functions). The detailed class diagram represents the detailed design using UML.

- Concerned with producing a short design specification (minispec) of each function. This should describe the processing, inputs and outputs
- These descriptions should be managed in a data dictionary
- From these descriptions, detailed design descriptions, expressed in a PDL or programming language, can be produced

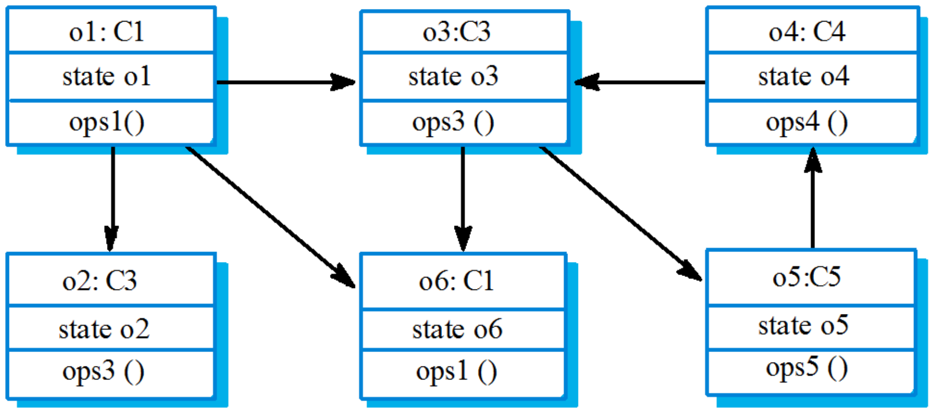
Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated. Objects

communicate by message passing.

- Objects may be distributed and may execute sequentially or in parallel.

Interacting objects



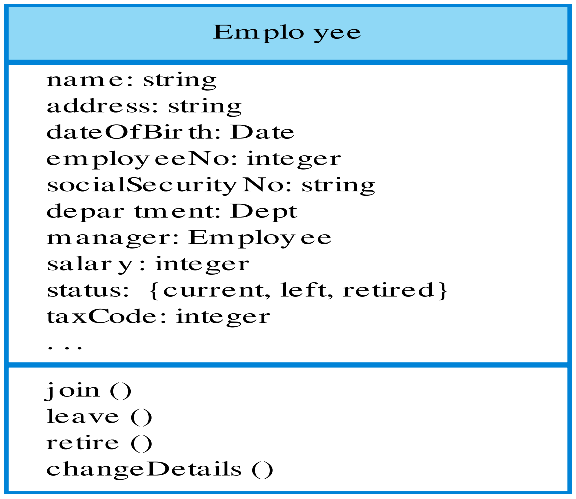
Advantages of OOD

- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

The Unified Modelling Language

- Several different notations for describing object-oriented designs were proposed in the 1980s and 1990s.
- The Unified Modelling Language is an integration of these notations.
- It describes notations for a number of different models that may be produced during OO analysis and design.
- It is now a de facto standard for OO modelling.

Employee object class (UML)



Object communication

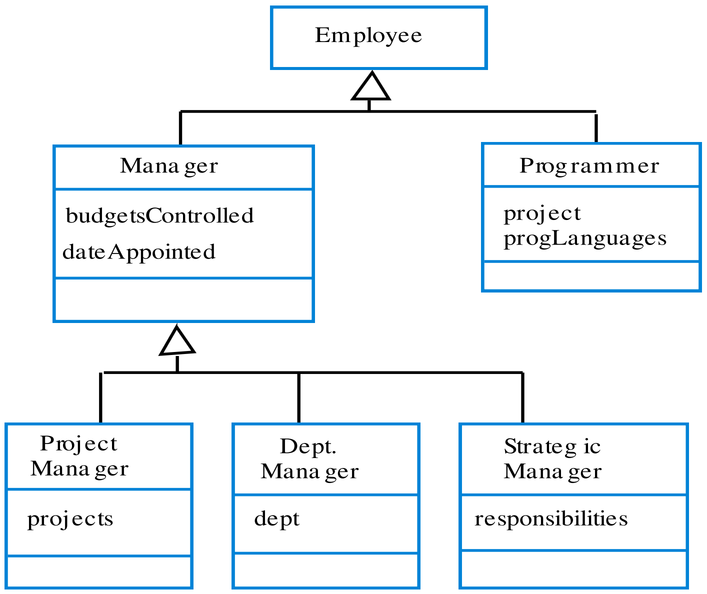
- Conceptually, objects communicate by message passing.
- Messages
    - The name of the service requested by the calling object; Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
    - Name = procedure name;
    - Information = parameter list.

Generalisation and inheritance

- Objects are members of classes that define attribute types and operations.
- Classes may be arranged in a class hierarch where one class (a super-class) is a generalisation of one or more other classes (sub-classes).
- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.

Generalisation in the UML is implemented as inheritance in OO programming languages.

A generalisation hierarchy



UNIT-03/LECTURE-07

Design Pattern Implementation Strategy:

Significant changes are taking place in management and especially project management today. We hear that organizations, like the New York Times, Tribune Co., Ernst & Young switched from the so-called top-down management style to bottom-up management. Others, including some of the world's biggest corporations, such as Toyota and IBM, implemented bottom-up management style elements in some of their departments. The popularity of the bottom-up approach to management is growing. In spite of this fact, the discussions about the two major approaches are still hot. Why have organizations become so anxious about changing their management style? If we compare the two management approaches, the answer to this question will be clear.

Managing projects top-down

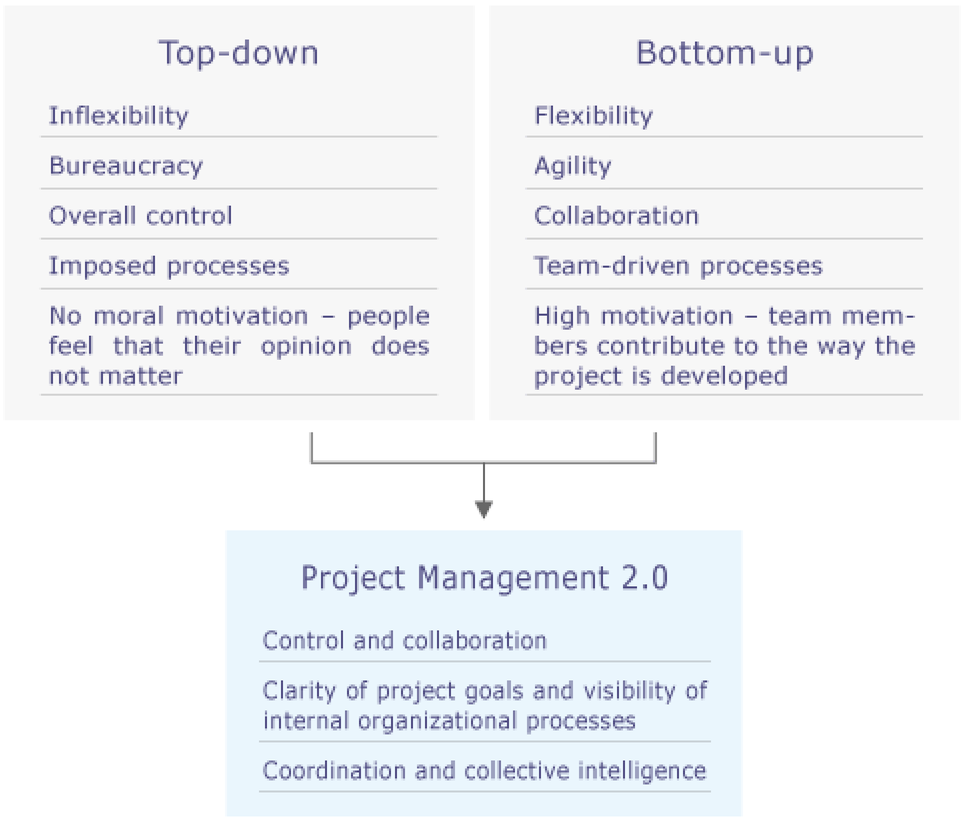
The top-down approach remains extremely popular in contemporary project management. The phrase "top-down" means that all the directions come from the top. Project objectives are established by the top management. Top managers provide guidelines, information, plans and fund processes. All of the project manager's expectations are clearly communicated to each project participant. Following this approach, ambiguity opens the door for potential failure, and the managers should be as specific as possible when communicating their expectations. Process formality is very important for this approach.

Examples of the top-down approach applications can be found in many organizations. One of such example is the New York Times, a leader in the newspaper industry. Several years ago, American Journalism Review (www.ajr.com) reported that The Times' executive management felt that they were far from what was necessary for creation of a vibrant workplace and a successful organization. Power was centralized and masthead editors experienced overall control. Editors introduced the same management pattern in the projects for which they were responsible. One person's emotions and opinions influenced all the project decisions, and this person was the project manager. What was the result? Team members felt that they weren't listened to, that their voices didn't count. There was no effective collaboration between the journalists. They were not morally motivated to do their jobs. The managing executives then realized that they needed to give more freedom to the teams and change their management style. It took quite a while to introduce bottom-up management to the organization. But, obviously, it was worth the time and effort, as New York Times employees say that collaboration became much more efficient, and team members now work together more productively.

Similar problems caused by utilizing the top-down approach can be observed in many organizations with a traditional management style. Experience shows that this top-down management often results in reduced productivity and causes bottlenecks or so-called lockdowns. A lockdown gives the project manager total control over his team. Such lockdowns can lead to unnecessary pain and significantly slow down a project's completion.

Bottom-up project management options

The factors mentioned above may play a vital role in a project's failure, and this is the reason why numerous organizations have turned to a bottom-up management style or at least some of its elements. The New York Times is one of the good examples. The bottom-up approach implies proactive team input in the project executing process. Team members are invited to participate in every step of the management process. The decision on a course of action is taken by the whole team. Bottom-up style allows managers to communicate goals and value, e.g. through milestone planning. Then team members are encouraged to develop personal to-do lists with the steps necessary to reach the milestones on their own. The choice of methods and ways to perform their tasks is up to the team. The advantage of this approach is that it empowers team members to think more creatively. They feel involved into the project development and know that their initiatives are appreciated. The team members' motivation to work and make the project a success is doubled. Individual members of the team get an opportunity to come up with project solutions that are focused more on practical requirements than on abstract notions. The planning process is facilitated by a number of people, which makes it flow significantly faster. The to-do lists of all the team members are collected into the detailed general project plan. Schedules, budgets and results are transparent. Issues are made clear by the project manager to avoid as many surprises as possible. Bottom-up project management can also be viewed as a way of coping with the increasing gap between the information necessary to manage knowledge workers and the ability of managers to acquire and apply this information.


Top-down
Inflexibility
Bureaucracy
Overall control
Imposed processes
No moral motivation – people feel that their opinion does not matter
Bottom-up
Flexibility
Agility
Collaboration
Team-driven processes
High motivation – team members contribute to the way the project is developed
Project Management 2.0
Control and collaboration
Clarity of project goals and visibility of internal organizational processes
Coordination and collective intelligence

Today, leading companies use a combination of integration techniques to implement their end to end business processes. Typically these include a messaging backbone, integration brokers, Enterprise Application Integration, object request brokers and transaction monitors. But it is both difficult and time-consuming to manage all of these middleware infrastructure products and develop application adapters. Middleware companies are well aware of this and their products are improving all the time to simplify process integration.

Piecemeal approaches to process integration do not work. As processes change in the business, integration costs escalate out of control. Point to point solutions creates an unmanageable and complex topology. What starts as a neat top down design deteriorates rapidly. The disconnected activities of application integration and B2B integration are not geared to supporting end to end process design and deployment.

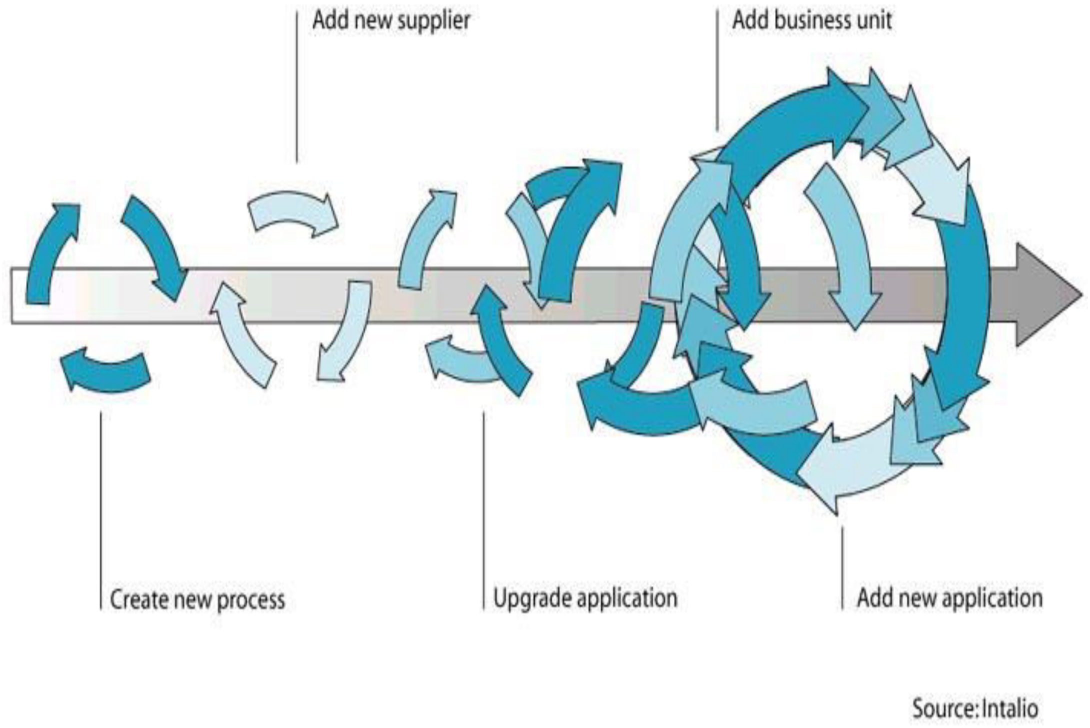


Figure - Without process management, as processes change, integration costs escalate out of control

Process management products unify these disparate middleware activities. Each element acts as a distributed computing service supporting the process management system. Communication between systems is based on the processes that drive their use, not on the connections between or configuration of lower level services. The advantages are clear:

- Process design is independent of technical deployment
- Process models developed in different parts of the enterprise, or by partners, can be composed (or decomposed) and related to each other.
- High level models of the business can be refined by further modelling.
- Abstract models can act as blueprints for subsequent concrete models, thereby ensuring that all business activities fall within the agreed strategies.
- Process improvements achieved in one part of the business can be exported to other parts of the business, with adaptation as required.
- High level business models expressed by the CEO or other senior managers can be codified and used to drive further modelling.
- Deployed processes can be incrementally refined.

The result is that the process model the business wants to deploy is the model that actually is deployed, and that model drives the integration and automation activities across the many systems of the business and its partners. Constraints at every level are respected and enforced by the process management system, and changes are propagated across all the systems that participate in the process.

This top down approach does not imply that everything has to be done at once, nor does it imply there must be a single enterprise process model – clearly impractical. Rather, the term 'top down' implies the ability to model at all levels simultaneously, to combine models yet retain their meaning, and to use process design patterns to constrain the behaviour of sub-processes. This top down approach will often be used in conjunction with bottom up integration and aggregation of web services.

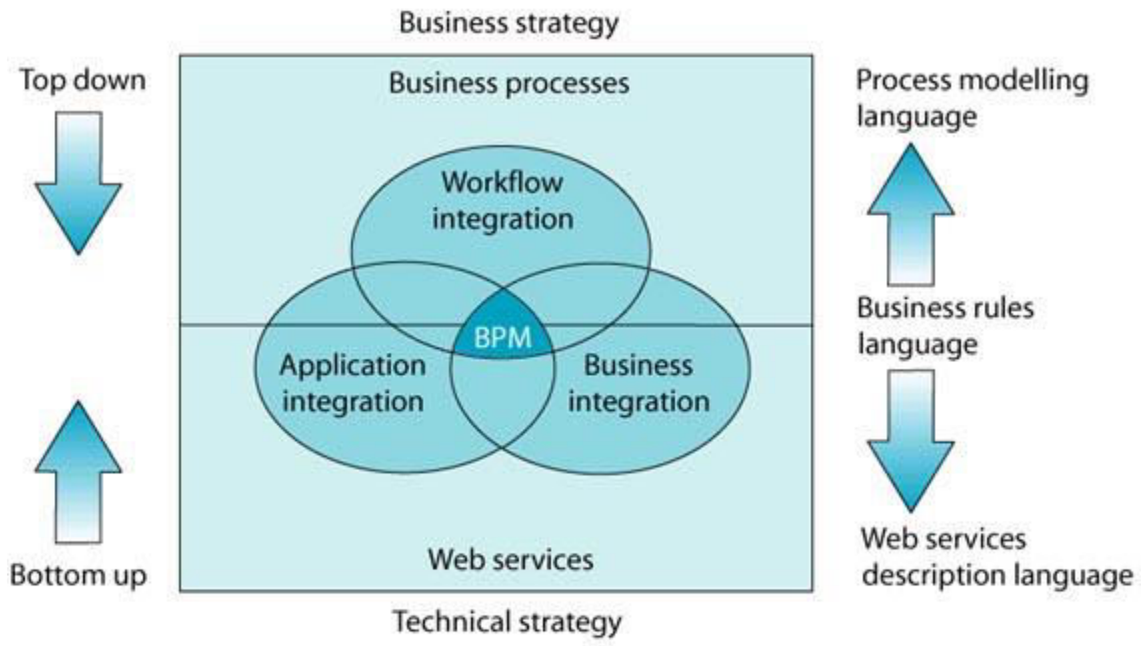


Figure - Top down business process management complements bottom up web services integration

For an initial implementation of process management, we advise companies to focus on a manageable problem, understand how to model it, and then deploy it on a BPMS. As experience grows, business managers will gain confidence that the process management methodology can be extended to larger problems, such as supporting the supply chain.

The top down approach is very different from the traditional software engineering cycle, where business strategies are translated to business requirements, to business objects and finally to software code. Process management is straight through - there is no translation to executable code. The live system can be tuned live. Process improvements can be metered and measured, and investments in IT justified. Process management delivers a continuous and predictable return on investment.

UNIT-03/LECTURE-08

Formal & informal Specification

Formal specification is part of a more general collection of techniques that are known as 'formal methods'.

- These are all based on mathematical representation and analysis of software.
- Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.

- **Acceptance of formal methods**
- Formal methods have not become mainstream software development techniques as was once predicted
  - Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced;
  - Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market;
  - The scope of formal methods is limited. They are not well-suited to specifying and analysing user interfaces and user interaction;
  - Formal methods are still hard to scale up to large systems.

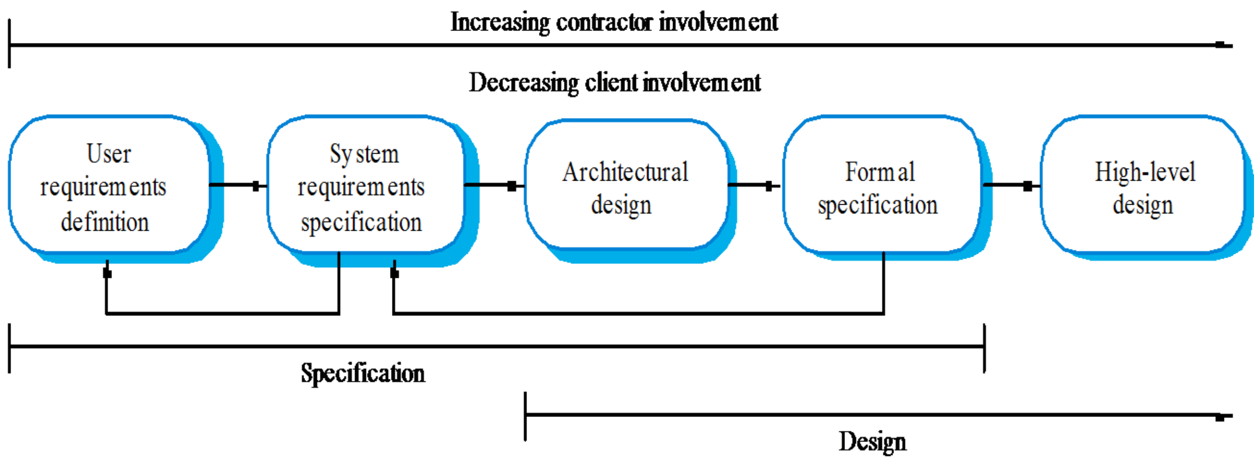
Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

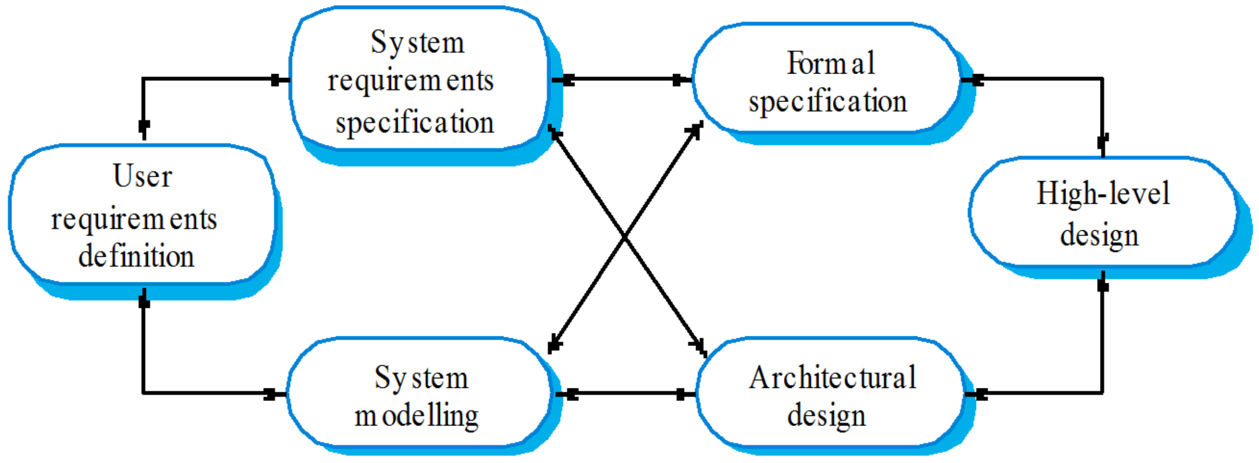
- **Specification in the software process**
- Specification and design are inextricably intermingled.

- Architectural design is essential to structure a specification and the specification process.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and design


Increasing contractor involvement
Decreasing client involvement
User requirements definition
System requirements specification
Architectural design
Formal specification
High-level design
Specification
Design

Specification in the software process



Use of formal specification

- Formal specification involves investing more effort in the early phases of software development.
- This reduces requirements errors as it forces a detailed analysis of the requirements.
- Incompleteness and inconsistencies can be discovered and resolved.
- Hence, savings as made as the amount of rework due to requirements problems is reduced.

UNIT-03/LECTURE-09

Goals of Analysis Modeling: [RGPV/June 2011(10)]

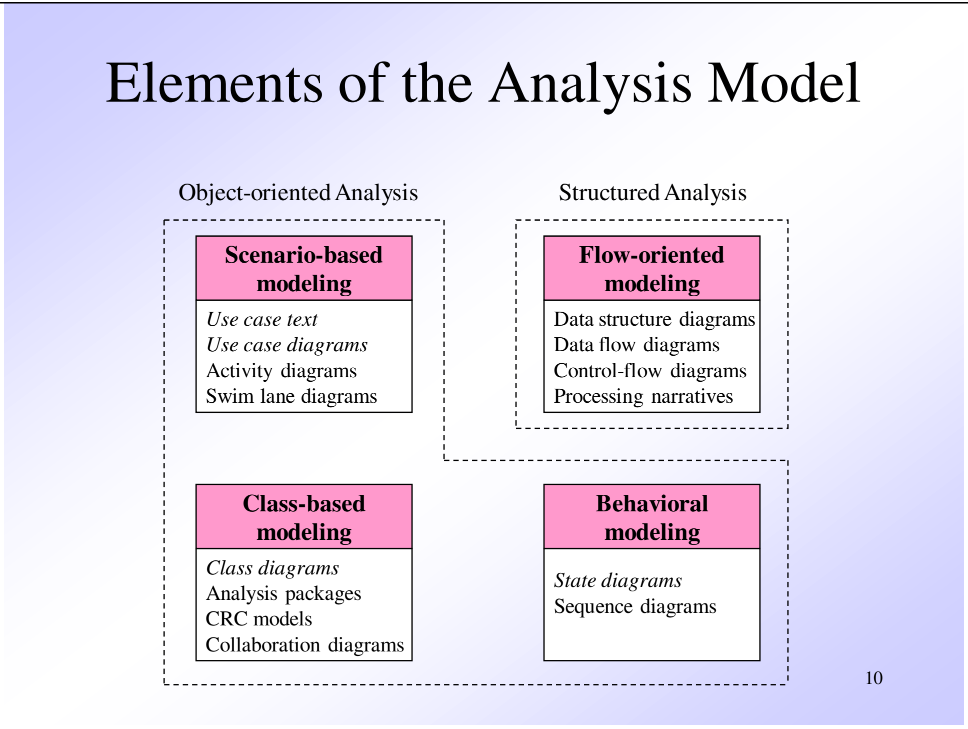
- Provides the first technical representation of a system
- Is easy to understand and maintain
- Deals with the problem of size by partitioning the system
- Uses graphics whenever possible
- Differentiates between essential information versus implementation information
- Helps in the tracking and evaluation of interfaces
- Provides tools other than narrative text to describe software logic and policy

A Set of Models

- **Flow-oriented modeling** – provides an indication of how data objects are transformed by a set of processing functions
- **Scenario-based modeling** – represents the system from the user's point of view
- **Class-based modeling** – defines objects, attributes, and relationships
- **Behavioral modeling** – depicts the states of the classes and the impact of events on these states

Analysis Modeling Approaches

- Structured analysis
  - Considers data and the processes that transform the data as separate entities
  - Data is modeled in terms of only attributes and relationships (but no operations)
  - Processes are modeled to show the 1) input data, 2) the transformation that occurs on that data, and 3) the resulting output data
- Object-oriented analysis
  - Focuses on the definition of classes and the manner in which they collaborate with one another to fulfill customer requirements



Elements of the Analysis Model

Flow-oriented Modeling

1. **Data Model**

- Identify the following items
  - Data objects (Entities)
  - Data attributes
  - Relationships
  - Cardinality (number of occurrences)

Data Flow and Control Flow

- Data Flow Diagram
  - Depicts how input is transformed into output as data objects move through a system.
- Process Specification
  - Describes data flow processing at the lowest level of refinement in the data flow diagrams.
- Control Flow Diagram
  - Illustrates how events affect the behavior of a system through the use of state diagrams.

2. **Scenario-based Modeling**

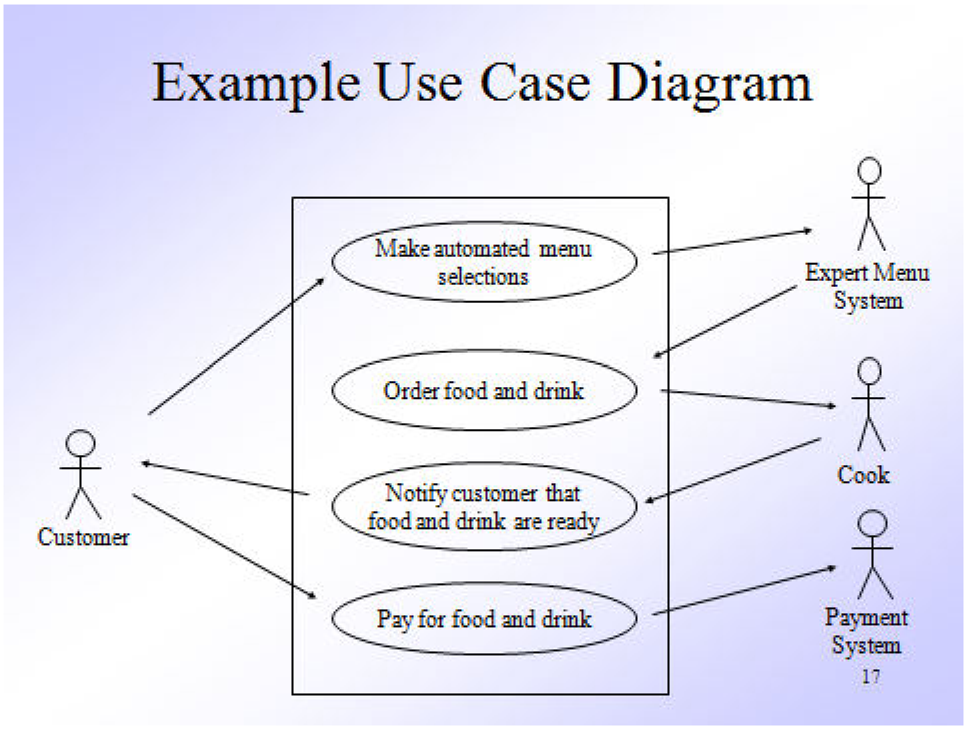
   Writing Use Cases

- Writing of use cases was previously described in Chapter 7 – Requirements Engineering.
- It is effective to use the first person “I” to describe how the actor interacts with the software.
- Format of the text part of a use case.

Use-case title:

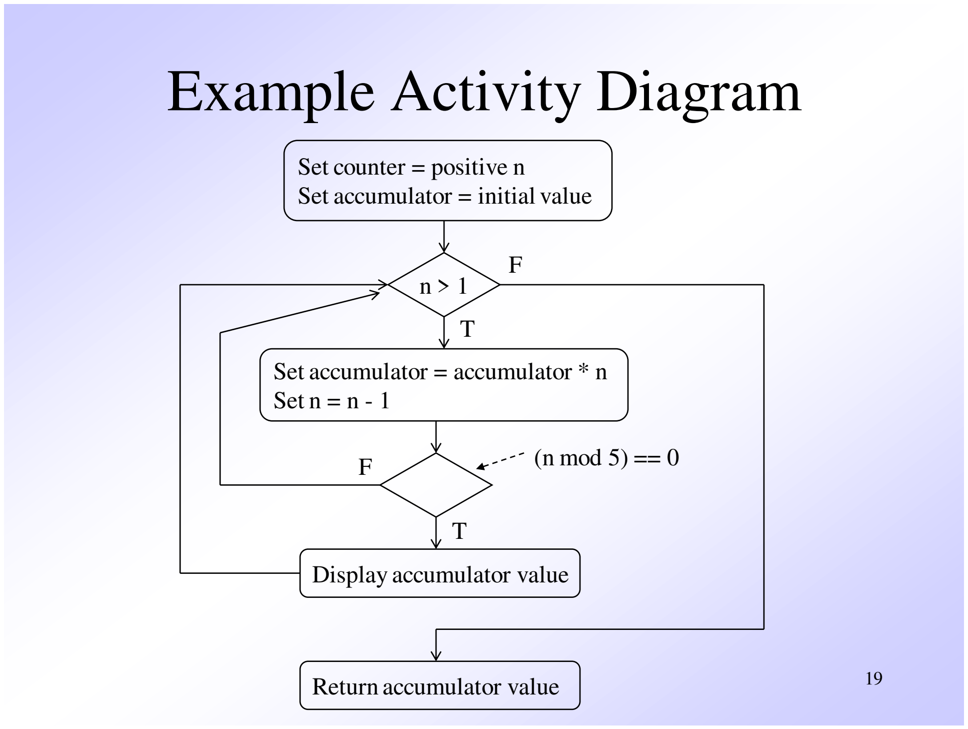
Actor:

Description: I …



Activity Diagrams

- Creation of activity diagrams was previously described in Chapter 7 – Requirements Engineering.
- Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.
- Uses flowchart-like symbols
  - Rounded rectangle - represent a specific system function/action.
  - Arrow - represents the flow of control from one function/action to another.
  - Diamond - represents a branching decision.
  - Solid bar – represents the fork and join of parallel activities.



Example Activity Diagram

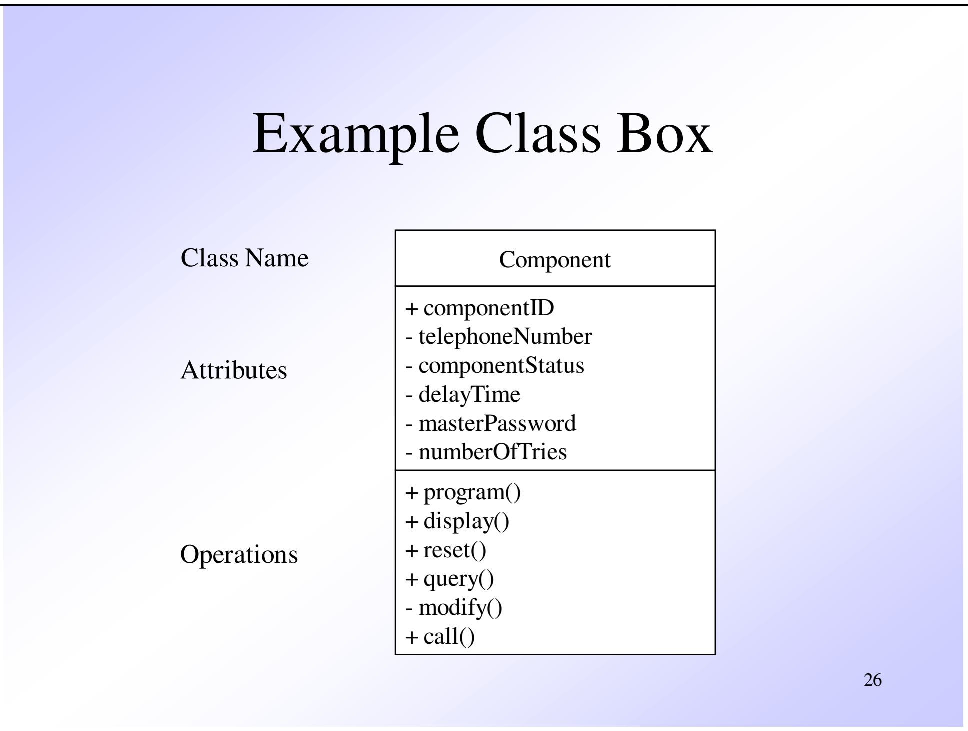
Set counter = positive n
Set accumulator = initial value
n > 1
F
T
Set accumulator = accumulator * n
Set n = n - 1
F
(n mod 5) == 0
T
Display accumulator value
Return accumulator value
19

Class-based Modeling

Identifying Analysis Classes

1) Perform a grammatical parse of the problem statement or use cases.
2) Classes are determined by underlining each noun or noun clause.
3) A class required to implement a solution is part of the solution space.
4) A class necessary only to describe a solution is part of the problem space.
5) A class should NOT have an imperative procedural name (i.e., a verb)
6) List the potential class names in a table and "classify" each class according to some taxonomy and class selection characteristics.
7) A potential class should satisfy nearly all (or all) of the selection characteristics to be considered a legitimate problem domain class.

- General classifications for a potential class
- External entity (e.g., another system, a device, a person)
- Thing (e.g., report, screen display)
- Occurrence or event (e.g., movement, completion)
- Role (e.g., manager, engineer, salesperson)
- Organizational unit (e.g., division, group, team)
- Place (e.g., manufacturing floor, loading dock)
- Structure (e.g., sensor, vehicle, computer)



Example Class Box

Class Name

Attributes

Operations

Component
+ componentID - telephoneNumber - componentStatus - delayTime - masterPassword - numberOfTries

+ program()
+ display()
+ reset()
+ query()
- modify()
+ call()

26

Association, Generalization and Dependency

- Association
  - Represented by a solid line between two classes directed from the source class to the target class
  - Used for representing (i.e., pointing to) object types for attributes
  - May also be a part-of relationship (i.e., aggregation), which is represented by a diamond-arrow
- Generalization
  - Portrays inheritance between a super class and a subclass
  - Is represented by a line with a triangle at the target end
- Dependency
  - A dependency exists between two elements if changes to the definition of one element (i.e., the source or supplier) may cause changes to the other element (i.e., the client)
  - Examples
    - One class calls a method of another class
    - One class utilizes another class as a parameter of a method

3. **Behavioral Modeling**

1. Identify events found within the use cases and implied by the attributes in the class diagrams.
2. Build a state diagram for each class, and if useful, for the whole software system.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the requirement analysis & Modelling.	Jun.2011	10

REFERENCCE

BOOK	**AUTHOR**	**PRIORITY**
Software Engineering	P,S. Pressman	1
Software Engineering	Pankaj jalote	2