SYLLABUS OF SEMESTER –I, M.C.A. (Master In Computer Application)

Course Code: MCT543 **Course: Concept in Software Engineering**

Section -II (Weightage – 50%, Minimum Theory Teaching Hours -15)

Software Project Management : Software project estimation and planning, Decomposition techniques, Risk Management, Requirement analysis.

Object Oriented Analysis: Object oriented analysis and data modeling, Object oriented concepts, Class Based Modeling.

Agile Development: About Agility, Agility and cost of change, Agile process, Agile process models (Adaptive software development, Scrum, Dynamic system development method), Agile Software development Approaches

Software Design Engineering: The design process and fundamentals, Effective modular Design, Data flow oriented design, Transform analysis, Transaction analysis, Design heuristics.

Software Project Management : Software project estimation and planning, Decomposition techniques, Risk Management, Requirement analysis.

THE MANAGEMENT SPECTRUM

- Effective software project management focuses on the **four P’s: people, product, process, and project.**
- The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management.
- A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem.
- The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum.
- The manager who embarks without a solid project plan jeopardizes the success of the product.

The People

- The Software Engineering Institute has developed a ***People Capability Maturity Model (People-CMM).***
- The people capability maturity model *(People-CMM)* defines the key practice areas for software people: **staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others**.
- Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

The Product

- Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

- As a software developer, you and other stakeholders must meet to define product objectives and scope.

- Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved.

- Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

- Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

The Process

- ❑ A software process provides the framework from which a comprehensive plan for software development can be established.
- ❑ A small number of framework activities are applicable to all software projects, regardless of their size or complexity.
- ❑ A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.
- ❑ Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

- In a study of 250 large software projects between 1998 and 2004, Capers Jones found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were terminated without completion."
- Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.
- To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project.

SOFTWARE PROJECT PLANNING

- Software project management begins with a set of activities that are collectively called ***project planning***.
- Before the project can begin, the manager and the software team must *estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish*.
- Project planning provides the road map for successful software engineering.
- ***Project complexity*** has a strong effect on the uncertainty inherent in planning. Complexity, however, is a relative measure that is affected by familiarity with past effort. The first-time developer of a sophisticated e-commerce application might consider it to be exceedingly complex.
- ***Project size*** is another important factor that can affect the accuracy and efficacy of estimates. *As size increases, the interdependency among various elements of the software grows rapidly.*

THE PROJECT PLANNING PROCESS

- ❑ The planning objective is achieved through a process of information discovery that leads to reasonable estimates.
- ❑ The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- ❑ In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.
- ❑ Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks.
- ❑ Therefore, the plan must be adapted and updated as the project proceeds.

SOFTWARE SCOPE AND FEASIBILITY

- *Software scope* describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a consequence of using the software; and the performance, constraints, interfaces, and reliability that bound the system.
- *Scope is defined using one of two techniques:*

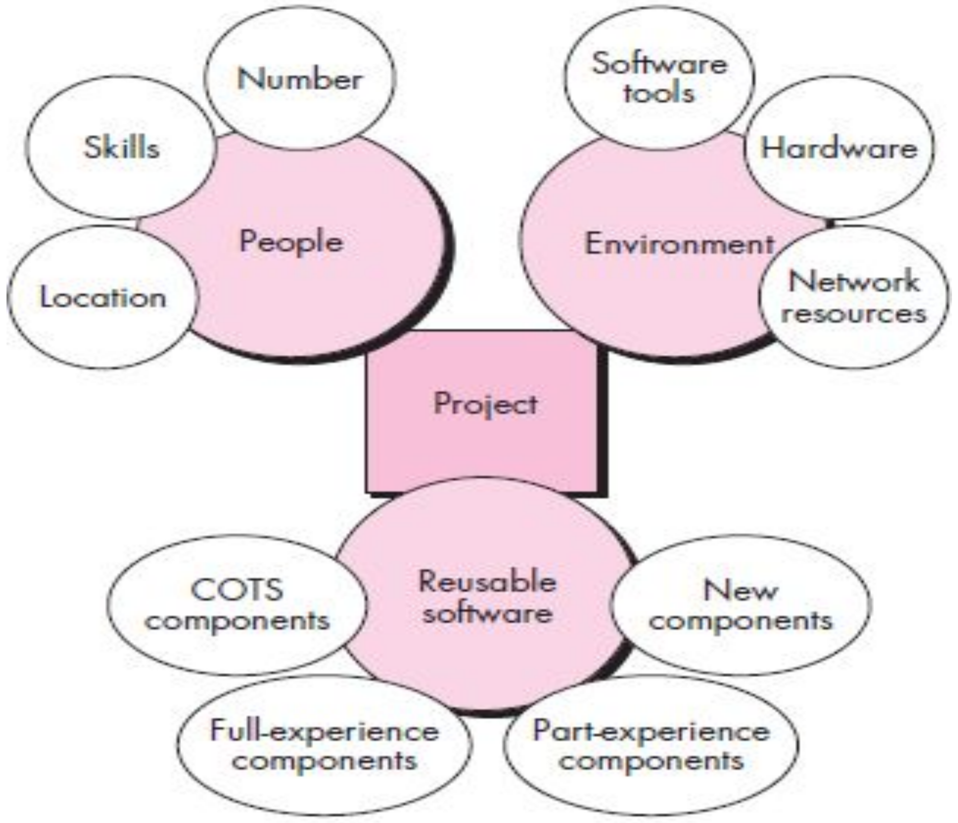
    1. A narrative description of software scope is developed after communication with all stakeholders.

    2. A set of use cases is developed by end users.

- Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful.
- Performance considerations encompass processing and response time requirements.
- Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

RESOURCES

- The second planning task is estimation of the resources required to accomplish the software development effort. Figure depicts the three major categories of software engineering resources — ***people, reusable software components, and the development environment (hardware and software tools)***.
- Each resource is specified with four characteristics: **description of the resource**, **a statement of availability**, **time when the resource will be required**, **and duration of time that the resource will be applied.**



Project resources

Human Resources

- **The planner begins by evaluating software scope and selecting the skills required to complete development.** Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are specified.
- For relatively **small projects** (a few person-months), **a single individual may perform all software engineering tasks**, consulting with specialists as required.
- For **larger projects**, **the software team may be geographically dispersed** across a number of different locations.
- Hence, the **location of each human resource is specified**.
- The number of **people required for a software project can be determined only after an estimate of development effort** (e.g., person-months) is made.

Reusable Software Resources

- ❑ **Component-based software engineering** emphasizes reusability—that is, the **creation and reuse of software building blocks**.
- ❑ Such building blocks, often called ***components***, must be **cataloged for easy reference, standardized for easy application, and validated for easy integration**.
- ❑ *Bennatan* suggests **four software resource categories** that should be considered as planning proceeds:
  - ***Off-the-shelf components.*** *Existing software that can be acquired from a third party or from a past project.* COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.
  - ***Full-experience components.*** *Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project.* Members of the current software team have had full experience in the application area represented by these components. *Therefore, modifications required for full-experience components will be relatively low risk.*

- **_Partial-experience components._** *Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification.* Members of the current software team have only limited experience in the application area represented by these components. *Therefore, modifications required for partial-experience components have a fair degree of risk.*

- **_New components._** *Software components must be built by the software team specifically for the needs of the current project.* Ironically, reusable software components are often neglected during planning, only to become a paramount concern later in the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

Environmental Resources

- The environment that supports a software project, often called the ***software engineering environment (SEE)***, incorporates hardware and software.
- Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.
- Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available.
- When a computer-based system is to be engineered, the software team may require access to hardware elements being developed by other engineering teams.
- For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development.
- Each hardware element must be specified as part of planning.

SOFTWARE PROJECT ESTIMATION

- ❑ In the early days of computing, software costs constituted a small percentage of the overall computer-based system cost. An order of magnitude error in estimates of software cost had relatively little impact.
- ❑ Today, software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss. Cost overrun can be disastrous for the developer.
- ❑ Software cost and effort estimation will never be an exact science. Too many variables— human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it.
- ❑ *To achieve reliable cost and effort estimates, a number of options arise:*

    1. Delay estimation until late in the project (obviously, we can achieve 100% accurate estimates after the project is complete!).
    2. Base estimates on similar projects that have already been completed.
    3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
    4. Use one or more empirical models for software cost and effort estimation.

- *Decomposition techniques* take a "divide and conquer" approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion.
- *Empirical estimation models* can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right.
- A model is based on experience (historical data) and takes the form

  $d = f(vi)$

  where *d is one of a number of estimated values (e.g., effort, cost, project duration)* and *vi are selected independent parameters (e.g., estimated LOC or FP).*
- *Automated estimation* tools implement one or more decomposition techniques or empirical models. When combined with a graphical user interface, automated tools provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data.
- Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. **If no historical data exist, costing rests on a very shaky foundation.**

DECOMPOSITION TECHNIQUES

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

(1) the degree to which you have properly estimated the size of the product to be built;

(2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);

(3) the degree to which the project plan reflects the abilities of the software team;

(4) the stability of product requirements and the environment that supports the software engineering effort.

- *Putnam and Myers suggest four different approaches to the sizing problem:*
  - ***"Fuzzy logic" sizing.*** This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.
  - ***Function point sizing.*** The planner develops estimates of the information domain characteristics.
  - ***Standard component sizing.*** Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.
  - ***Change sizing.*** This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation: (1) as estimation variables to “size” each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.

A three-point or expected value can then be computed. The *expected value for the* estimation variable (size) *S can be computed as a weighted average of the optimistic* (*sopt), most likely (sm), and pessimistic (spess) estimates. For example,*

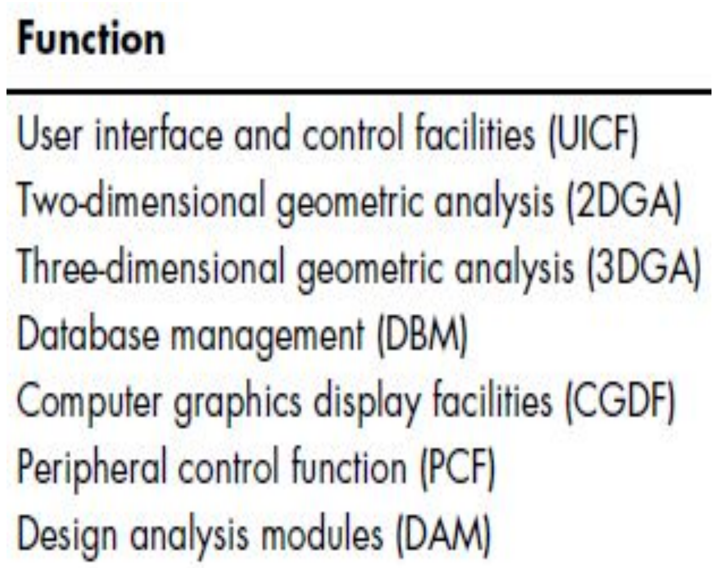
$$S = (sopt + 4sm + spess)/6$$

gives heaviest credence to the “most likely” estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Process-Based Estimation

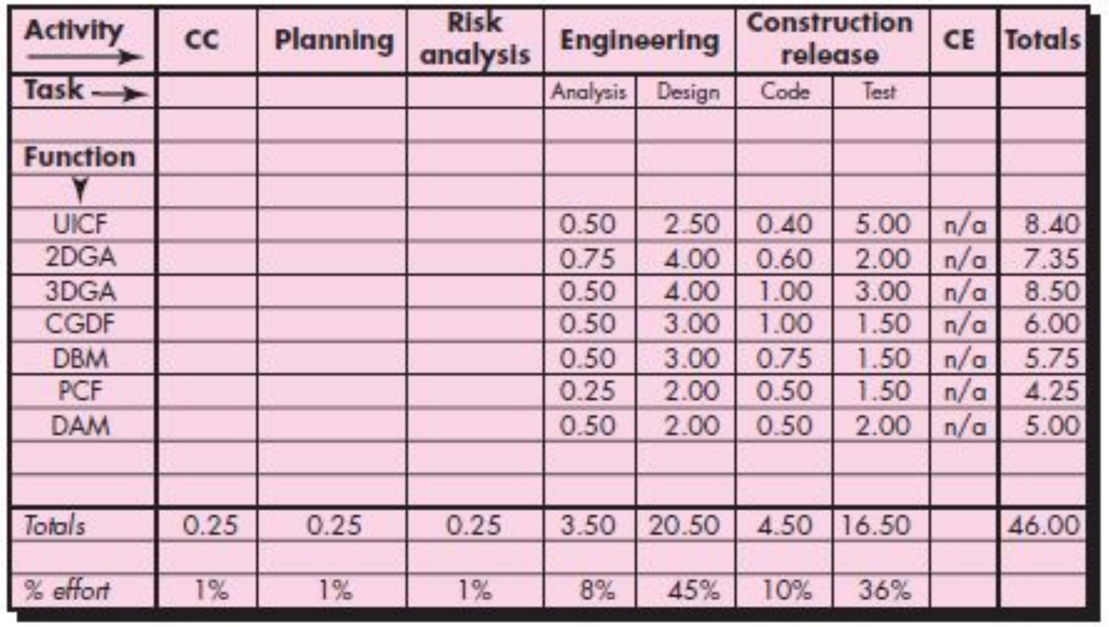
The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function. Functions and related framework activities8 may be represented as part of a table similar to the one presented in Figure.



Function

User interface and control facilities (UICF)
Two-dimensional geometric analysis (2DGA)
Three-dimensional geometric analysis (3DGA)
Database management (DBM)
Computer graphics display facilities (CGDF)
Peripheral control function (PCF)
Design analysis modules (DAM)



Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Process-based estimation table

Estimation with Use Cases

- However, developing an estimation approach with use cases is problematic for the following reasons :
  - Use cases are described using many different formats and styles—there is no standard form.
  - Use cases represent an external view (the user's view) of the software and can therefore be written at many different levels of abstraction.
  - Use cases do not address the complexity of the functions and features that are described.
  - Use cases can describe complex behavior (e.g., interactions) that involve many functions and features.
- Unlike an LOC or a function point, one person's "use case" may require months of effort while another person's use case may be implemented in a day or two.

EMPIRICAL ESTIMATION MODELS

- ❑ An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.
- ❑ The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments.

The Structure of Estimation Models

- ❑ A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form

  $$E = A + B \times (ev)^C$$

  where A, B, and C are empirically derived constants, E is effort in person-months, and *ev is the estimation variable (either LOC or FP).*
- ❑ In addition to the relationship noted in Equation, the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment).

- Among the many **LOC-oriented** estimation models proposed in the literature are

$E = 5.2 \times (KLOC)^{0.91}$	*Walston-Felix model*
$E = 5.5 + 0.73 \times (KLOC)^{1.16}$	*Bailey-Basili model*
$E = 3.2 \times (KLOC)^{1.05}$	*Boehm simple model*
$E = 5.288 \times (KLOC)^{1.047}$	*Doty model for KLOC > 9*

FP-oriented models have also been proposed. These include:

$E= -91.4 + 0.355\ FP$	*Albrecht and Gaffney model*
$E= -37 + 0.96\ FP$	*Kemerer model*
$E= -12.88 +0.405\ FP$	*Small project regression model*

The COCOMO II Model

- In his classic book on "software engineering economics," Barry Boehm introduced a hierarchy of software estimation models bearing the name COCOMO, for ***COnstructive COst MOdel.***

- The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, **called COCOMOII**.

- Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

  - ***Application composition model.*** Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

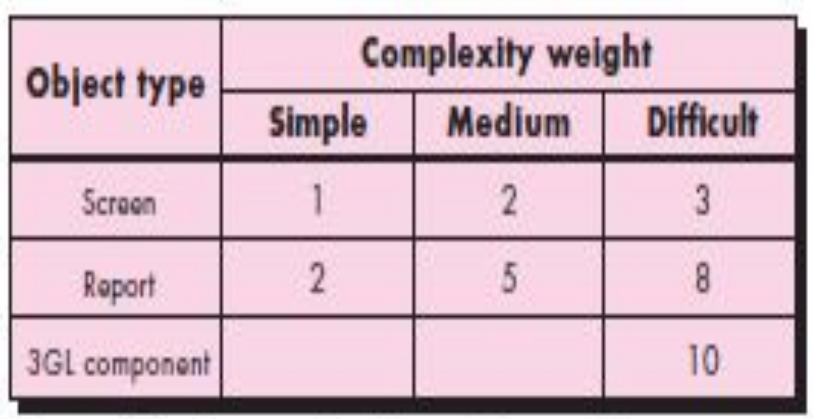
  - ***Early design stage model.*** Used once requirements have been stabilized and basic software architecture has been established.

  - ***Post-architecture-stage model.*** Used during the construction of the software.

- Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: **object points, function points, and lines of source code**.

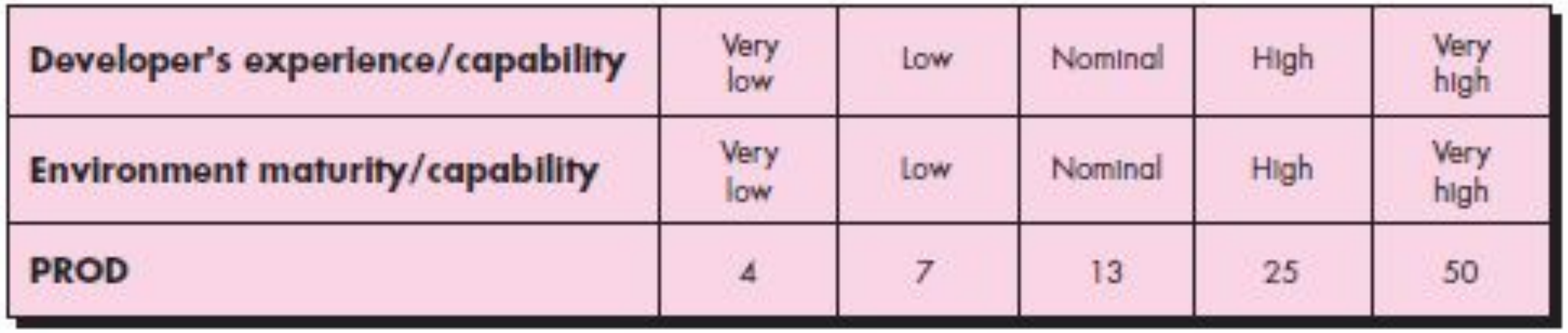
- Like function points, the *object point is an indirect software measure that is computed* using counts of the number of **(1) screens (at the user interface), (2) reports,** and **(3) components** likely to be required to build the application.
- Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult) using criteria suggested by Boehm.
- Complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.
- The object point count is then determined by multiplying the original number of object instances by the weighting factor in the figure and summing to obtain a total object point count. When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

  NOP = (object points) x [(100 - %reuse)/100]
  where NOP is defined as **new object points**.



Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Complexity weighting for object types



Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

Productivity rate for object points

❑ To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived. Figure presents the productivity rate

PROD = NOP/ person-month

for different levels of developer experience and development environment maturity.

❑ Once the productivity rate has been determined, an estimate of project effort is computed using

Estimated effort = NOP/ PROD

RISK MANAGEMENT

REACTIVE VERSUS PROACTIVE RISK STRATEGIES

- ***Reactive risk strategies*** have been laughingly called the “Indiana Jones school of risk management”. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, “Don’t worry, I’ll think of something!” Never worrying about problems until they happened, Indy would react in some heroic way.

- The majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems.

- More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire-fighting mode.* When this fails, “crisis management” takes over and the project is in real jeopardy.

- A considerably more intelligent strategy for risk management is to be *proactive*. A ***proactive strategy*** begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance.

- Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

SOFTWARE RISKS

- ❑ Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics:
  - ***Uncertainty***—*the risk may or may not happen; that is, there are no 100% probable* risks.
  - ***Loss***—*if the risk becomes a reality, unwanted consequences or losses will* occur.
- ❑ When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.
- ❑ ***Project risks*** *threaten the project plan.* That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.
- ❑ ***Technical risks*** threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.
- ❑ In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors.

- ❑ **Business risks** threaten the viability of the software to be built. Business risks often jeopardize the project or the product.
- ❑ *Candidates for the top five business risks are* **(1)** building a excellent product or system that no one really wants (**market risk**), **(2)** building a product that no longer fits into the overall business strategy for the company (**strategic risk**), **(3)** building a product that the sales force doesn't understand how to sell, **(4)** losing the support of senior management due to a change in focus or a change in people (**management risk**), and **(5)** losing budgetary or personnel commitment (**budget risks**). Some risks are simply unpredictable in advance.
- ❑ *Another general categorization of risks has been proposed by Charette.*
  - ▪ ***Known risks*** are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).
  - ▪ ***Predictable risks*** are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).
  - ▪ ***Unpredictable risks*** are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

RISK IDENTIFICATION

- **Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.)**. By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

- There are two distinct types of risks: ***generic risks and product-specific risks***.
  - ***Generic risks*** are a potential threat to every software project.
  - ***Product-specific risks*** can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

- **One method for identifying risks is to create a risk item checklist.** The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:
  - ***Product size***—risks associated with the overall size of the software to be built or modified.
  - ***Business impact***—risks associated with constraints imposed by management or the marketplace.

- ***Stakeholder characteristics***—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
- ***Process definition***—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.
- ***Technology to be built***—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- ***Staff size and experience***—risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk Components and Drivers

- The risk components are defined in the following manner:
  - ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
  - ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
  - ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
  - ***Schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.
- The impact of each risk driver on the risk component is divided into one of four impact categories—*negligible, marginal, critical, or catastrophic*.

RISK PROJECTION

Risk projection, also called ***risk estimation***, attempts to rate each risk in two ways—

(1) the likelihood or probability that the risk is real and

(2) the consequences of the problems associated with the risk, should it occur.

You work along with other managers and technical staff to perform four risk projection steps:

1. Establish a scale that reflects the perceived possibility of a risk.
2. Define the consequences of the risk.
3. Estimate the impact of the risk on the project and the product.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

RISK MITIGATION, MONITORING, AND MANAGEMENT

- All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk.
- An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.
- If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.* For example, assume that high staff turnover is noted as a project risk r1. Based on past history and management intuition, the likelihood l1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule.
- To **mitigate this risk**, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:
  - Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
  - Mitigate those causes that are under your control before the project starts.

- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is “up to speed”).
- Assign a backup staff member for every critical technologist.

Requirements Analysis

- Requirements analysis
  - specifies software's operational characteristics
  - indicates software's interface with other system elements
  - establishes constraints that software must meet
- Requirements analysis allows the software engineer (called an *analyst* or *modeler* in this role) to:
  - elaborate on basic requirements established during earlier requirement engineering tasks
  - build models that depict
    - user scenarios,
    - functional activities,
    - problem classes and their relationships,
    - system and class behavior, and
    - the flow of data as it is transformed.

Object Oriented Analysis: Object oriented analysis and data modeling, Object oriented concepts, Class Based Modeling.

UNIT-III

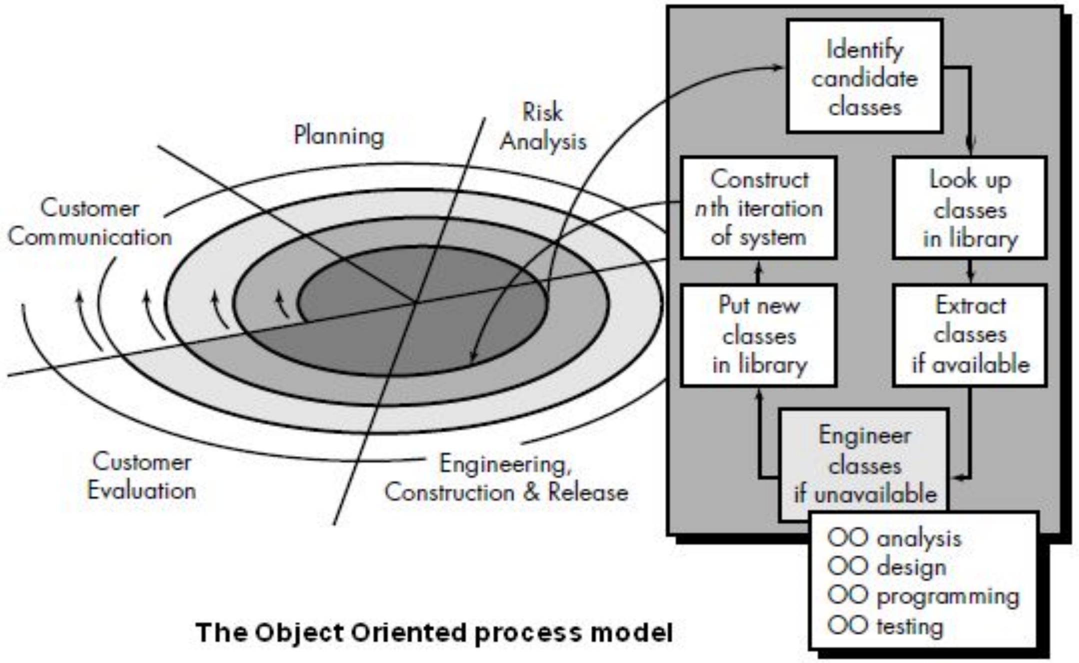
Object Oriented Analysis: Object oriented analysis and data modeling, Object oriented concepts, Class Based Modeling, Inter object communication, Finalizing object definition, Object oriented analysis modeling.

OBJECT-ORIENTED CONCEPTS

- An object-oriented approach to the development of software was **first proposed in the late 1960s.**
- **Throughout the 1990s, object-oriented software engineering became the paradigm of choice for many software product builders** and a growing number of information systems and engineering professionals.
- **Object technologies lead to reuse**, and reuse (of program components) leads to faster software development and higher-quality programs.
- **Object-oriented software is easier to maintain** because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer.
- In addition, **object-oriented systems are easier to adapt and easier to scale** (i.e., large systems can be created by assembling reusable subsystems).

THE OBJECT-ORIENTED PARADIGM

- The term *object oriented (OO)* was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Ada95, Java, C++, Eiffel, Smalltalk). Today, the **OO paradigm encompasses a complete view of software engineering**.
- **The OO process moves through an evolutionary spiral that starts with customer communication.** It is here that the problem domain is defined and that basic problem classes are identified.
- **Planning and risk analysis establish a foundation for the OO project plan**. The technical work associated with OO software engineering follows the iterative path shown in the shaded box. OO software engineering emphasizes reuse.

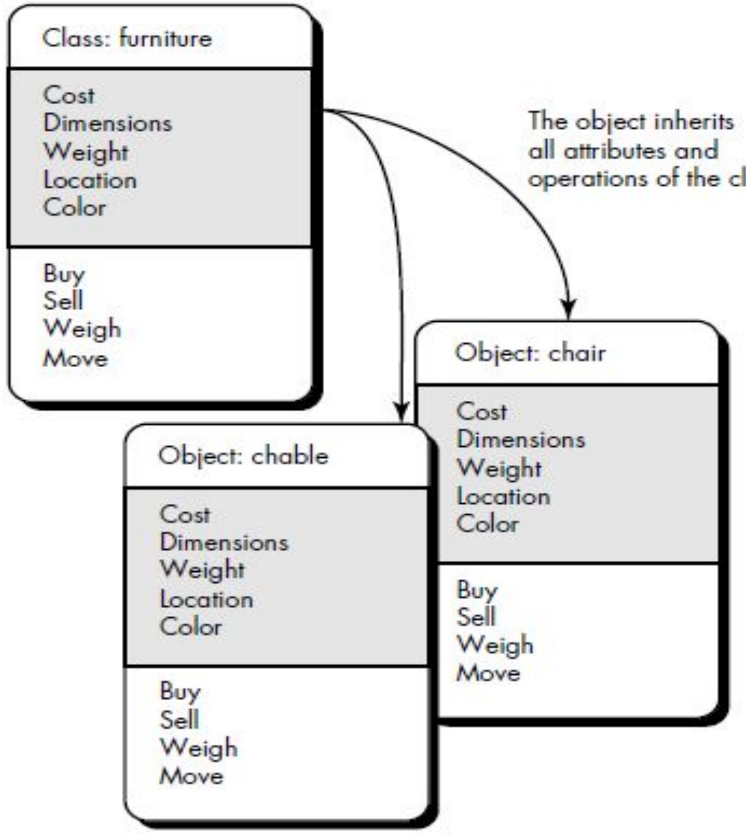

Planning
Risk Analysis
Customer Communication
Customer Evaluation
Engineering, Construction & Release
Identify candidate classes
Look up classes in library
Extract classes if available
Engineer classes if unavailable
Put new classes in library
Construct nth iteration of system
OO analysis
OO design
OO programming
OO testing

The Object Oriented process model

- **When a class cannot be found in the library**, the software engineer applies **object-oriented analysis (OOA), object-oriented design (OOD), object-oriented programming (OOP), and object-oriented testing (OOT)** to create the class and the objects derived from the class.
- **The new class is then put into the library** so that it may be reused in the future.

OBJECT-ORIENTED CONCEPTS

- To understand the object-oriented point of view, consider an example of a real world object—the thing you are sitting in right now—a chair. **Chair** is a member (the term *instance* is also used*)* of a much larger class of objects that we call ***furniture****.*
- *A* set of generic attributes can be associated with every object in the class **furniture.** For example, all furniture has a *cost, dimensions, weight, location, and color*, among many possible *attributes.*
- These apply whether we are talking about a table or a chair, a sofa or an armoire. Because **chair** is a member of **furniture, chair** *inherits* all attributes defined for the class.

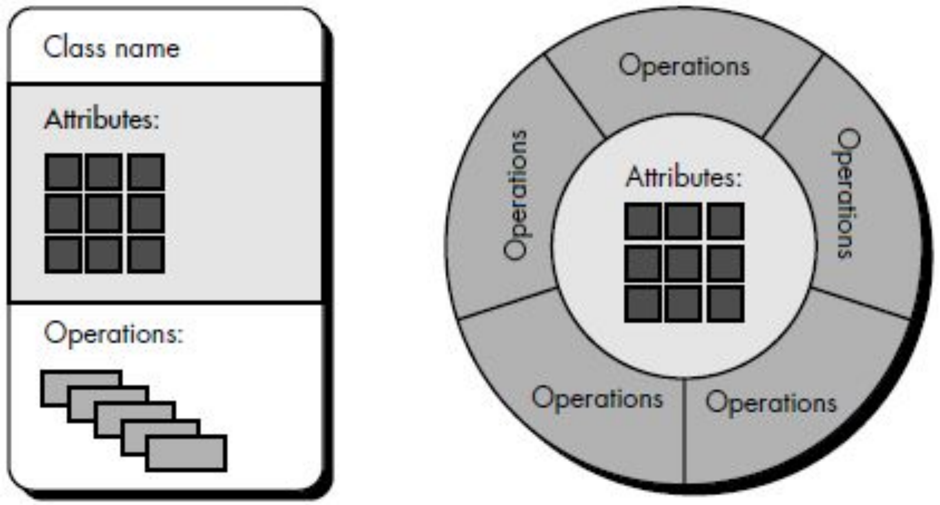


- The object **chair** (and all objects in general) **encapsulates data** (the attribute values that define the chair), **operations** (the actions that are applied to change the attributes of chair), **other objects** (composite objects can be defined), **constants** (set values), and other related information.
- ***Encapsulation*** means that all of this information is packaged under one name and can be reused as one specification or program component.
- Coad and Yourdon define the term this way:

  object-oriented = objects + classification + inheritance + communication

Classes and Objects

- The fundamental concepts that lead to high-quality design apply equally to systems developed using conventional and object-oriented methods.
- For this reason, an OO model of computer software must exhibit data and procedural abstractions that lead to effective modularity.
- **A class is an OO concept that encapsulates the data and procedural abstractions** required to describe the content and behavior of some real world entity.
- The data abstractions (attributes) that describe the class are enclosed by a "wall" of procedural abstractions (called *operations, methods, or services)* that are capable of manipulating the data in some way.
- The only way to reach the attributes (and operate on them) is to go through one of the methods that form the wall.



Operations, Methods, and Services

- **An object encapsulates data** (represented as a collection of attributes) and **the algorithms that process the data**. These algorithms are called operations, methods, or services and can be viewed as modules in a conventional sense.

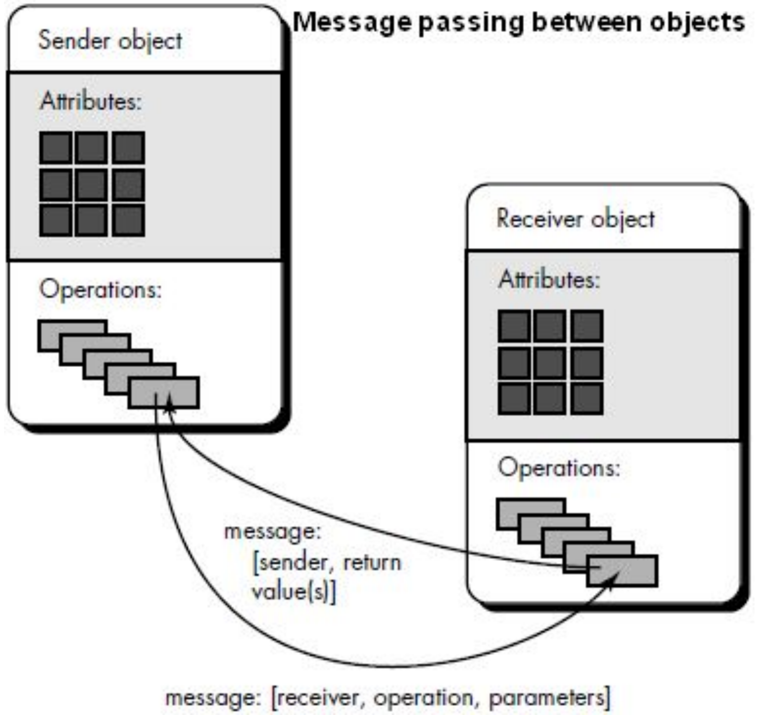
Messages

- **Messages are the means by which objects interact**.
- A message stimulates some behavior to occur in the receiving object.
- The behavior is accomplished when an operation is executed.
- An operation within a sender object generates a message of the form

Message: [destination, operation, parameters]

where ***destination*** defines the receiver object that is stimulated by the message, ***operation*** refers to the operation that is to receive the message, and ***parameters*** provides information that is required for the operation to be successful.

Message passing between objects

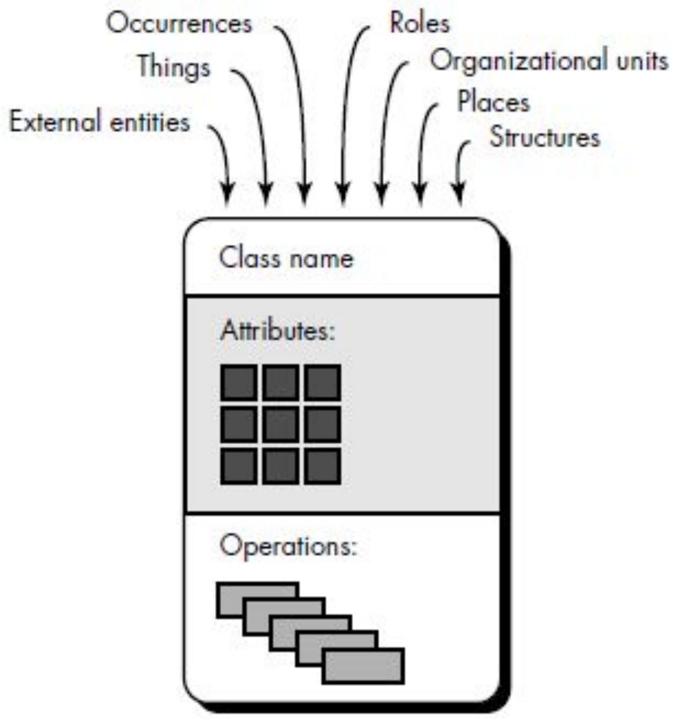

Sender object
Attributes:
Operations:
Receiver object
Attributes:
Operations:
message: [sender, return value(s)]
message: [receiver, operation, parameters]

Identifying Classes and Objects

Objects manifest themselves in one of the ways represented in figure.

Objects can be:

- ***External entities*** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- ***Things*** (e.g, reports, displays, letters, signals) that are part of the information domain for the problem.
- ***Occurrences or events*** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.



- ***Roles*** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- ***Organizational units*** (e.g., division, group, team) that are relevant to an application.
- ***Places*** *(*e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- ***Structures*** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or in the extreme, related classes of objects.

Coad and Yourdon suggest **six selection characteristics** that should be used as an analyst considers **each potential object for inclusion** in the analysis model:

1. **Retained information.** The potential object will be useful during analysis only if information about it must be remembered so that the system can function.
2. **Needed services.** The potential object must have a set of identifiable operations that can change the value of its attributes in some way.
3. **Multiple attributes.** During requirement analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another object during the analysis activity.
4. **Common attributes.** A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.
5. **Common operations.** A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.
6. **Essential requirements.** External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

OBJECT-ORIENTED ANALYSIS

- ❑ The objective of object-oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer-defined requirements.
- ❑ OOA, like the conventional analysis methods, builds a multipart analysis model to satisfy this objective.
- ❑ OOA is grounded in a set of basic principles. In order to build an analysis model, **five basic principles were applied:**
  (1) the information domain is modeled; (2) function is described; (3) behavior is represented; (4) data, functional, and behavioral models are partitioned to expose greater detail; and (5) early models represent the essence of the problem while later models provide implementation details.
- ❑ The goal of OOA is to define all classes that are relevant to the problem to be solved—the operations and attributes associated with them, the relationships between them, and behavior they exhibit. *To accomplish this, a number of tasks must occur:*
  1. **Basic user requirements must be communicated between the customer and the software engineer.**
  2. **Classes must be identified (i.e., attributes and methods are defined).**
  3. **A class hierarchy must be specified.**
  4. **Object-to-object relationships (object connections) should be represented.**
  5. **Object behavior must be modeled.**
  6. **Tasks 1 through 5 are reapplied iteratively until the model is complete.**

Conventional vs. OO Approaches

- Stated simply, structured analysis (SA) takes a distinct input-process-output view of requirements.
- Data are considered separately from the processes that transform the data. System behavior, although important, tends to play a secondary role in structured analysis.
- The structured analysis approach makes heavy use of functional decomposition.
- *Fichman and Kemerer* suggest 11 “modeling dimensions” that may be used to compare various conventional and object-oriented analysis methods:
    1. **Identification/classification of entities**
    2. **General-to-specific and whole-to-part entity relationships**
    3. **Other entity relationships**
    4. **Description of attributes of entities**
    5. **Large-scale model partitioning**
    6. **States and transitions between states**
    7. **Detailed specification for functions**
    8. **Top-down decomposition**
    9. **End-to-end processing sequences**
    10. **Identification of exclusive services**
    11. **Entity communication (via messages or events)**

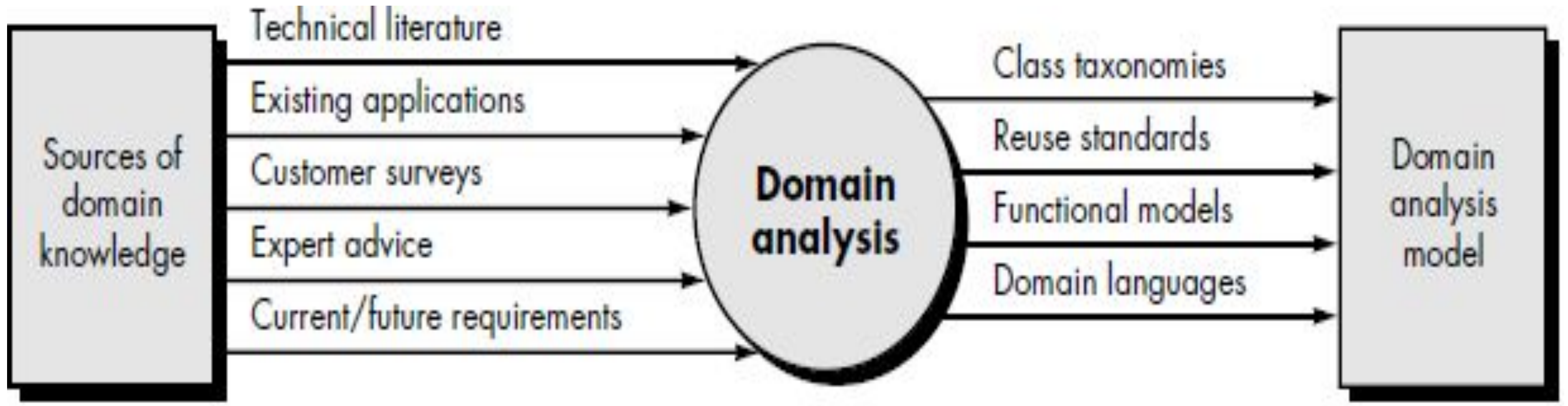
It can be stated that modeling dimensions 8 and 9 are always present with SA and never present when OOA is used.

Reuse and Domain Analysis

- Object-technologies are influenced through reuse. *Consider a simple example. The analysis of requirements for a new application indicates that 100 classes are needed.*
- Two teams are assigned to build the application. Each will design and construct a final product. Each team is populated by people with the same skill levels and experience.
- Team A does not have access to a class library, and therefore, it must develop all 100 classes from scratch.
- Team B uses a robust class library and finds that 55 classes already exist.
- It is highly likely that:
    1. Team B will finish the project much sooner than Team A.
    2. The cost of Team B’s product will be significantly lower than the cost of Team A’s product.
    3. The product produced by Team B will have fewer delivered defects than Team A’s product.
- But where did the “robust class library” come from? How were the entries in the library determined to be appropriate for use in new applications? To answer these questions, the organization that created and maintained the library had to apply domain analysis.

The Domain Analysis Process

- Figure illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.
- In essence domain analysis is quite similar to **knowledge engineering**. The knowledge engineer investigates a specific area of interest in an attempt to extract key facts that may be of use in creating an expert system or artificial neural network. During domain analysis, *object (and class) extraction* occurs.
- The domain analysis process can be characterized by a series of activities that begin with the identification of the domain to be investigated and end with a specification of the objects and classes that characterize the domain.



Input and output for domain analysis

- Berard suggests the following activities:

Define the domain to be investigated. To accomplish this, the analyst must first isolate the business area, system type, or product category of interest. Next, both OO and non-OO “items” must be extracted. OO items include specifications, designs, and code for existing OO application classes; support classes (e.g., GUI classes or database access classes); commercial off-the-shelf (COTS) component libraries that are relevant to the domain; and test cases. Non-OO items encompass policies, procedures, plans, standards, and guidelines; parts of existing non-OO applications (including specification, design, and test information); metrics; and COTS non-OO software.

Categorize the items extracted from the domain. The items are organized into categories and the general defining characteristics of the category are defined. A classification scheme for the categories is proposed and naming conventions for each item are defined. When appropriate, classification hierarchies are established.

Collect a representative sample of applications in the domain. To accomplish his activity, the analyst must ensure that the application in question has items that fit into the categories that have already been defined. Berard notes that during the early stages of use of object-technologies, a software organization will have few if any OO applications. Therefore, the domain analyst must “identify the conceptual (as opposed to physical) objects in each [non-OO] application.”

Analyze each application in the sample. The following steps are followed by the analyst:

- Identify candidate reusable objects.
- Indicate the reasons that the object has been identified for reuse.
- Define adaptations to the object that may also be reusable.
- Estimate the percentage of applications in the domain that might make reuse of the object.
- Identify the objects by name and use configuration management techniques to control them. In addition, once the objects have been defined, the analyst should estimate what percentage of a typical application could be constructed using the reusable objects.

Develop an analysis model for the objects. The analysis model will serve as the basis for design and construction of the domain objects.

GENERIC COMPONENTS OF THE OO ANALYSIS MODEL

- To develop a “precise, concise, understandable, and correct model of the real world,” a software engineer must select a notation that implements a set of generic components of an OO analysis model. Monarchi and Puhr define a set of generic representational components that appear in all OO analysis models.
- ***Static components*** are structural in nature and indicate characteristics that hold throughout the operational life of an application. These characteristics distinguish one object from other objects.
- ***Dynamic components*** focus on control and are sensitive to timing and event processing. They define how one object interacts with other objects over time.
- The following components are identified:

Static view of semantic classes. Requirements are assessed and classes are extracted (and represented) as part of the analysis model. These classes persist throughout the life of the application and are derived based on the semantics of the customer requirements.

Static view of attributes. Every class must be explicitly described. The attributes associated with the class provide a description of the class, as well as a first indication of the operations that are relevant to the class.

Static view of relationships. Objects are “connected” to one another in a variety of ways. The analysis model must represent these relationships so that operations (that affect these connections) can be identified and the design of a messaging approach can be accomplished.

Static view of behaviors. The relationships just noted define a set of behaviors that accommodate the usage scenario (use-cases) of the system. These behaviors are implemented by defining a sequence of operations that achieve them.

Dynamic view of communication. Objects must communicate with one another and do so based on a series of events that cause transition from one state of a system to another.

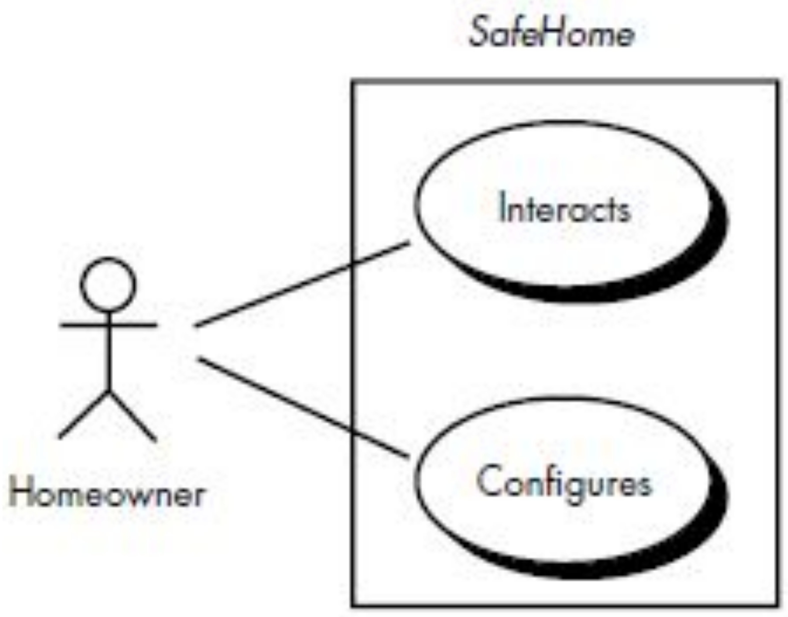
Dynamic view of control and time. The nature and timing of events that cause transitions among states must be described.

THE OOA PROCESS

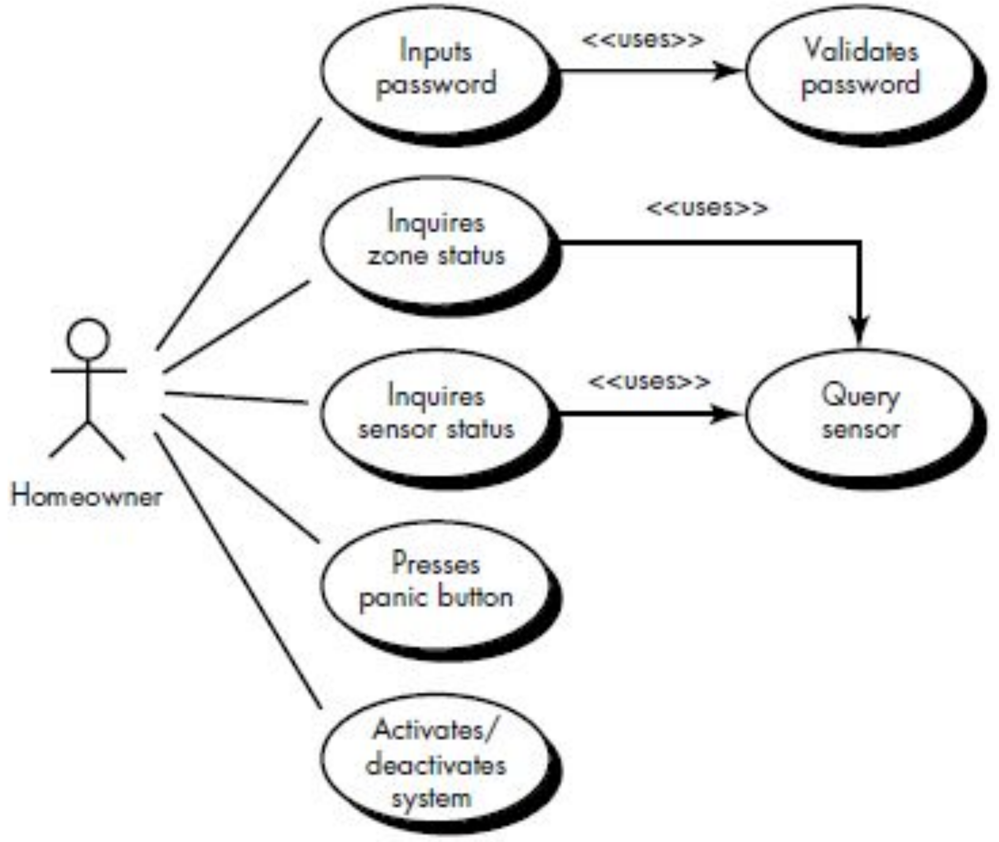
- The OOA process does not begin with a concern for objects.
- It begins with an understanding of the manner in which the system will be used—by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications.
- Once the scenario of usage has been defined, the modeling of the software begins.

Use-Cases

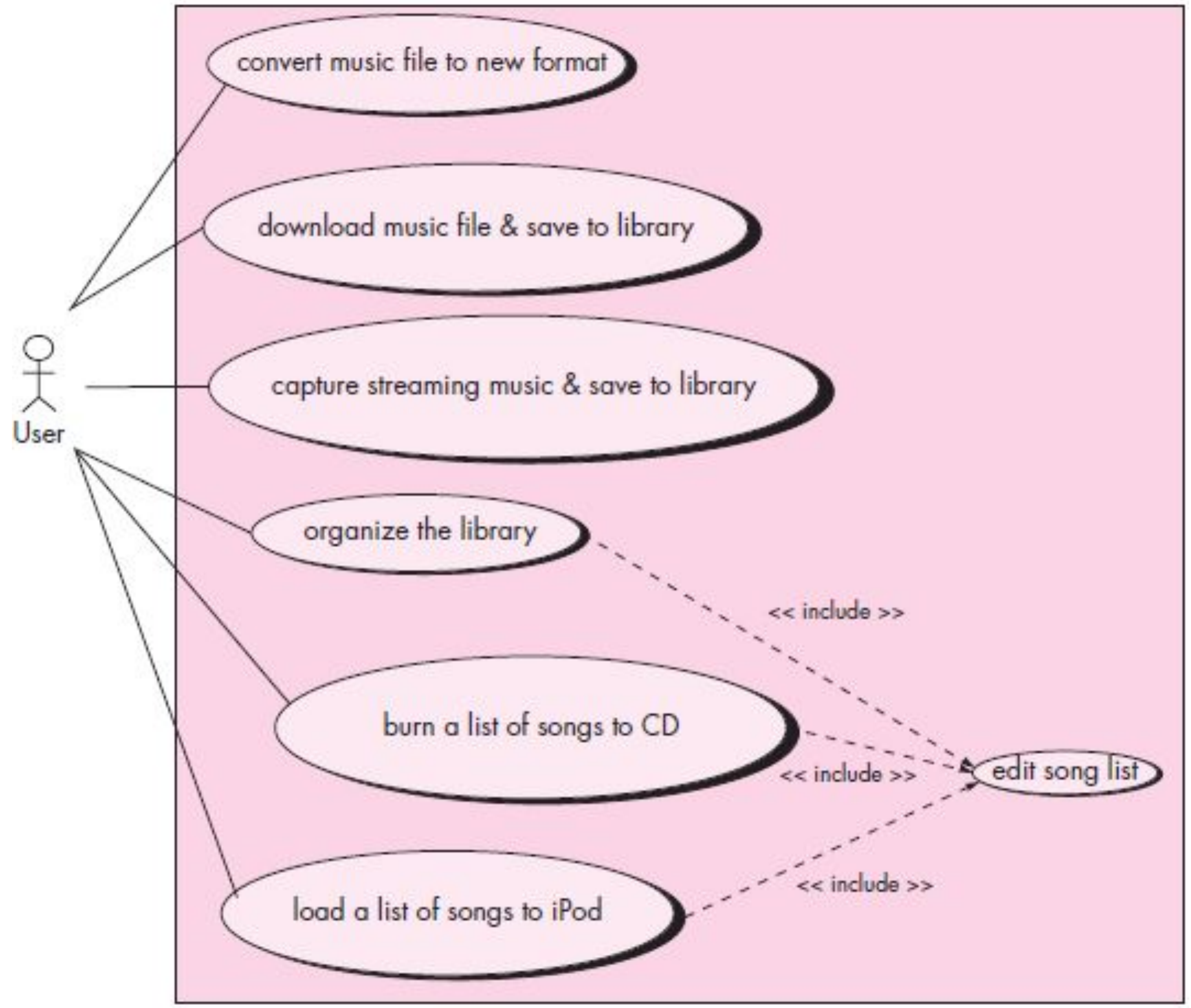
- use-cases model the system from the end-user's point of view.
- Created during requirements elicitation, use-cases should achieve the following objectives:
  - To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.
  - To provide a clear and unambiguous description of how the end-user and the system interact with one another.
  - To provide a basis for validation testing.
- Use-cases serve as the basis for the first element of the analysis model.
- Using UML notation, a diagrammatic representation of a use-case, called a *use-case diagram,* can be created.
- Like many elements of the analysis model, the *use-case diagram* can be represented at many levels of abstraction.
- The use-case diagram contains actors and use-cases.
- Actors are entities that interact with the system. They can be *human users or other machines or systems* that have defined interfaces to the software.



High-level use-case diagram



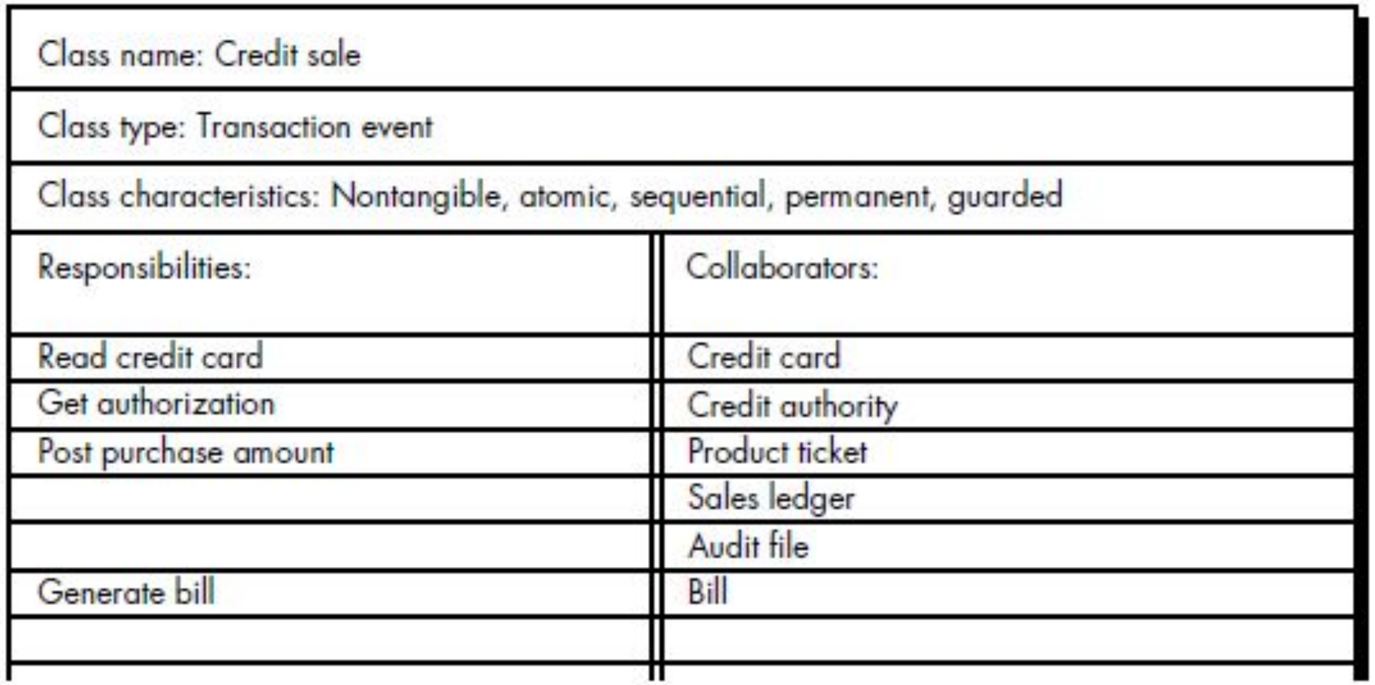
Elaborated use-case diagram


convert music file to new format
download music file & save to library
capture streaming music & save to library
User
organize the library
<< include >>
burn a list of songs to CD
<< include >>
edit song list
load a list of songs to iPod
<< include >>

A use-case diagram with included use cases

Class-Responsibility-Collaborator Modelling

- Once basic usage scenarios have been developed for the system, it is time to identify candidate classes and indicate their responsibilities and collaborations.
- ***Class responsibility- collaborator (CRC) modeling*** provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.
- Ambler describes CRC modeling in the following way:
  - A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class.



Class name: Credit sale	
Class type: Transaction event	
Class characteristics: Nontangible, atomic, sequential, permanent, guarded	
Responsibilities:	Collaborators:
Read credit card	Credit card
Get authorization	Credit authority
Post purchase amount	Product ticket
	Sales ledger
	Audit file
Generate bill	Bill

A CRC model index card

- ❑ In reality, the CRC model may make use of actual or virtual ***index cards***. The intent is to develop an organized representation of classes.
- ❑ ***Responsibilities*** are the attributes and operations that are relevant for the class. Stated simply, a responsibility is “anything the class knows or does”.
- ❑ ***Collaborators*** *are those classes that are* required to provide a class with the information needed to complete a responsibility.
- ❑ In general, a collaboration implies either a request for information or a request for some action.

Classes

- Basic guidelines for identifying classes and objects were presented in.
- To summarize, objects manifest themselves in a variety of forms: external entities, things, occurrences, or events; roles; organizational units; places; or structures.
- One technique for identifying these in the context of a software problem is to perform a grammatical parse on the processing narrative for the system.
- All nouns become potential objects. However, not every potential object makes the cut.
- Six selection characteristics were defined:

  1. **Retained information.**
  2. **Needed services.**
  3. **Multiple attributes.**
  4. **Common attributes.**
  5. **Common operations.**
  6. **Essential requirements.**

Firesmith extends this taxonomy of class types by suggesting the following additions:

- **Device classes model external entities such as sensors, motors, keyboards.**
- **Property classes represent some important property of the problem environment** (e.g., credit rating within the context of a mortgage loan application).
- **Interaction classes model interactions that occur among other objects**

(e.g., a purchase or a license).

In addition, objects and classes may be categorized by a set of characteristics:

Tangibility. Does the class represent a tangible thing (e.g., a keyboard or sensor) or does it represent more abstract information (e.g., a predicted outcome)?

Inclusiveness. Is the class *atomic (i.e., it includes no other classes) or is it aggregate (it includes at least one nested object)?*

Sequentiality. Is the class concurrent (i.e., it has its own thread of control) or sequential (it is controlled by outside resources)?

Persistence. Is the class *transient (i.e., it is created and removed during program* operation), *temporary (it is created during program operation and* removed once the program terminates), or *permanent (it is stored in a* database)?

Integrity. Is the class *corruptible (i.e., it does not protect its resources from* outside influence) or *guarded (i.e., the class enforces controls on access to its* resources)?

Collaborations

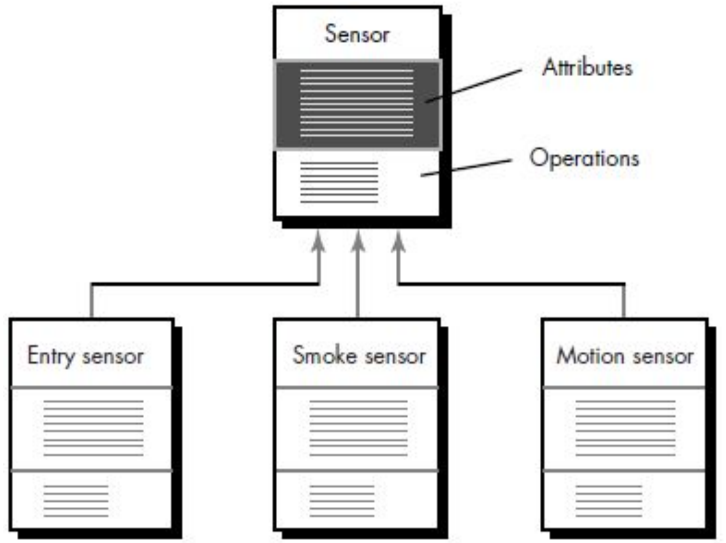
- *Classes fulfill their responsibilities in one of two ways:* (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes.
- Collaborations identify relationships between classes. When a set of classes all collaborate to achieve some requirement, they can be organized into a subsystem (a design issue).
- Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.
- As an example, consider the *SafeHome application. As part of the activation procedure, the* ***control panel object*** must determine whether any sensors are open. A responsibility named *determine sensor- status is defined. If sensors are open* ***control panel must set a status attribute*** to “not ready.” Sensor information can be acquired from the **sensor object. Therefore,** the responsibility *determine-sensor-status can be fulfilled only if* ***control panel*** works in collaboration with **sensor.**

Responsibilities

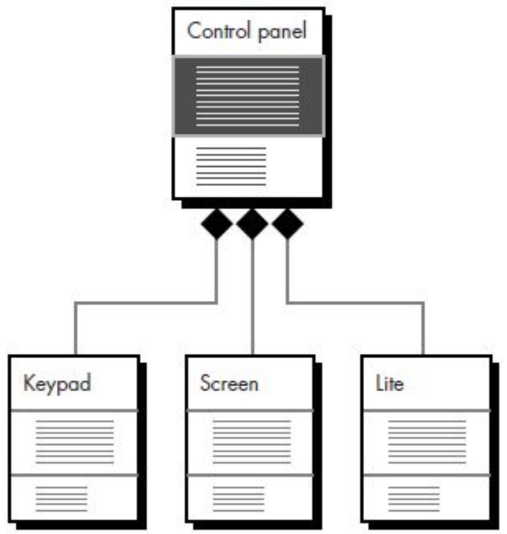
- To summarize, attributes represent stable features of a class; that is, information about the class that must be retained to accomplish the objectives of the software specified by the customer.
- Attributes can often be extracted from the statement of scope or discerned from an understanding of the nature of the class.
- Operations can be extracted by performing a grammatical parse on the processing narrative for the system. All verbs become candidate operations. Each operation that is chosen for a class exhibits a behavior of the class.
- Wirfs-Brock and her colleagues suggest five guidelines for allocating responsibilities to classes:
  1. System intelligence should be evenly distributed.
  2. Each responsibility should be stated as generally as possible.
  3. Information and the behavior related to it should reside within the same class.
  4. Information about one thing should be localized with a single class, not distributed across multiple classes.
  5. Responsibilities should be shared among related classes, when appropriate.

Defining Structures and Hierarchies

- Once classes and objects have been identified using the CRC model, the analyst begins to focus on the structure of the class model and the resultant hierarchies that arise as classes and subclasses emerge. Using UML notation, a variety of class diagrams can be created. ***Generalization/specialization*** class structures can be created for identified classes.
- In other cases, an object represented in the initial model might actually be composed of a number of component parts that could themselves be defined as objects.
- These aggregate objects can be represented as a ***composite aggregate*** *and* are defined using the notation represented in Figure. The diamond implies an **assembly relationship**. It should be noted that the connecting lines may be augmented with additional symbols (not shown) to represent cardinality.

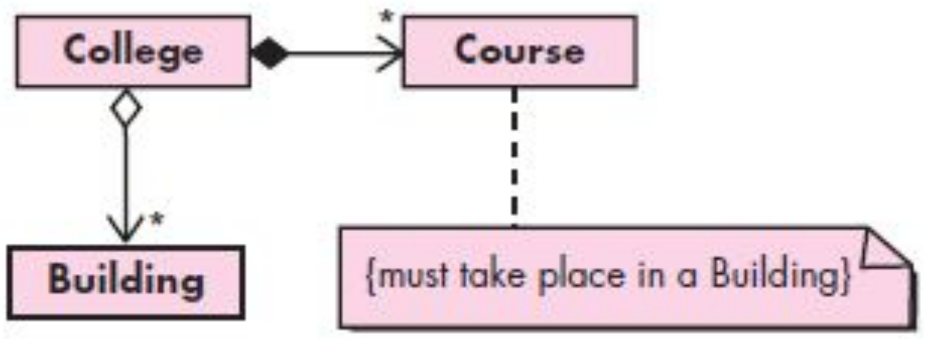


Class diagram for generalization/specialization



Class diagram for composite aggregates

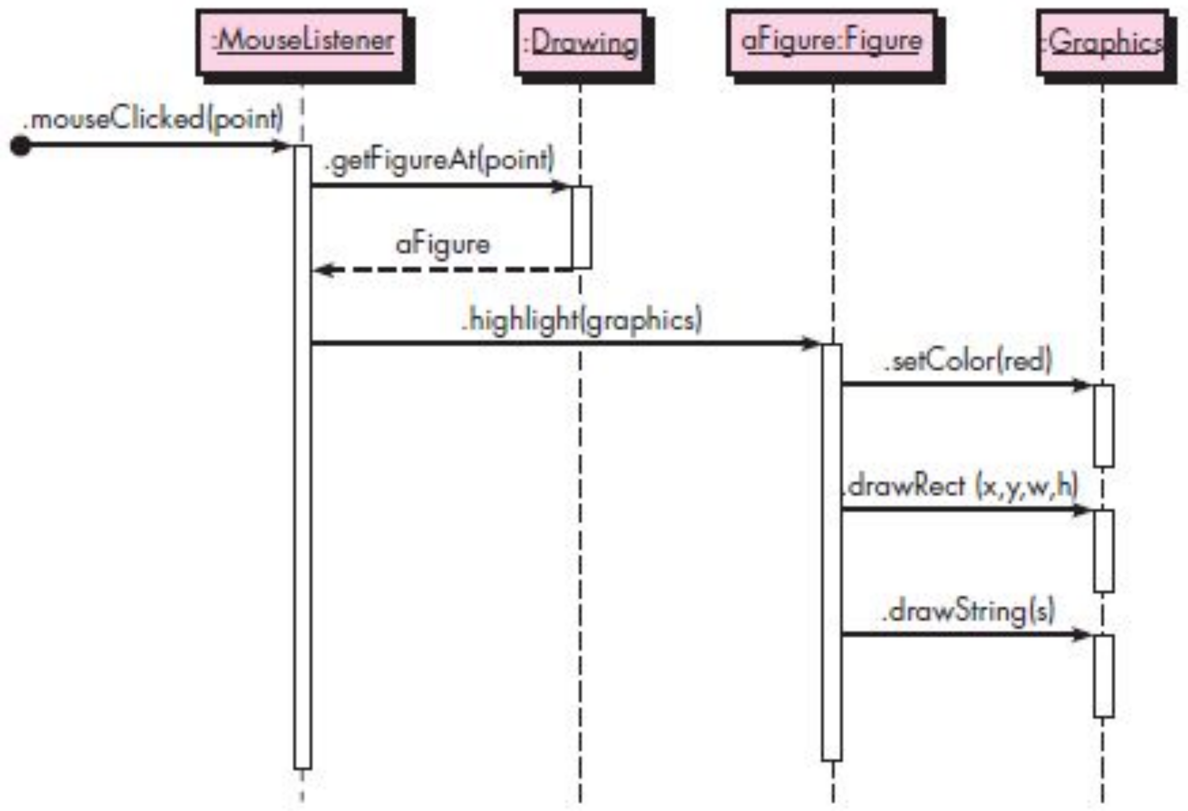
- **An *aggregation*** is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a “whole/part” relationship, in that the class to which the arrow points is considered a “part” of the class at the diamond end of the association.
- **A *composition*** is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner.
- A **College** has an aggregation of **Building** objects, which represent the buildings making up the campus. The college also has a collection of courses. If the college were to fold, the buildings would still exist (assuming the college wasn’t physically destroyed) and could be used for other things, but a **Course** object has no use outside of the college at which it is being offered. If the college were to cease to exist as a business entity, the **Course** object would no longer be useful and so it would also cease to exist.



The relationship between Colleges, Courses, and Buildings

SEQUENCE DIAGRAMS

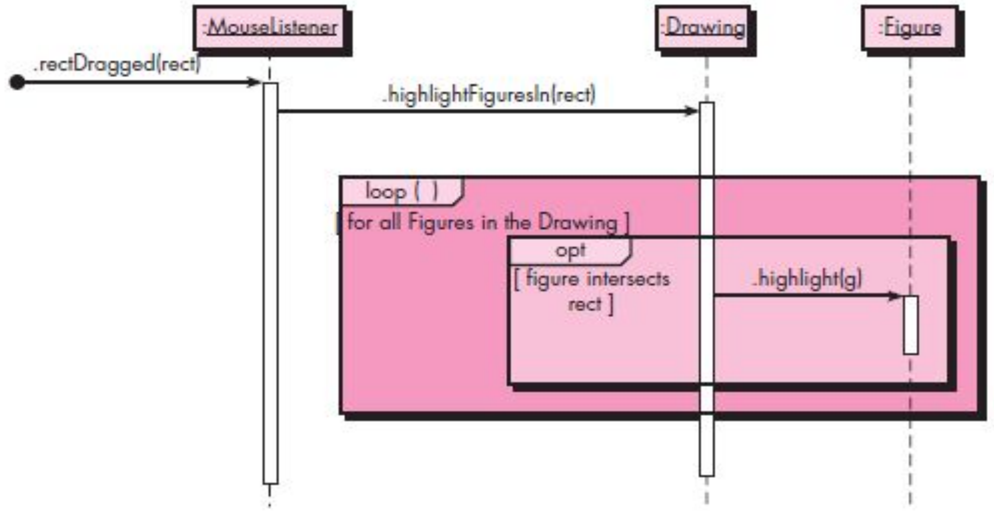
- A ***sequence diagram*** is used to show the dynamic communications between objects during execution of a task. It shows the temporal order in which messages are sent between the objects to accomplish that task. One might use a sequence diagram to show the interactions in one use case or in one scenario of a software system.
- The diagram shows, each box in the row at the top of the diagram usually corresponds to an object, although it is possible to have the boxes model other things, such as classes.
- If the box represents an object (as is the case in all our examples), then inside the box you can optionally state the type of the object preceded by the colon. You can also precede the colon and type by a name for the object, as shown in the third box in Figure.
- Below each box there is a dashed line called the ***lifeline*** of the object. The vertical axis in the sequence diagram corresponds to time, with time increasing as you move downward.
- A sequence diagram shows method calls using horizontal arrows from the ***caller*** to the ***callee****,* labeled with the method name and optionally including its parameters, their types, and the return type. For example, in Figure, the **MouseListener** calls the **Drawing's** ***getFigureAt()*** method. When an object is executing a method (that is, when it has an activation frame on the stack), you can optionally display a white bar, called an ***activation bar****,* down the object's **lifeline**.



A sample sequence diagram

- ❑ The diagram in above Figure is very straightforward and contains no conditionals or loops. If logical control structures are required, it is probably best to draw a separate sequence diagram for each case. That is, if the message flow can take two different paths depending on a condition, then draw two separate sequence diagrams, one for each possibility.

- If you insist on including loops, conditionals, and other control structures in a sequence diagram, you can use ***interaction frames***, which are rectangles that surround parts of the diagram and that are labeled with the type of control structures they represent.
- Figure illustrates this, showing the process involved in highlighting all figures inside a given rectangle.
- The **MouseListener** is sent the rectDragged message.
- The **MouseListener** then tells the drawing to highlight all figures in the **rectangle** by called the method ***highlightFigures()****,* passing the rectangle as the argument.
- The method loops through all **Figure** objects in the **Drawing** object and, if the **Figure** intersects the rectangle, the **Figure** is asked to highlight itself. The phrases in square brackets are called *guards,* which are Boolean conditions that must be true if the action inside the interaction frame is to continue.


:MouseListener
:Drawing
:Figure
.rectDragged(rect)
.highlightFiguresIn(rect)
loop ( )
[for all Figures in the Drawing]
opt
[figure intersects rect]
.highlight(g)

A sequence diagram with two interaction frames

- There are many other special features that can be included in a sequence diagram. For example:

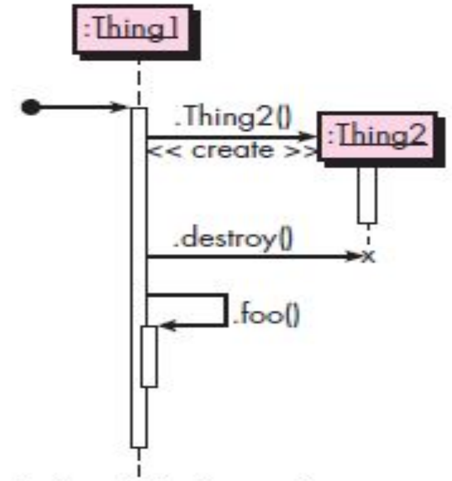
1. You can distinguish between synchronous and asynchronous messages.

Synchronous messages are shown with solid arrowheads while asynchronous messages are shown with stick arrowheads.

2. You can show an object sending itself a message with an arrow going out from the object, turning downward, and then pointing back to the same object.

3. You can show object creation by drawing an arrow appropriately labeled (for example, with a «create» label) to an object's box. In this case, the box will appear lower in the diagram than the boxes corresponding to objects already in existence when the action begins.

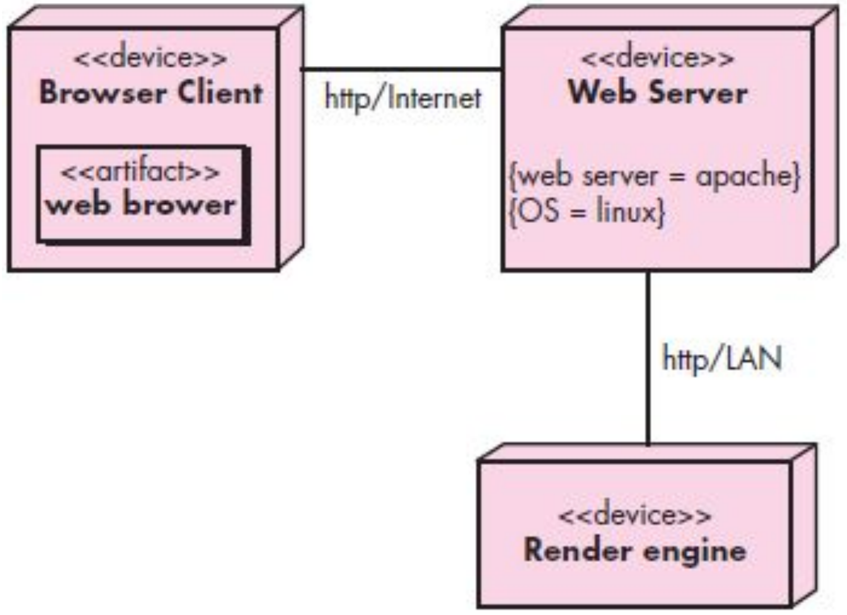
4. You can show object destruction by a big X at the end of the object's lifeline. Other objects can destroy an object, in which case an arrow points from the other object to the X. An X is also useful for indicating that an object is no longer usable and so is ready for garbage collection.



Creation, destruction, and loops in sequence diagrams

DEPLOYMENT DIAGRAMS

- A UML *deployment diagram* focuses on the structure of a software system and is useful for showing the physical distribution of a software system among hardware platforms and execution environments.
- Suppose, for example, you are developing a Web-based graphics-rendering package. Users of your package will use their Web browser to go to your website and enter rendering information.

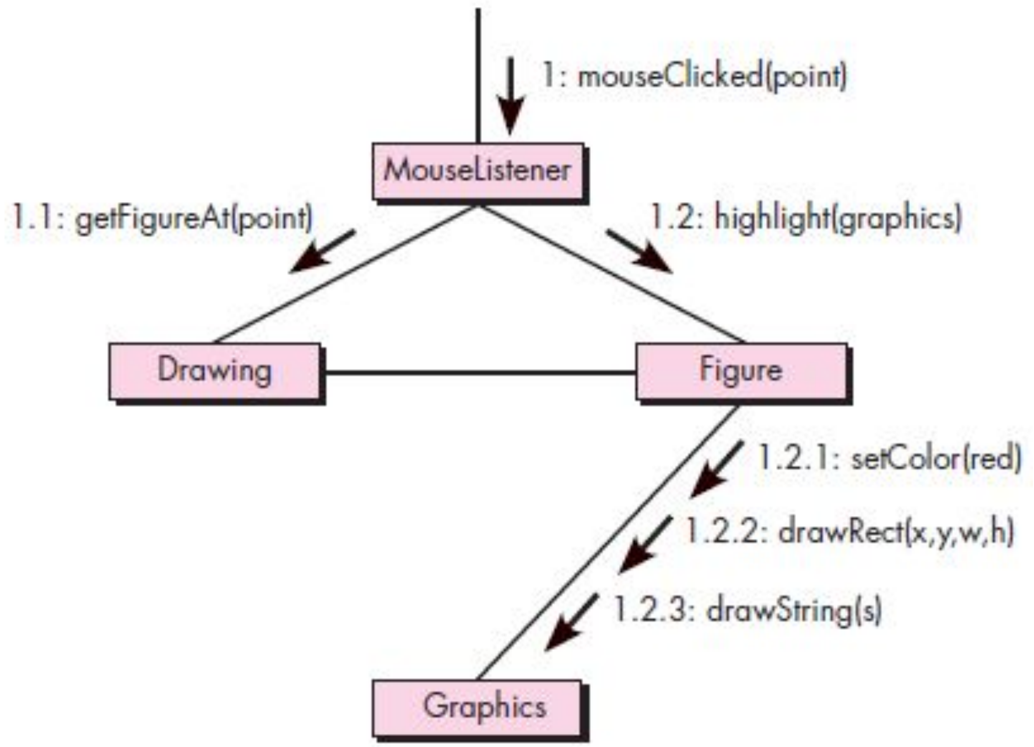


A deployment diagram

- Figure shows the deployment diagram for such a package. In such a diagram, hardware components are drawn in boxes labeled with “«device»”. Communication paths between hardware components are drawn with lines with optional labels. In Figure, the paths are labeled with the communication protocol and the type of network used to connect the devices.
- Each node in a deployment diagram can also be annotated with details about the device. For example, in Figure, the browser client is depicted to show that it contains an artifact consisting of the Web browser software. An artifact is typically a file containing software running on a device. You can also specify tagged values, as is shown in Figure in the Web server node. These values define the vendor of the Web server and the operating system used by the server.
- Deployment diagrams can also display execution environment nodes, which are drawn as boxes containing the label “«execution environment»”. These nodes represent systems, such as operating systems, that can host other software.

COMMUNICATION DIAGRAMS

The UML *communication diagram (called a **"collaboration diagram"** in UML 1.X) provides* another indication of the temporal order of the communications but emphasizes the relationships among the objects and classes instead of the temporal order.



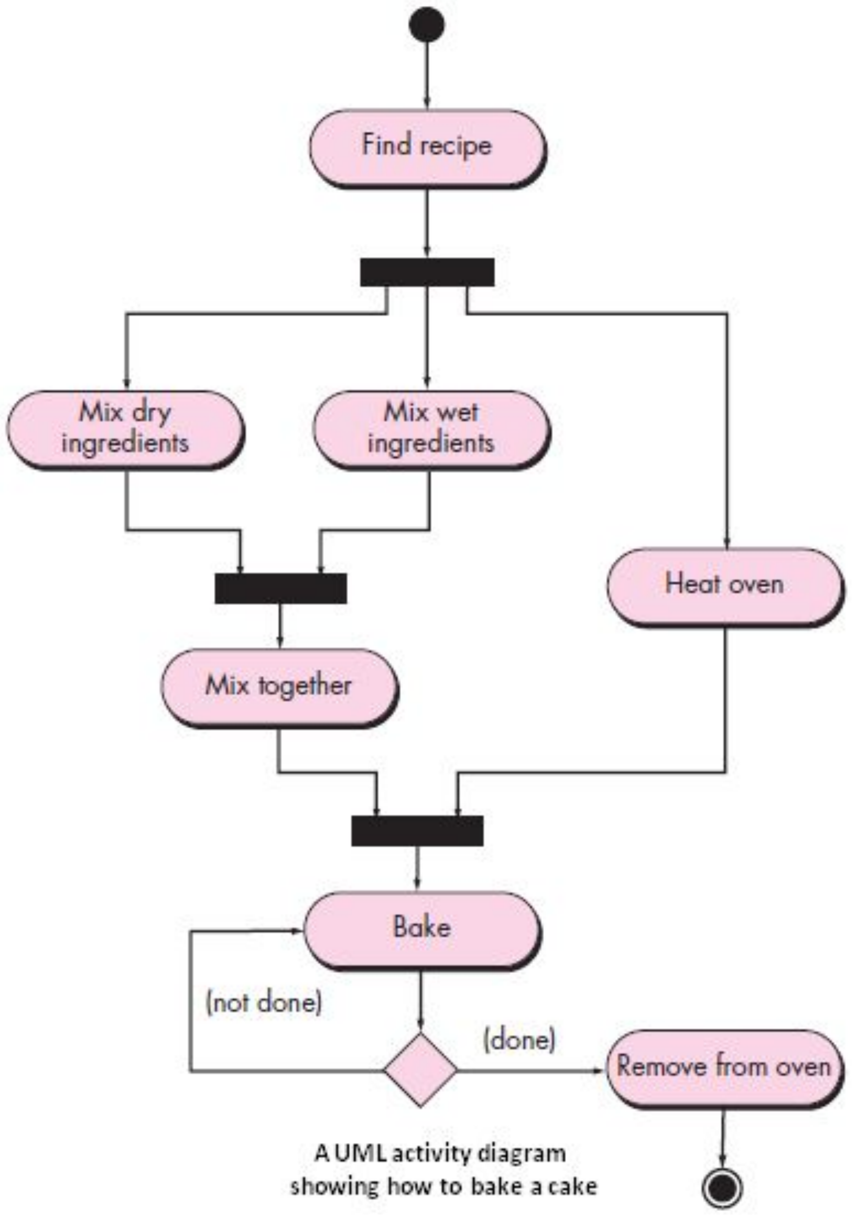
A UML communication diagram

A communication diagram, illustrated in Figure, displays the same actions shown in the sequence diagram in Figure.

- In a communication diagram the interacting objects are represented by rectangles. Associations between objects are represented by lines connecting the rectangles.
- There is typically an incoming arrow to one object in the diagram that starts the sequence of message passing. **That arrow is labeled with a number and a message name.**
- If the incoming message is labeled with the number 1 and if it causes the receiving object to invoke other messages on other objects, then those messages are represented by arrows from the sender to the receiver along an association line and are given numbers 1.1, 1.2, and so forth, in the order they are called.
- If those messages in turn invoke other messages, another decimal point and number are added to the number labeling these messages, to indicate further nesting of the message passing.

ACTIVITY DIAGRAMS

- ❑ A UML ***activity diagram*** depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows.
- ❑ The main component of an activity diagram is an ***action node****,* represented by a rounded rectangle, which corresponds to a task performed by the software system.
- ❑ **Arrows** from one action node to another indicate the flow of control. That is, an arrow between two action nodes means that after the first action is complete the second action begins.
- ❑ A solid black dot forms the ***initial node*** that indicates the starting point of the activity. A black dot surrounded by a black circle is the ***final node*** indicating the end of the activity.
- ❑ Figure shows a sample activity diagram involving baking a cake. The first step is finding the recipe. Once the recipe has been found, the dry ingredients and wet ingredients can be measured and mixed and the oven can be preheated. The mixing of the dry ingredients can be done in parallel with the mixing of the wet ingredients and the preheating of the oven.

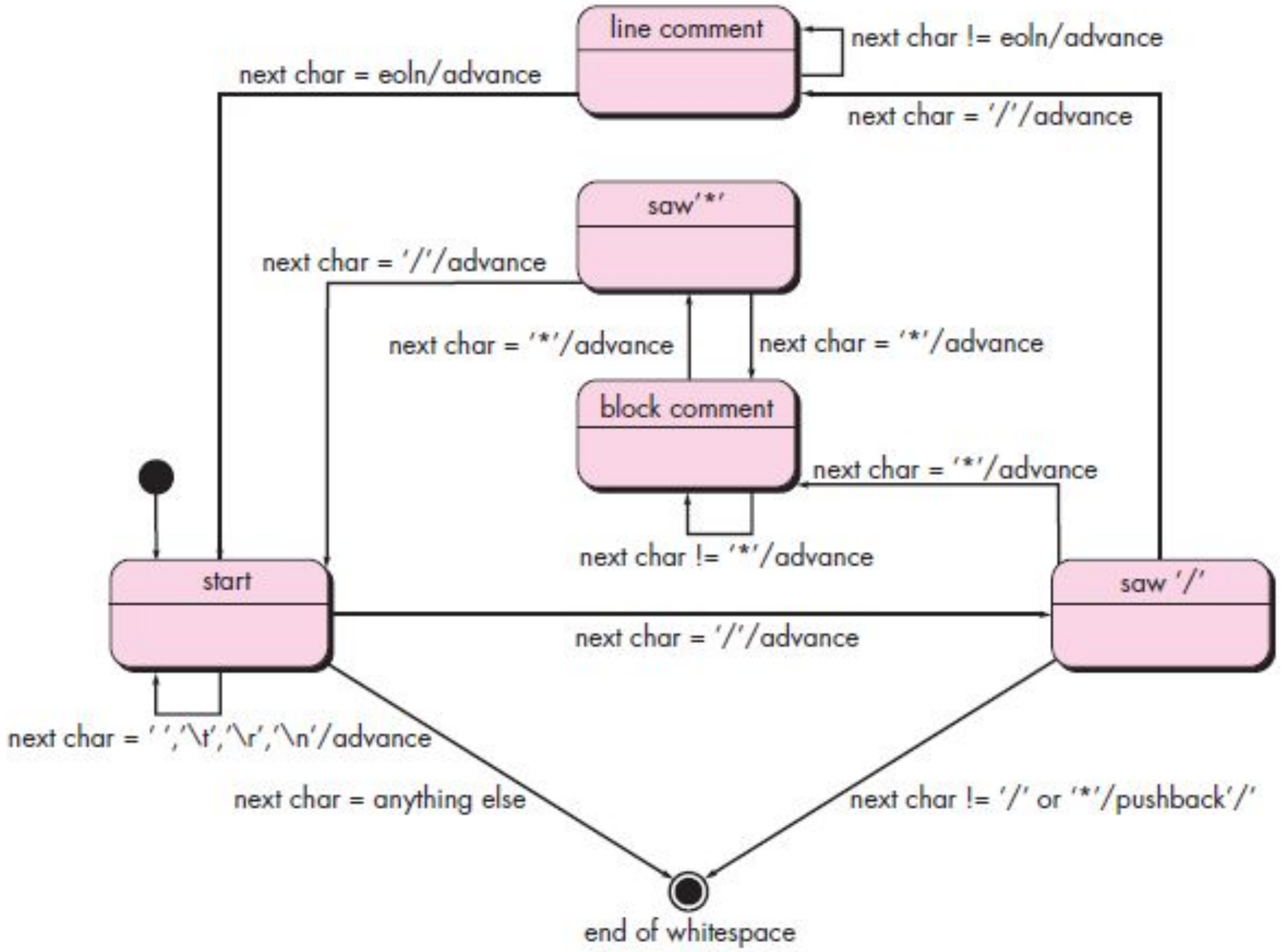


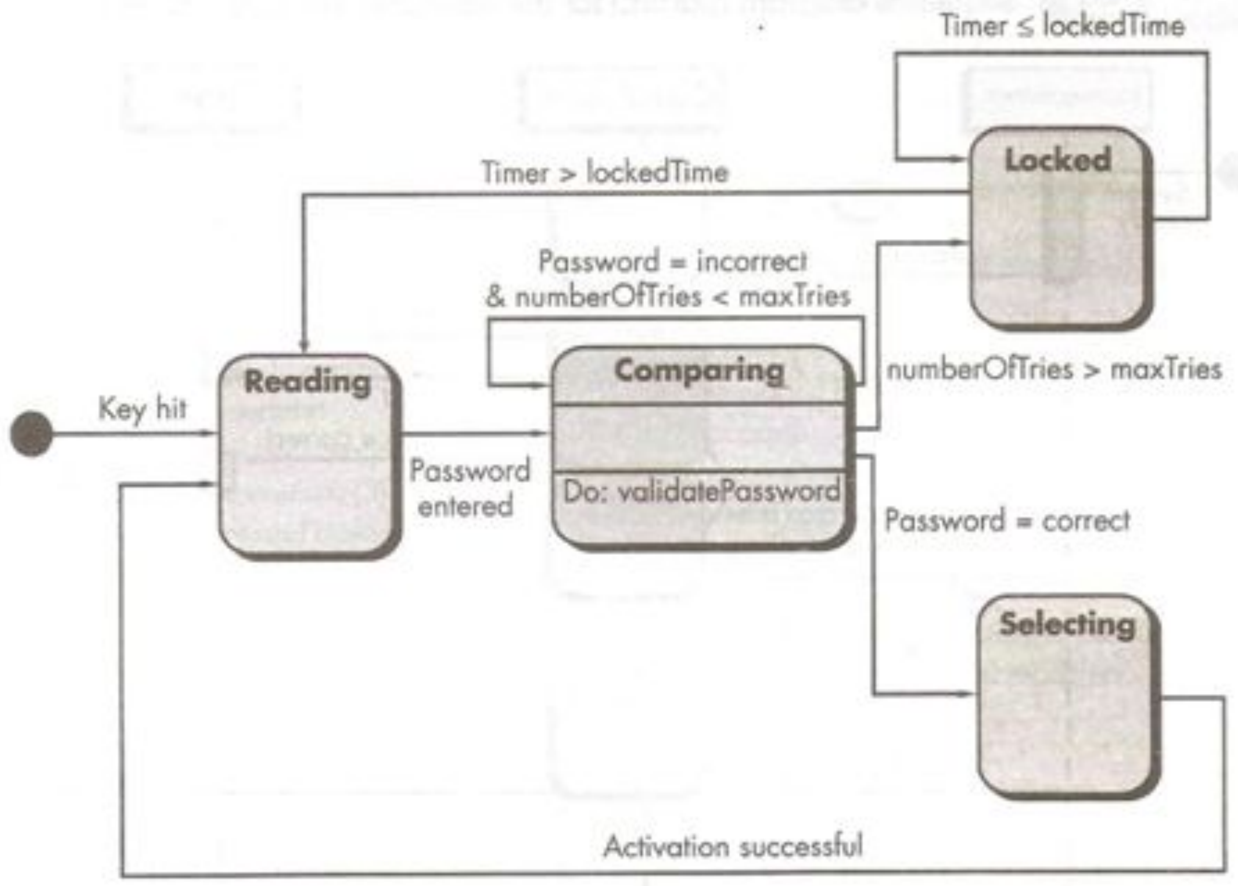
A UML activity diagram showing how to bake a cake

STATE DIAGRAMS

- A UML ***state diagram*** models an object's states, the actions that are performed depending on those states, and the transitions between the states of the object.
- As an example, consider the state diagram for a part of a Java compiler. The input to the compiler is a text file, which can be thought of as a long string of characters. The compiler reads characters one at a time and from them determines the structure of the program.
- One small part of this process of reading the characters involves ignoring "white-space" characters (e.g., the *space, tab, newline, and return characters)* and characters inside a comment.

A state diagram for advancing past white space and comments in Java





State diagram for the Control Panel Class

Agile Development: About Agility, Agility and cost of change, Agile process, Agile process models (Adaptive software development, Scrum, Dynamic system development method), Agile Software development Approaches

What is agility?

- ✔ ***Agility has become today's buzzword when describing a modern software process. Everyone* is agile.**
- ✔ **An agile team is able to appropriately respond to changes.**
- ✔ **What is Change?**
- ✔ **Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product.**
- ✔ **An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.**
- ✔ **In Jacobson's view, the pervasiveness of change is the primary driver for agility. Software engineers must be quick on their feet if they are to accommodate the rapid changes that Jacobson describes.**
- ✔ **But agility is more than an effective response to change.**
- ✔ **It encourages team structures and attitudes that make communication (among team members, between technologists and business people, between software engineers and their managers) more facile.**
- ✔ **It emphasizes rapid delivery of operational software and de-emphasizes the importance of intermediate work products (not always a good thing);**
- ✔ **It adopts the customer as a part of the development team and works to eliminate the "us and them" attitude that continues to pervade many software projects;**
- ✔ **It recognizes that planning in an uncertain world has its limits and that a project plan must be flexible.**

Agility Principles

- The Agile Alliance defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

Agility Principles

8. **Agile processes promote sustainable development. The sponsors, developers,** and users should be able to maintain a constant pace indefinitely.

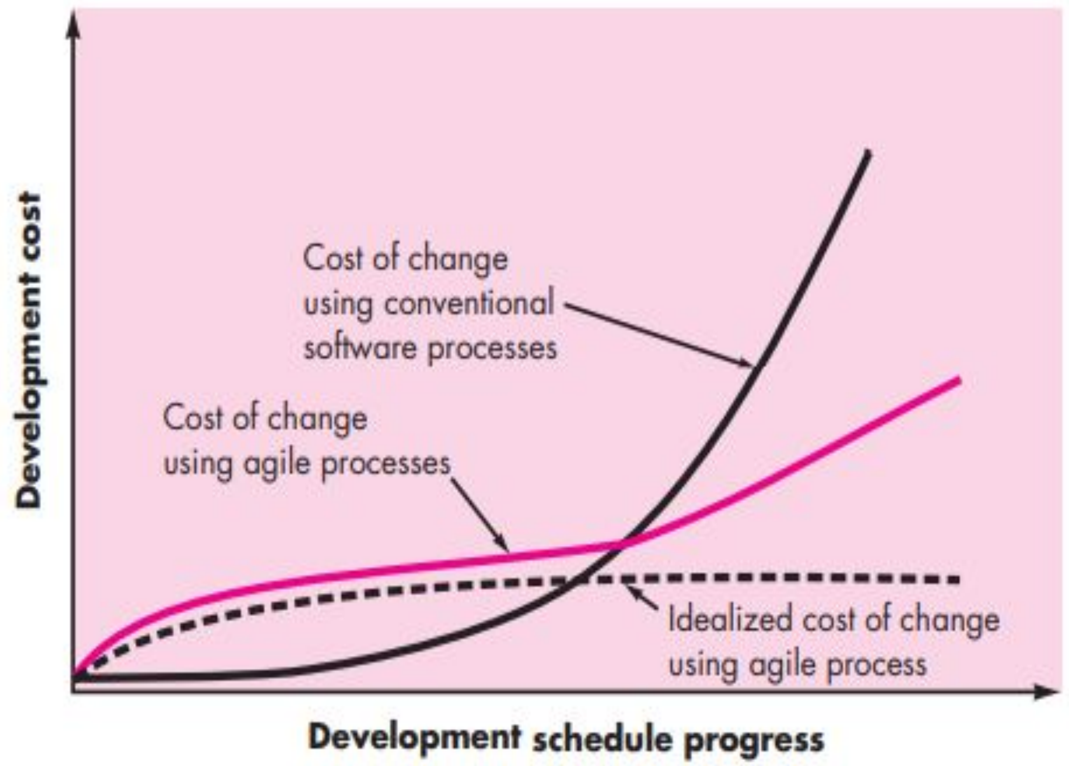
9. **Continuous attention to technical excellence and good design enhances** agility.

10. **Simplicity—the art of maximizing the amount of work not done—is** essential.

11. **The best architectures, requirements, and designs emerge from self–** organizing teams.

12. **At regular intervals, the team reflects on how to become more effective, then** tunes and adjusts its behavior accordingly.

Change cost as a function of time in development



What is an agile process?

- Any agile software process is characterized in a manner that addresses a number of key assumptions [Fow02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change.

- It is equally difficult to predict how customer priorities will change as the project proceeds.

2. For many types of software, design and construction are interleaved.

- **That is,** both activities should be performed in tandem(aggregation) so that design models are proven as they are created.
- It is difficult to predict how much design is necessary before construction is used to prove the design.

3. Analysis, design, construction, and testing are not as predictable (from a planning point of view).

- An agile process, therefore, must be *adaptable.*

Human Factors

- Proponents(supporter) of agile software development take great pains to emphasize the importance of "people factors."
- As Cockburn and Highsmith [Coc01a] state,
  - "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams."

- **following key traits must exist among the people on an agile team and the team itself:**

- **Competence**
- **Common focus**
- **Collaboration**
- **Decision-making ability**
- **Fuzzy problem-solving ability**
- **Mutual trust and respect**
- **Self-organization**

Human Factors

 Competence

- **In an agile development (as well as software engineering)** context, “competence” encompasses innate(inborn) talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply.
- Skill and knowledge of process can and should be taught to all people who serve as agile team members.

 Common focus

- **Although members of the agile team may perform different** tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised.
- To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Human Factors

Collaboration

- **Software engineering (regardless of process) is about assessing,** analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer.
- To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability

- **Any good software team (including agile teams)** must be allowed the freedom to control its own destiny.
- This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Human Factors

 Fuzzy problem-solving ability

- **Software managers must recognize that** the agile team will continually have to deal with ambiguity and will continually be buffeted(thrashed) by change.
- In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow.
- However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

 Mutual trust and respect

- **The agile team must become what DeMarco** and Lister [DeM98] call a “jelled” team .
- A jelled(finalized) team exhibits the trust and respect that are necessary to make them “so strongly knit(bind) that the whole is greater than the sum of the parts.” [DeM98]

Human Factors

Self-organization

In the context of agile development, self-organization implies three things:

(1) the agile team organizes itself for the work to be done,

(2) the team organizes the process to best accommodate its local environment,

(3) the team organizes the work schedule to best achieve delivery of the software increment.

Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale(assurance).

In essence, the team serves as its own management. Ken Schwaber [Sch02] addresses these issues when he writes: “The team selects how much work it believes it can perform within the iteration, and the team commits to the work.

Nothing de-motivates a team as much as someone else making commitments for it.

Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself.”

Adaptive Software Development (ASD)

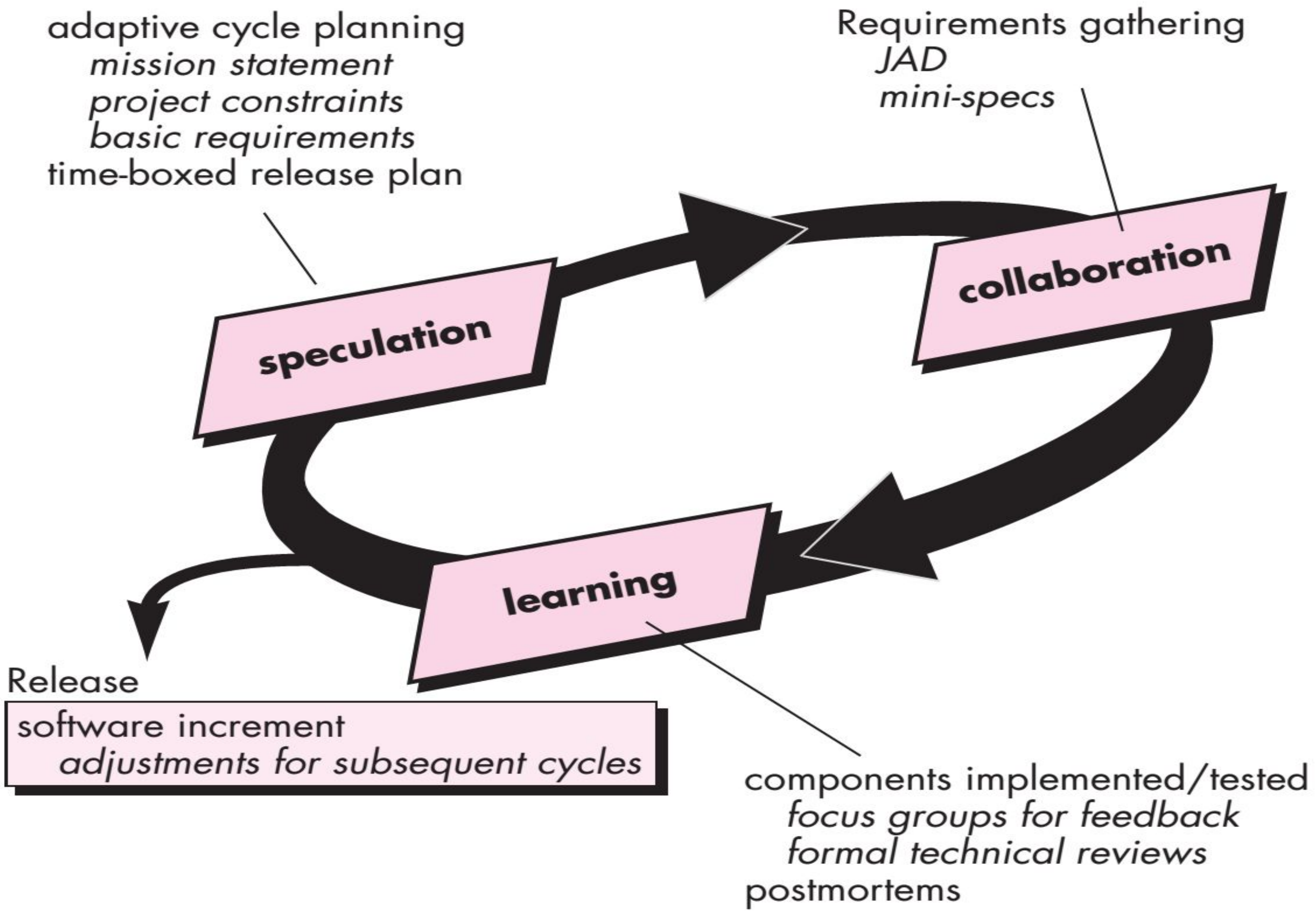
- *Adaptive Software Development* (ASD) has been proposed by Jim Highsmith [Hig00] as a technique for building complex software and systems.

- The philosophical foundation of ASD focus on human collaboration and team self-organization.

- Highsmith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering."

- He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.

Adaptive Software Development (ASD)



Adaptive Software Development (ASD)

- **Speculation:**
- During *speculation,* the project is initiated and *adaptive cycle planning* is conducted.
- Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.
- No matter how complete and farsighted(clear sighted/wise) the cycle plan, it will invariably change.
- Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working.

Adaptive Software Development (ASD)

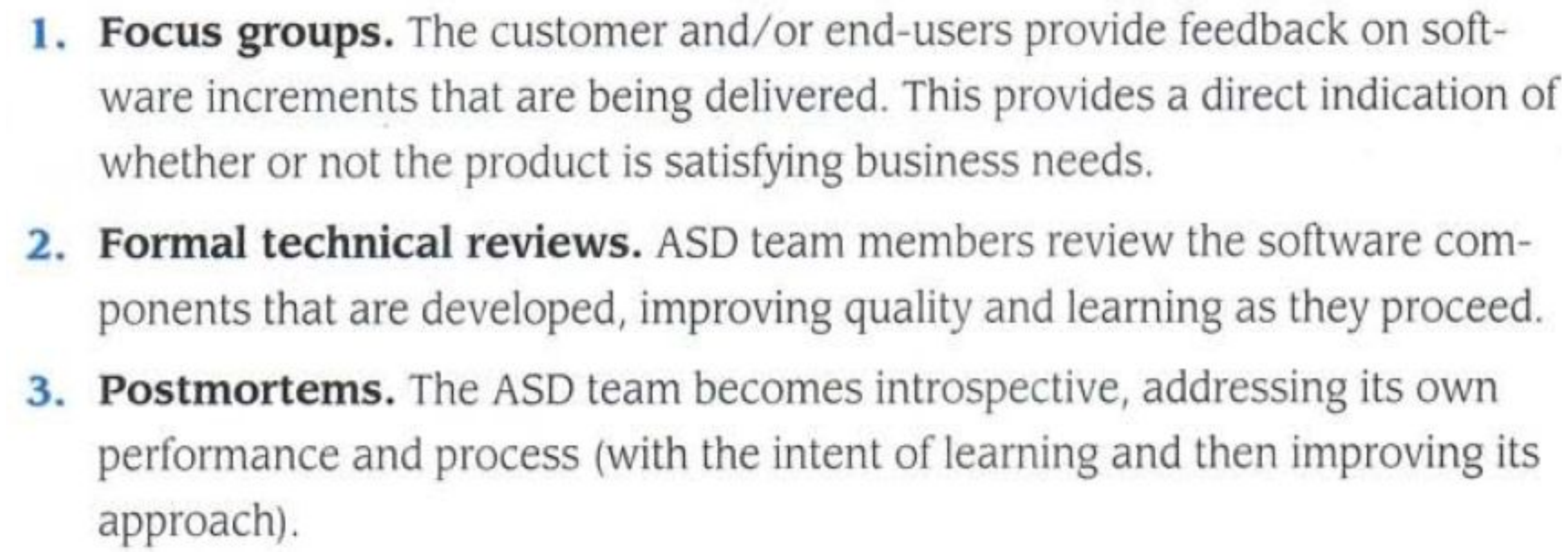
- **Collaboration:**
- Motivated people use *collaboration* in a way that multiplies their talent and creative output beyond their absolute numbers.
- This approach is a recurring theme in all agile methods.
- But collaboration is not easy.
- It involves communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking.
- It is, above all, a matter of trust.
- People working together must trust one another to (1) criticize without animosity (anger), (2) assist without resentment(hate), (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

Adaptive Software Development (ASD)

- **Learning:**
- As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on “learning” as much as it is on progress toward a completed cycle.
- In fact, Highsmith [Hig00] argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding.

Adaptive Software Development (ASD)

- ASD teams learn in three ways: focus groups, technical reviews, and project post-mortems.



1. **Focus groups.** The customer and/or end-users provide feedback on software increments that are being delivered. This provides a direct indication of whether or not the product is satisfying business needs.
2. **Formal technical reviews.** ASD team members review the software components that are developed, improving quality and learning as they proceed.
3. **Postmortems.** The ASD team becomes introspective, addressing its own performance and process (with the intent of learning and then improving its approach).

- The ASD philosophy has merit regardless of the process model that is used.
- ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

Scrum

- Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s.
- Scrum principles are consistent with the agile manifesto:
  1. Small working teams are organized to maximize communication, minimize overhead and maximize sharing of tacit(implicit), informal knowledge.
  2. The process must be adaptable to both technical and business changes to ensure the best possible product is produced.
  3. Process yields frequent software increments that can be inspected, adjusted, tested, documented and built on.
  4. Development work and people who perform it are partitioned into clean low coupling partitions or packets.
  5. Constant testing and documentation is performed as the product is built.
  6. The scrum process provides the ability to declare a product done whenever required.

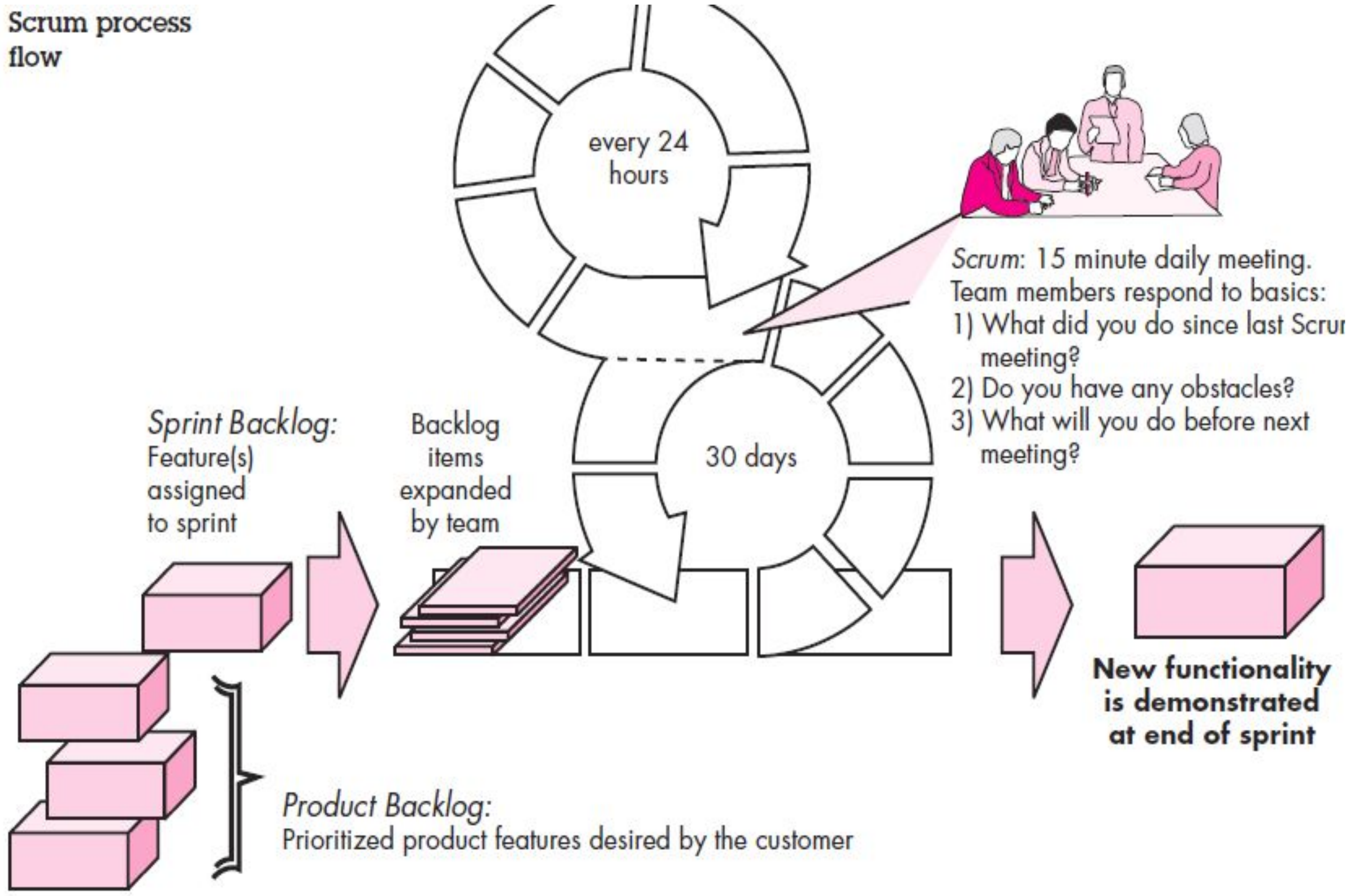
Scrum

- Scrum principles are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery.

- Within each framework activity, work tasks occur within a process pattern called a *sprint.*

- *The work conducted within a sprint (the number* of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.
- The overall flow of the Scrum process is illustrated in Figure.
- Scrum emphasizes the use of a set of software process patterns [Noy02] that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

- *Backlog—a prioritized list of project requirements or features that provide business* value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.


Scrum process flow
every 24 hours
Scrum: 15 minute daily meeting.
Team members respond to basics:
1) What did you do since last Scrur meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
30 days
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

Scrum

- *Sprints—consist of work units that are required to achieve a requirement defined* in the backlog that must be fit into a predefined time-box (typically 30 days).
- Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.
- *Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum* team. Three key questions are asked and answered by all team members [Noy02]:
- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?
- A team leader, called a *Scrum master, leads the meeting and assesses the responses* from each person. The Scrum meeting helps the team to uncover potential problems as early as possible.
- Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure.

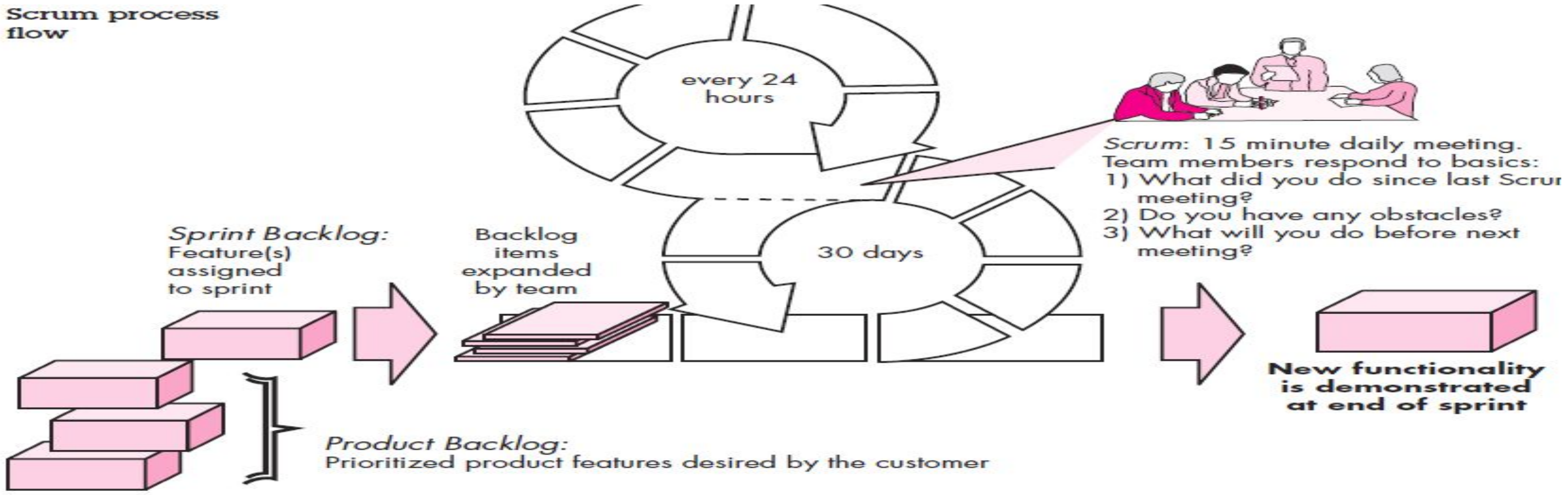
Scrum

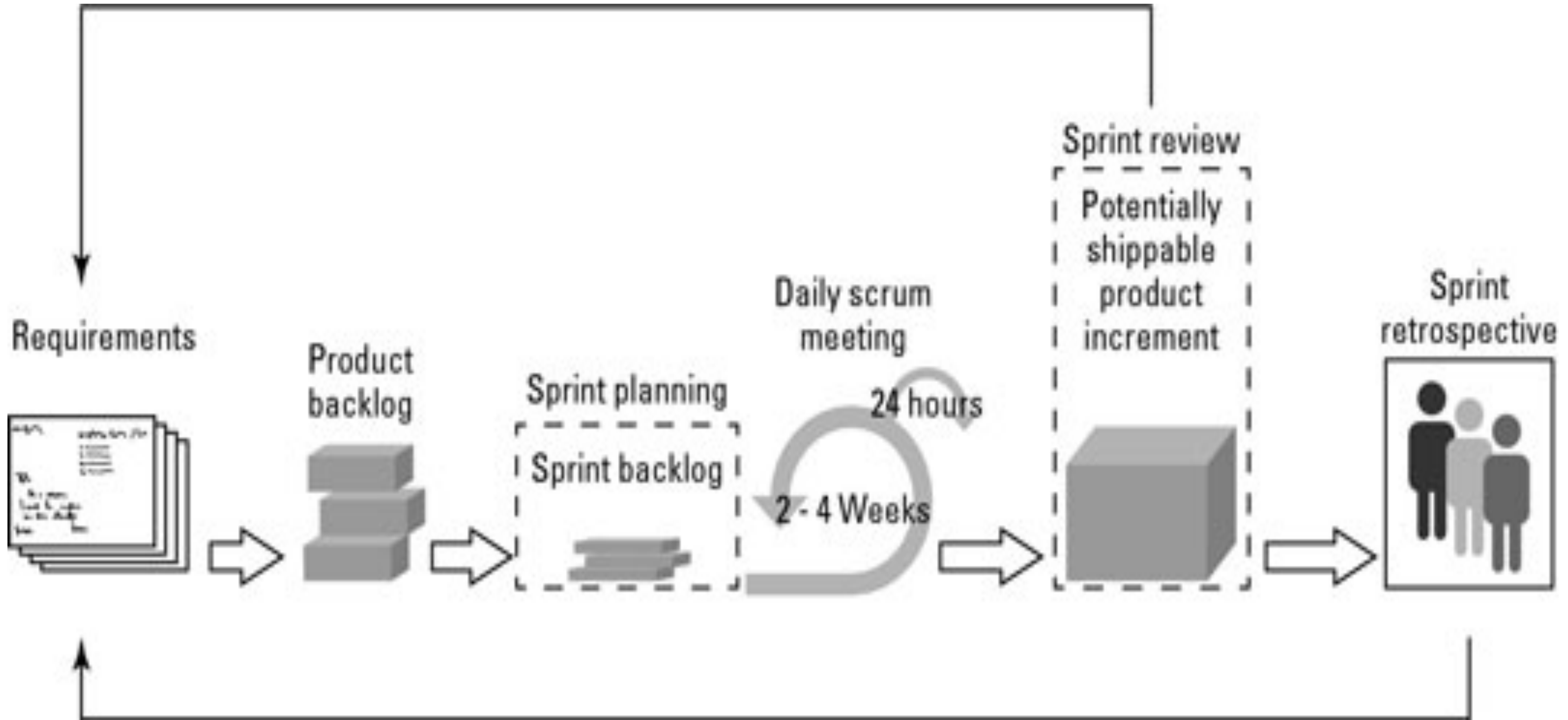
- *Demos— deliver the software increment to the customer so that functionality that* has been implemented can be demonstrated and evaluated by the customer.

- It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

- Beedle and his colleagues [Bee99] present a comprehensive(complete) discussion of these patterns in which they state: "Scrum assumes up-front the existence of chaos(confusion). . . . " The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible.

- The Scrum team should continue to follow the best practice, ignore the "not best practices" and implement the lessons learned during the consequent sprints. The retrospective meeting helps to implement the continuous improvement of the SCRUM process.


Scrum process flow
every 24 hours
30 days
Scrum: 15 minute daily meeting.
Team members respond to basics:
1) What did you do since last Scrur meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer


Requirements
Product backlog
Sprint planning
Sprint backlog
Daily scrum meeting
24 hours
2 - 4 Weeks
Sprint review
Potentially shippable product increment
Sprint retrospective

Dynamic Systems Development Method (DSDM)

- The *Dynamic Systems Development Method* (DSDM) [Sta97] is an agile software development approach that **"provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment"** CCS02].

- The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.
- DSDM is an iterative software process in which each iteration follows the 80 percent rule.

- That is, only enough work is required for each increment to facilitate movement to the next increment.

- The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

Dynamic Systems Development Method (DSDM)

- The DSDM Consortium (**www.dsdm.org**) is a worldwide group of member companies that collectively take on the role of "keeper" of the method.
- The consortium has defined an agile process model, called the *DSDM life cycle* that defines three different iterative cycles, preceded by two additional life cycle activities:
- *Feasibility study*—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.
- *Business study*—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.
- *Functional model iteration*—produces a set of incremental prototypes that demonstrate functionality for the customer.
- The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Dynamic Systems Development Method (DSDM)

- *Design and build iteration*—revisits prototypes built during *functional model iteration* to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users.
- In some cases, *functional model iteration* and *design and build iteration* occur concurrently.
- *Implementation*—places the latest software increment (an "operationalized" prototype) into the operational environment.
- It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place.
- In either case, DSDM development work continues by returning to the functional model iteration activity.
- DSDM can be combined with XP (Section 3.4) to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments.
- In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

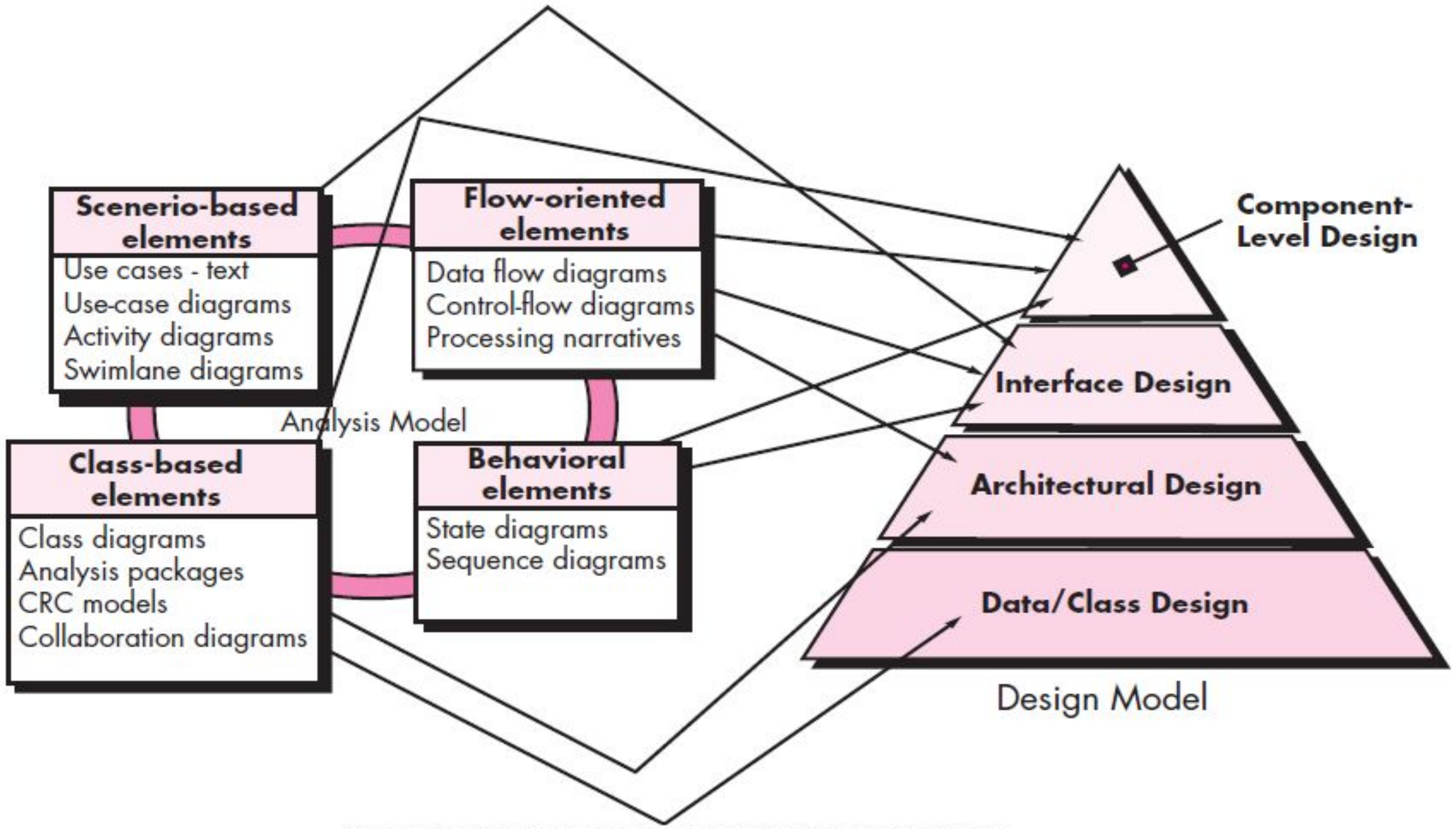
Software Design Engineering: The design process and fundamentals, Effective modular Design, Data flow oriented design, Transform analysis, Transaction analysis, Design heuristics.

SOFTWARE DESIGN CONCEPTS

- ❑ Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- ❑ Design principles establish an overriding philosophy that guides you in the design work you must perform.
- ❑ Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, *software design is the last software engineering action within the modeling activity* and sets the stage for **construction** (*code generation and testing*).



Translating the requirements model into the design model

- Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design.
- **The data/class design transforms class models into design class realizations** and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action.
- **The architectural design defines the relationship between major structural elements** of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.
- **The interface design describes how the software communicates with systems that interoperate** with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior.

- **The component-level design transforms structural elements of the software architecture into a procedural description** of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.
- The importance of software design can be stated with a single word—***quality***. Design is the place where quality is fostered in software engineering.
- Design provides you with representations of software that can be assessed for quality.
- Design is the only way that you can accurately translate stakeholder’s requirements into a finished software product or system.
- Software design serves as the foundation for all the software engineering and software support activities that follow.

Software Quality Guidelines and Attributes

❑ Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. **McGlaughlin** *suggests three characteristics that serve as a guide for the evaluation of a good design:*

- **The design must implement all of the explicit requirements** contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- **The design must be a readable, understandable guide** for those who generate code and for those who test and subsequently support the software.
- **The design should provide a complete picture of the software**, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design.

For software quality criteria, consider the following guidelines:

1. **A design should exhibit an architecture** that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics, and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. **A design should be modular**; that is, the software should be logically partitioned into elements or subsystems.
3. **A design should contain distinct representations** of data, architecture, interfaces, and components. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

4. **A design should lead to components** that exhibit independent functional characteristics.
5. **A design should lead to interfaces** that reduce the complexity of connections between components and with the external environment.
6. **A design should be derived using a repeatable method** that is driven by information obtained during software requirements analysis.
7. **A design should be represented using a notation** that effectively communicates its meaning.

Quality Attributes

Hewlett-Packard developed a set of software quality attributes that has been given the acronym **FURPS**—*functionality, usability, reliability, performance,* and *supportability*.

The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- ***Supportability*** combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.

DESIGN CONCEPTS

Important software design concepts that span both traditional and object-oriented software development.

1. Abstraction
2. Architecture
3. Patterns
4. Separation of Concerns
5. Modularity
6. Information Hiding
7. Functional Independence
8. Refinement
9. Aspects
10. Refactoring

Abstraction:

- ❖ At the highest level of abstraction, *a solution is stated in broad terms* using the language of the problem environment. At lower levels of abstraction, *a more detailed description of the solution is provided*. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution.
- ❖ A ***procedural abstraction*** refers to *a sequence of instructions* that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.
- ❖ An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (*e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).*
- ❖ A ***data abstraction*** is a *named collection of data that describes a data object*. In the context of the procedural abstraction *open,* we can define a data abstraction called **door.**
- ❖ Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (*e.g., door type, swing direction, opening mechanism, weight, dimensions*).
- ❖ It follows that the *procedural abstraction* ***open*** would make use of information contained in the attributes of the *data abstraction* **door.**

Architecture:

- ❖ Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- ❖ One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted.
- ❖ A set of architectural patterns enables a software engineer to solve common design problems.
- ❖ *Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:*

  Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

  Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

  Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

- The architectural design can be represented using one or more of a number of different models.
- ***Structural models*** represent architecture as an organized collection of program components.
- ***Framework models*** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
- ***Dynamic models*** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- ***Process models*** focus on the design of the business or technical process that the system must accommodate.
- ***Functional models*** can be used to represent the functional hierarchy of a system.
- A number of different ***architectural description languages*** *(ADLs) have been developed* to represent these models.
- Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

Patterns:

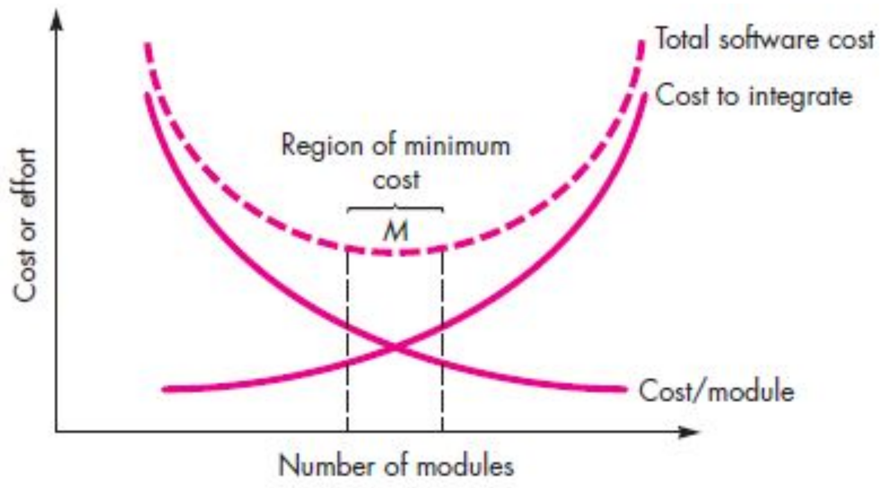
- ❖ A design pattern describes a design structure that solves a particular design problem within a specific context and amid “forces” that may have an impact on the manner in which the pattern is applied and used.
- ❖ The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns:

- ❖ ***Separation of concerns*** is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- ❖ A ***concern*** is a feature or behavior that is specified as part of the requirements model for the software.
- ❖ By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- ❖ For example two problems, *p1 and p2,* if the perceived complexity of *p1* is greater than the perceived complexity of *p2,* it follows that the effort required to solve *p1* is greater than the effort required to solve *p2.* As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

Modularity:

- Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called ***modules**,* that are integrated to satisfy problem requirements.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.
- It is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.



Modularity and software cost

- ❖ Given the same set of requirements, more modules means smaller individual size.
- ❖ However, *as the number of modules grows, the effort (cost) associated with integrating the modules also grows*. These characteristics lead to a total cost or effort curve shown in the figure.
- ❖ There is a number, *M,* of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.
- ❖ The curves shown in Figure do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of *M.* Undermodularity or overmodularity should be avoided.

Information Hiding:

- ❖ The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others."
- ❖ In other words, *modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information*.
- ❖ Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- ❖ Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.
- ❖ The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Effective Modular Design

Functional Independence:

- ❖ The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
- ❖ Functional independence is achieved by developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules.
- ❖ Stated another way, *you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure*.
- ❖ Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified.
- ❖ Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, **functional independence is a key to good design, and design is the key to software quality**.

Effective Modular Design …

- ❖ Independence is assessed using two qualitative criteria: *cohesion* and *coupling*.
- ❖ ***Cohesion*** is an indication of the relative functional strength of a module. ***Coupling*** is an indication of the relative interdependence among modules.
- ❖ **Cohesion** is a natural extension of the information-hiding.
- ❖ A cohesive module performs a single task, requiring little interaction with other components in other parts of a program.
- ❖ Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions.
- ❖ However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Effective Modular Design …

- **Coupling** is an indication of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect”, caused when errors occur at one location and propagate throughout a system.

Refinement:

- Stepwise refinement is a top-down design strategy originally proposed by **Niklaus Wirth**. A program is developed by successively refining levels of procedural detail.
- A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
- Refinement is actually a process of *elaboration.* You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.
- You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts.
- ***Abstraction*** enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details.
- ***Refinement*** helps you to reveal low-level details as design progresses.
- Both concepts allow you to create a complete design model as the design evolves.

Aspects:

- ❖ As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts".
- ❖ Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently.
- ❖ In practice, some of these concerns span the entire system and cannot be easily compartmentalized.
- ❖ As design begins, requirements are refined into a modular design representation. Consider two requirements, *A and B.* Requirement *A* crosscuts requirement *B "if a* software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account*".*

- An *aspect* is a representation of a crosscutting concern. Therefore, the design representation, B*, of the requirement a registered user must be validated prior *to using* ***SafeHomeAssured.com***, is an aspect of the SafeHome WebApp.
- It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur.
- In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are “scattered” or “tangled” throughout many components.
- To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

Refactoring:

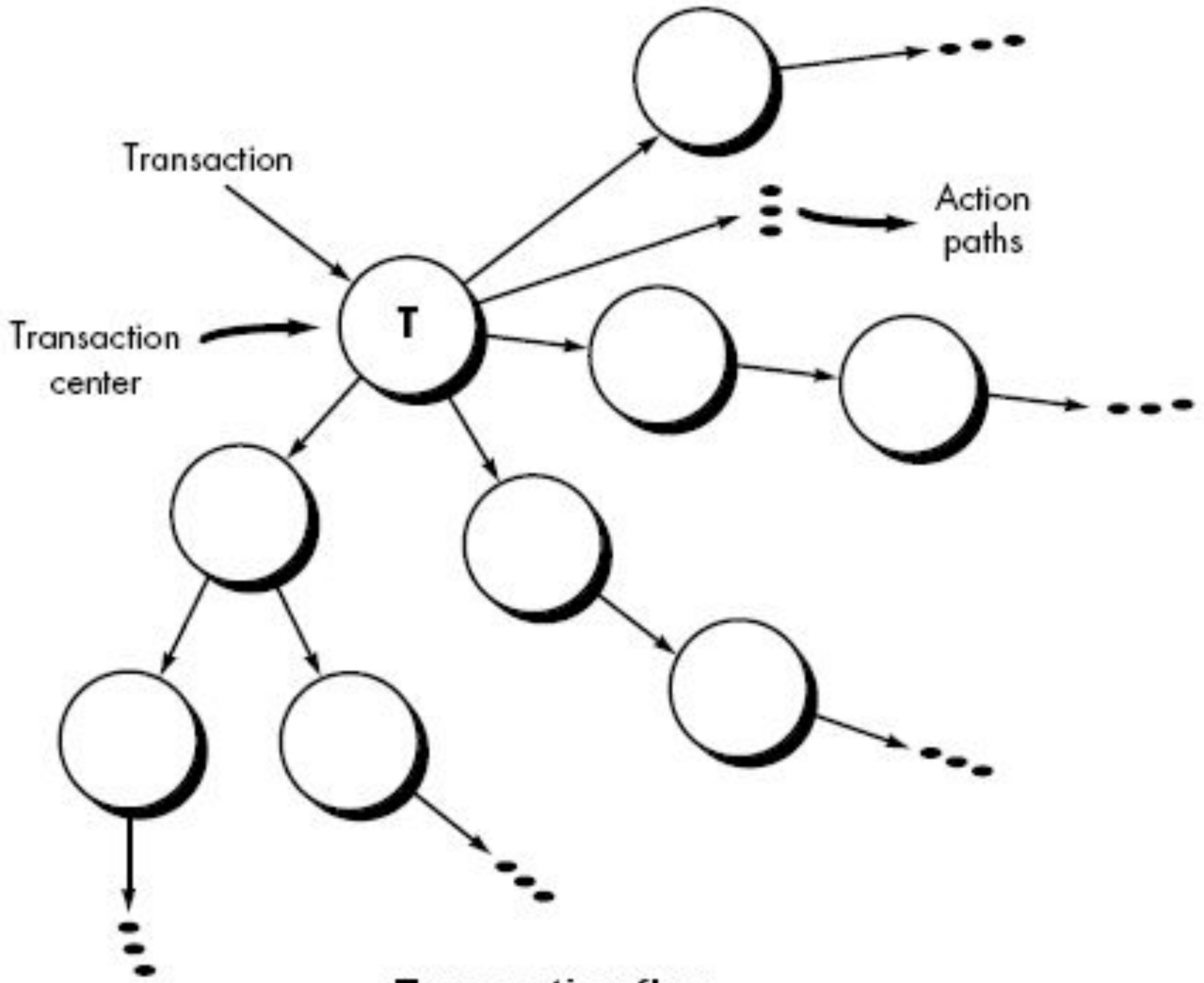
- ❖ An important design activity suggested for many agile methods, ***refactoring*** *is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior*.
- ❖ *Fowler* defines refactoring in the following manner: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”
- ❖ When software is refactored, the existing design is examined for redundancy, *unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected* to yield a better design.
- ❖ For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another).
- ❖ After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion.
- ❖ The result will be software that is easier to integrate, easier to test, and easier to maintain.

Transform Flow

- ❖ Recalling the fundamental system model (level 0 data flow diagram), information must enter and exit software in an "external world" form.
- ❖ For example, data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information. Such externalized data must be converted into an internal form for processing.
- ❖ Information enters the system along paths that transform external data into an internal form. These paths are identified as ***incoming flow****.*
- ❖ At the kernel of the software, a transition occurs. Incoming data are passed through a ***transform center*** and begin to move along paths that now lead "out" of the software.
- ❖ Data moving along these paths are called ***outgoing flow****.*
- ❖ The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight line" paths. When a segment of a data flow diagram exhibits these characteristics, ***transform flow*** is present.

Transaction Flow

- ❖ The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category.
- ❖ However, information flow is often characterized by a single data item, called a ***transaction***, that triggers other data flow along one of many paths. When a DFD takes the form shown in Figure, ***transaction flow*** is present.
- ❖ Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and, based on its value, flow along one of many ***action paths*** is initiated.
- ❖ The *hub of information flow* from which many action paths originate is called a ***transaction center***. It should be noted that, within a DFD for a large system, both transform and transaction flow may be present.
- ❖ For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

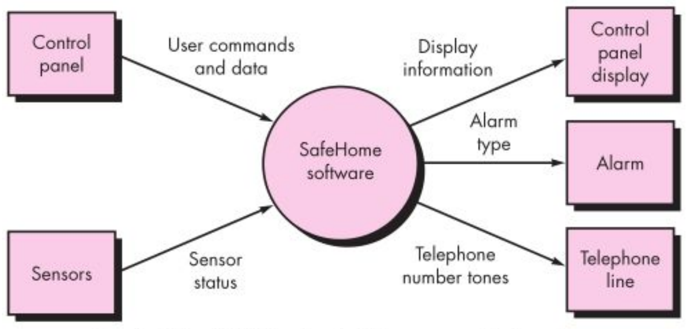


Transaction flow

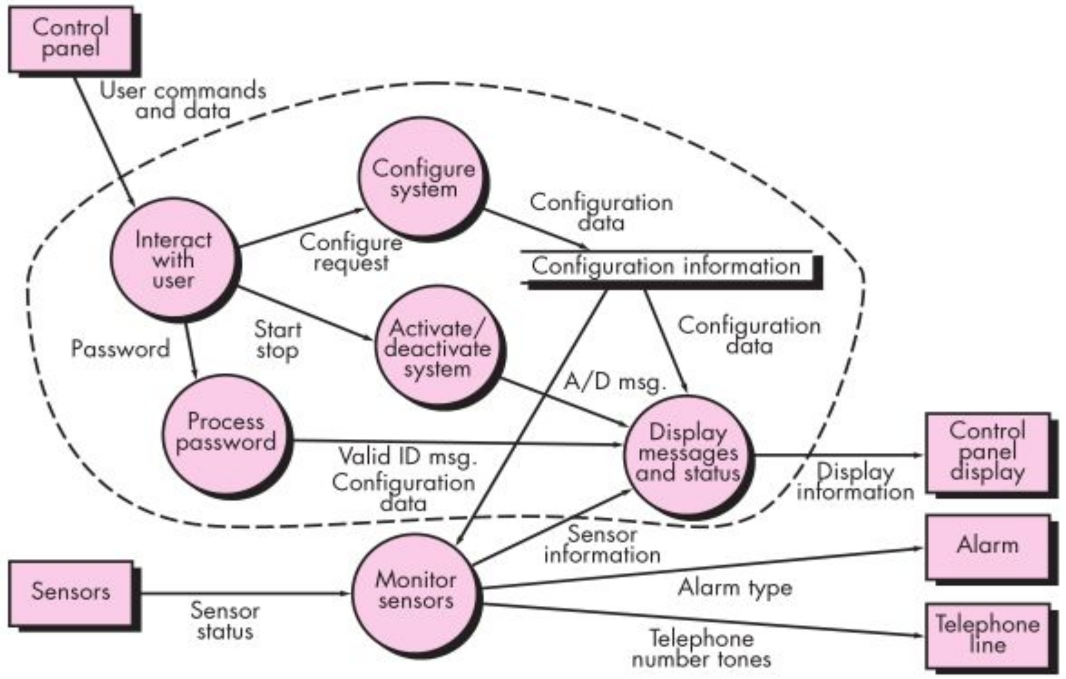
Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. One element of the analysis model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into a software architecture, you would initiate the following design steps:

Step 1. Review the fundamental system model. The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function.

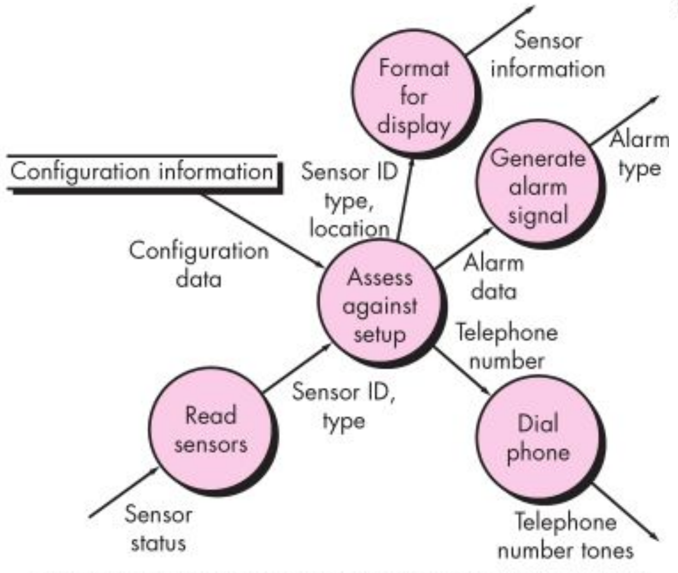


Context-level DFD for the SafeHome security function

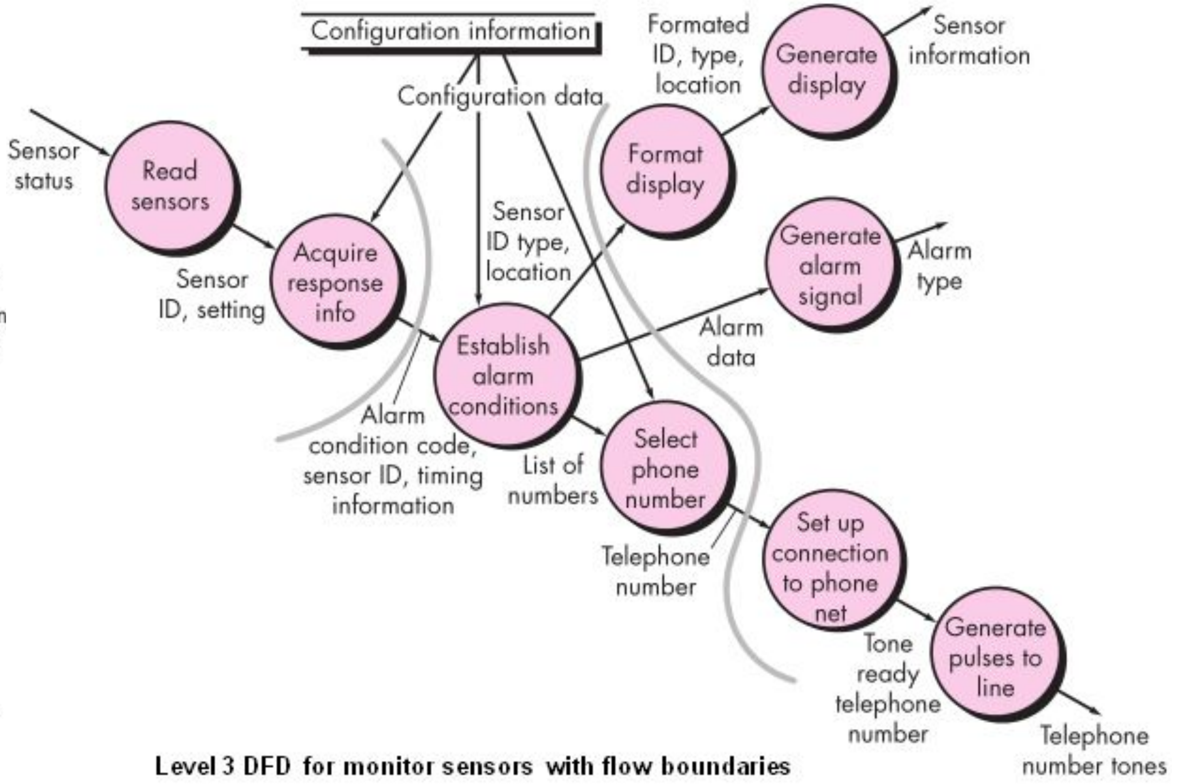


Level 1 DFD for the SafeHome security function

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail.



Level 2 DFD that refines the monitor sensors transform



Level 3 DFD for monitor sensors with flow boundaries

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD shown in Figure, we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

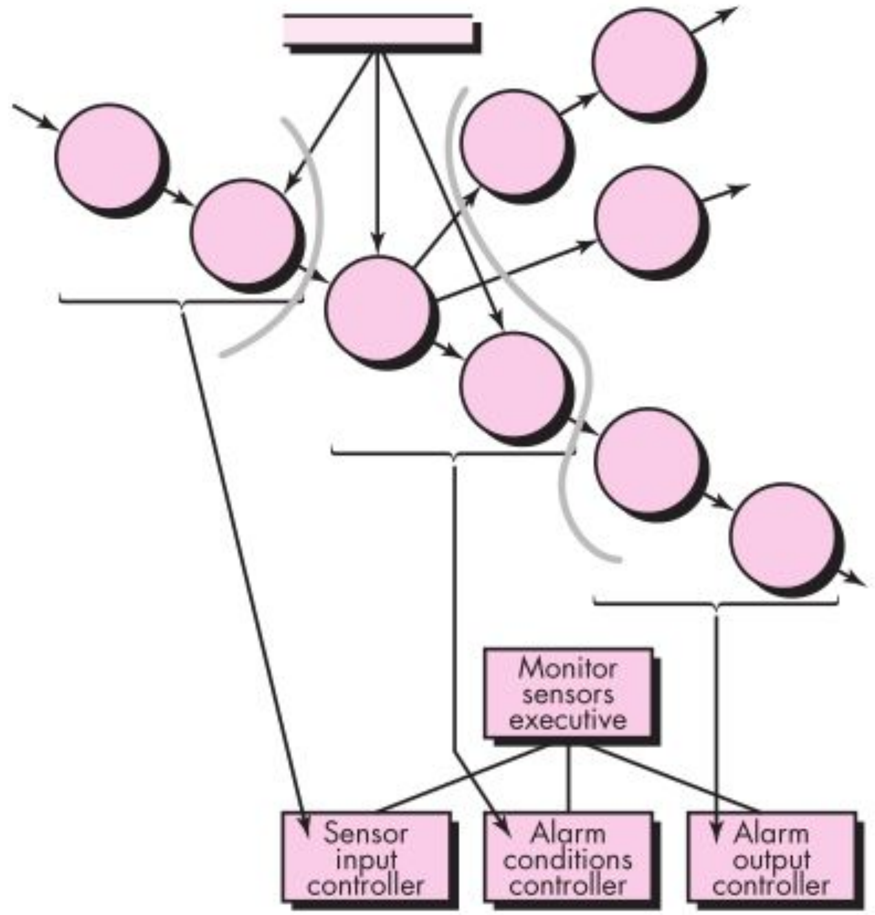
Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations.

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and low-level components perform most input, computation, and output work. Middle-level components 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.

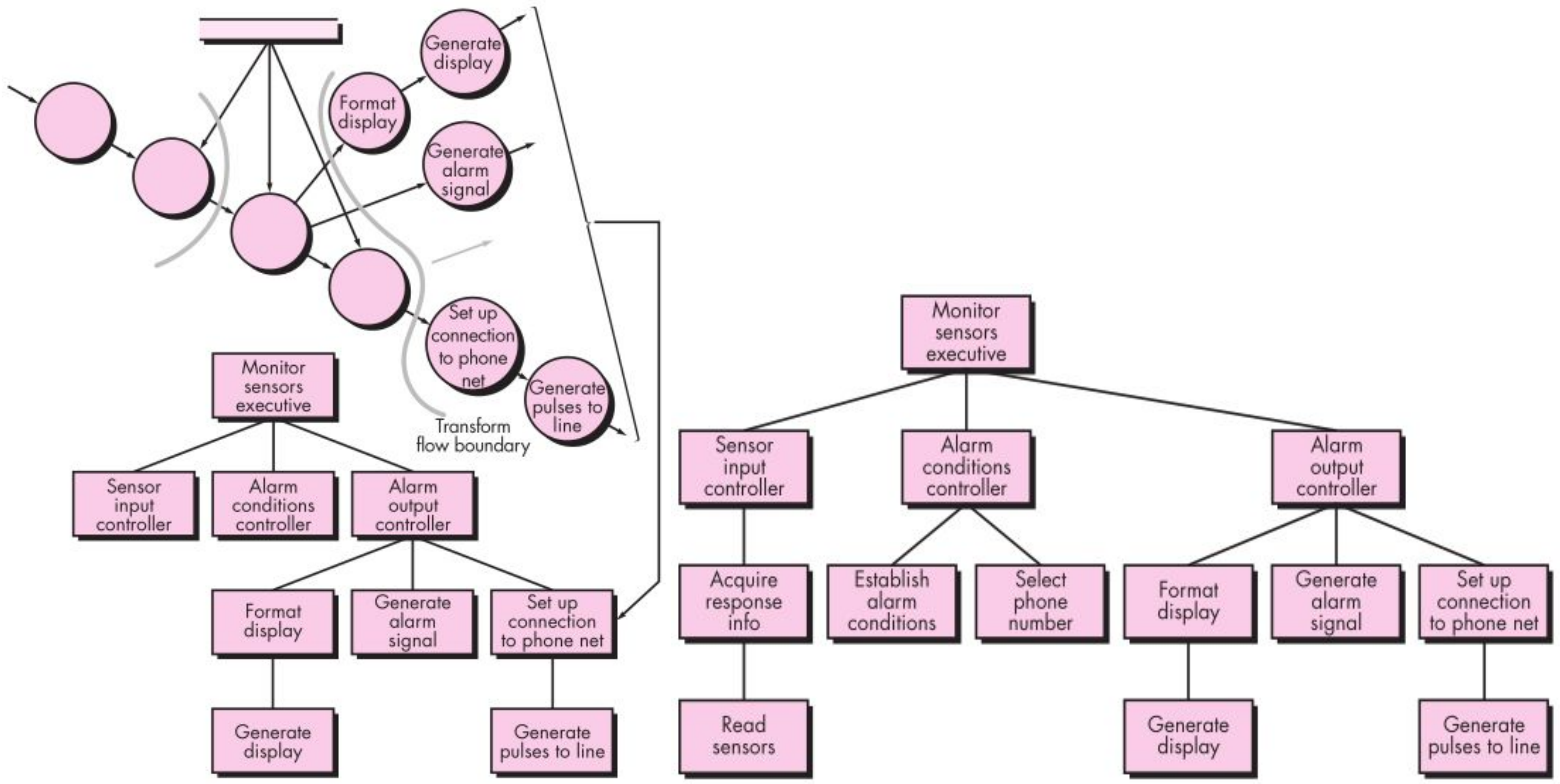
This first-level factoring for the monitor sensors subsystem is illustrated in Figure. A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:

- 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.



First-level factoring for monitor sensors

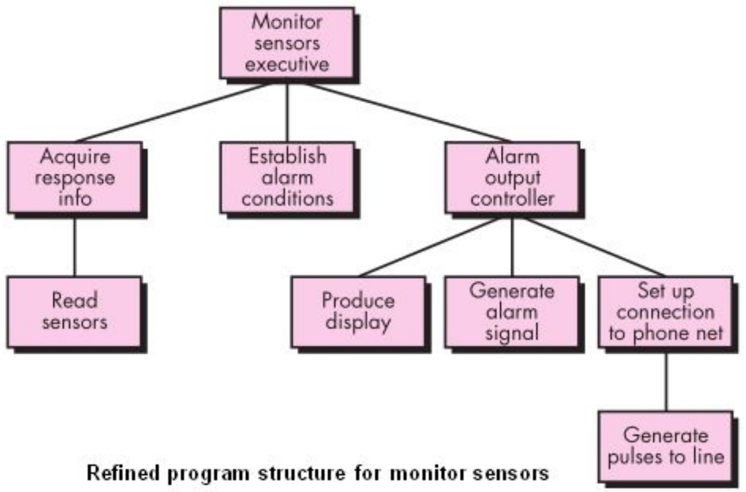
Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped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



Second-level factoring for monitor sensors

First-iteration structure for monitor sensors

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 functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.



Refined program structure for monitor sensors

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.

You should pause for a moment and consider the difference between the design approach described and the process of "writing programs." If code is the only representation of software, you and your colleagues will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty "seeing the forest for the trees."

TRANSACTION MAPPING

In many software applications, a single data item triggers one or a number of information flows that effect a function implied by the triggering data item. The data item, called a *transaction.*

Transaction mapping will be illustrated by considering the *user interaction subsystem* of the *SafeHome software.*

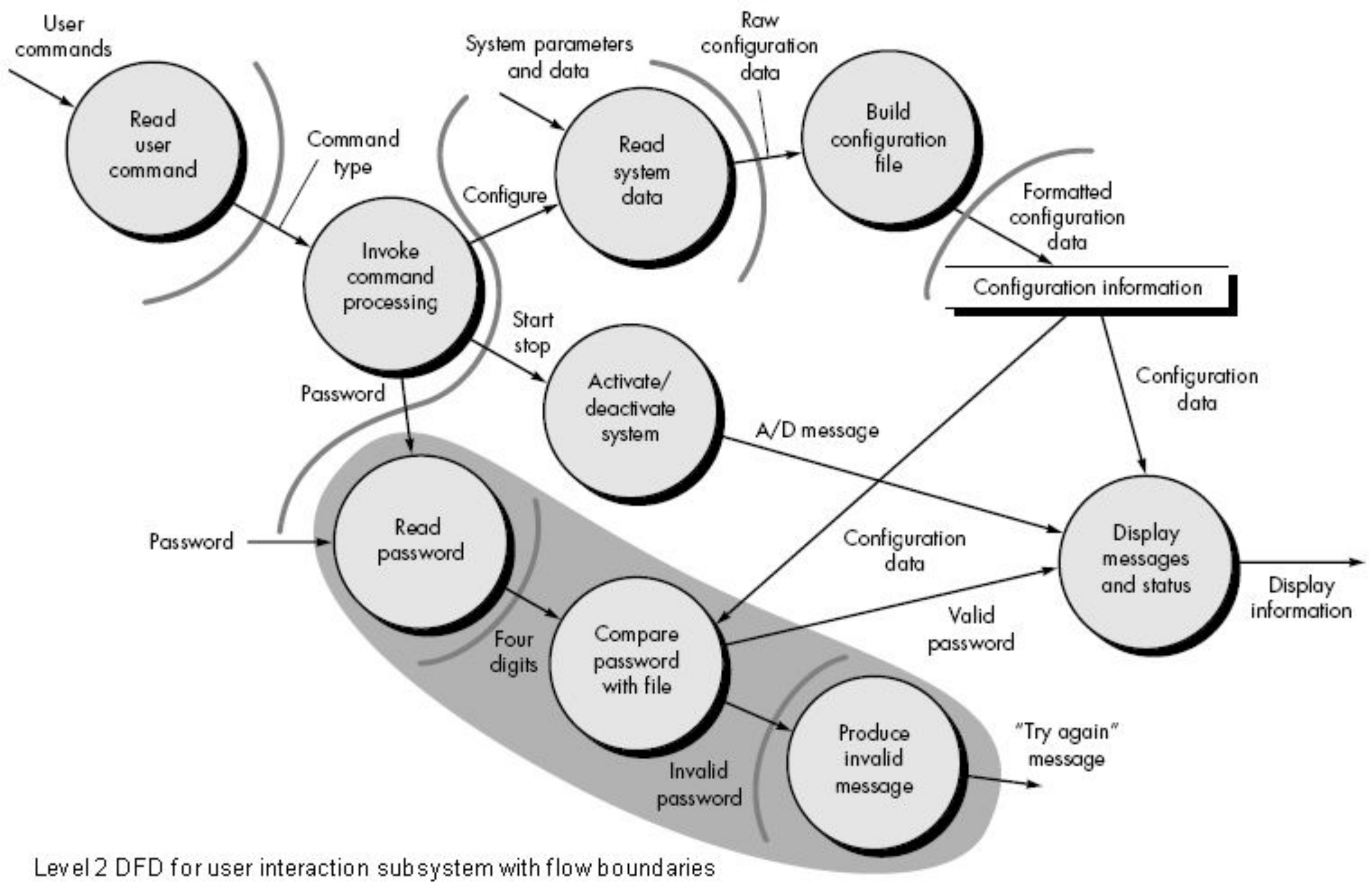
Design Steps:

The design steps for transaction mapping are similar and in some cases identical to steps for transform mapping. A major difference lies in the mapping of DFD to software structure.

Step 1. Review the fundamental system model.

Step 2. Review and refine data flow diagrams for the software.

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Steps 1, 2, and 3 are identical to corresponding steps in transform mapping. The DFD shown in Figure has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the *invoke command processing bubble appears to have transform flow characteristics. Therefore, flow* boundaries must be established for both flow types.



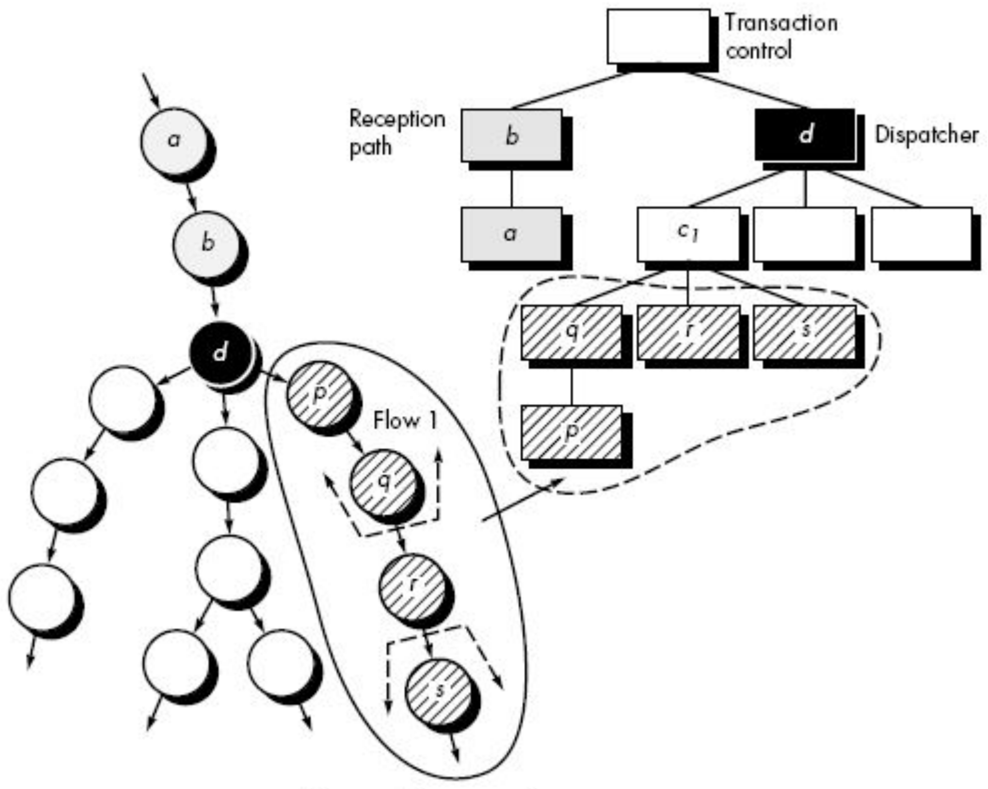
Level 2 DFD for user interaction subsystem with flow boundaries

Step 4. Identify the transaction center and the flow characteristics along each of the action paths. The location of the transaction center can be immediately discerned from the DFD. The transaction center lies at the origin of a number of actions paths that flow radially from it. For the flow shown in Figure, the *invoke command processing* bubble is the transaction center.

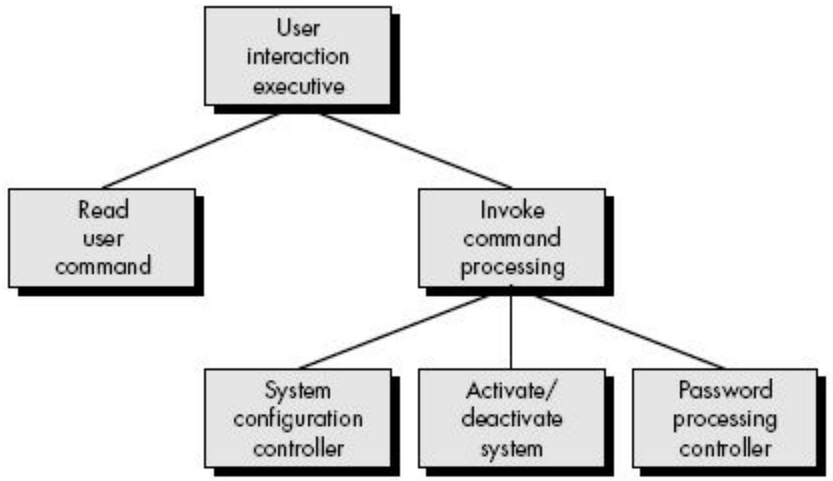
The incoming path (i.e., the flow path along which a transaction is received) and all action paths must also be isolated. Boundaries that define a reception path and action paths are also shown in the figure. Each action path must be evaluated for its individual flow characteristic.

Step 5. Map the DFD in a program structure amenable to transaction processing. Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch. The structure of the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into modules. The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules. Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics.

Considering the *user interaction subsystem data flow, first-level factoring for* step 5 is shown in Figure. The bubbles *read user command and activate/deactivate system map directly into the architecture without the need for intermediate control* modules. The transaction center, *invoke command processing, maps directly into* a dispatcher module of the same name.



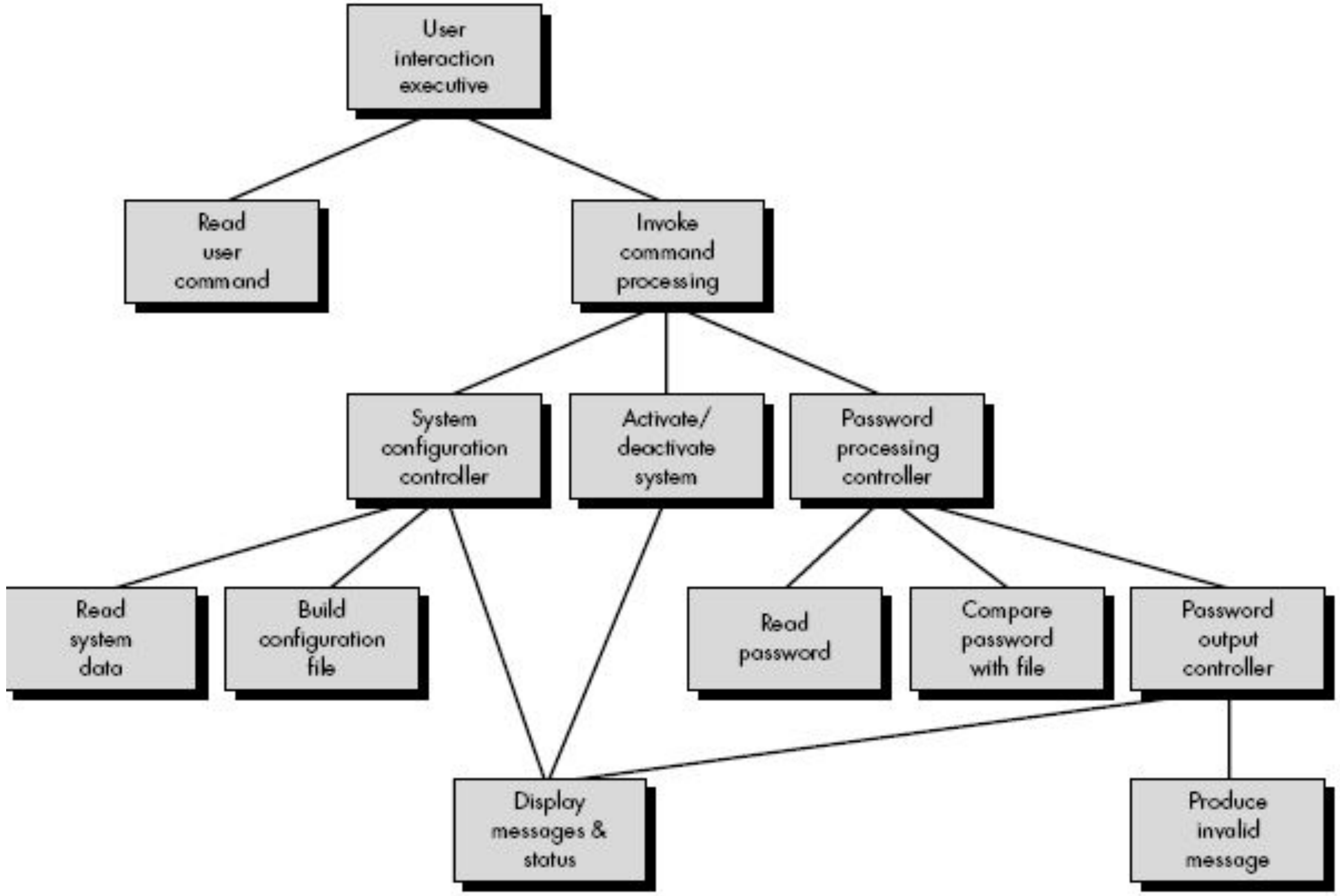
Transaction mapping


User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller
First-level factoring for user interaction subsystem

Step 6. Factor and refine the transaction structure and the structure of each action path. Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered.

As an example, consider the password processing information flow shown (inside shaded area) in Figure. The flow exhibits classic transform characteristics. A password is input (incoming flow) and transmitted to a transform center where it is compared against stored passwords. An alarm and warning message (outgoing flow) are produced (if a match is not obtained). The "configure" path is drawn similarly using the transform mapping. The resultant software architecture is shown in Figure.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.


User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller
Read system data
Build configuration file
Read password
Compare password with file
Password output controller
Display messages & status
Produce invalid message
First-iteration architecture for user interaction subsystem

DESIGN HEURISTICS FOR EFFECTIVE MODULARITY

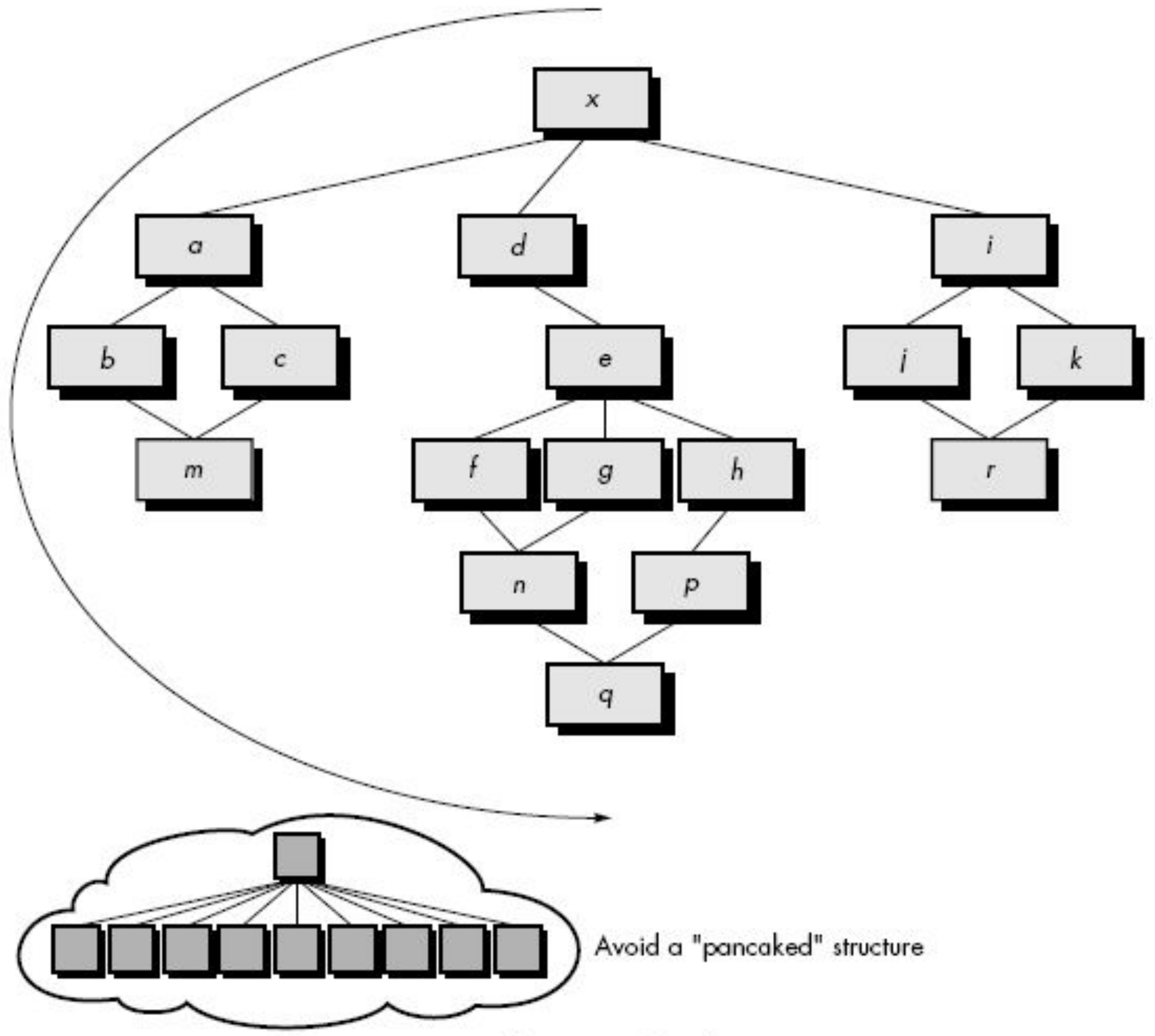
❖ Once program structure has been developed, effective modularity can be achieved by applying the design concepts. The program structure can be manipulated according to the following set of heuristics:

1. ***Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.*** Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence.

   An *exploded module* becomes two or more modules in the final program structure. An *imploded module* is the result of combining the processing implied by two or more modules. An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. ***Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.*** The structure shown inside the cloud in Figure does not make effective use of factoring. All modules are “pancaked” below a single control module.

   In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.



Program structures

3. *Keep the scope of effect of a module within the scope of control of that module.* The *scope of effect* of module *e* is defined as all other modules that *are* affected by a decision made in module *e.* The *scope of control of module e is* all modules that are subordinate and ultimately subordinate to module *e.* Referring to Figure, if module *e makes a decision that affects module r,* we have a violation of this heuristic, because module *r lies outside the scope* of control of module *e.*

4. *Evaluate module interfaces to reduce complexity and redundancy and improve consistency.* Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be reevaluated.

5. *Define modules whose function is predictable, but avoid modules that are overly restrictive.* A module is predictable when it can be treated as a *black box; that* is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use.

A module that restricts processing to a single subfunction exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. *Strive for “controlled entry” modules by avoiding "pathological connections."* This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. *Pathological connection* refers to branches or references into the middle of a module.