Table of Contents

Class 1: Multidisciplinary Projects as Software Projects

What Went Wrong?

- A 1995 Standish Group study (CHAOS): 16.2% of IT projects were successful in meeting scope, time, and cost goals; over 31% of IT projects were canceled.
- A PricewaterhouseCoopers study found >50% projects fail and 2.5% of meet their targets.

Formal Project Management

- Advantages of Using Formal Project Management:
    - Better control of financial, physical, and human resources
    - Improved customer relations
    - Shorter development times
    - Lower costs and improved productivity
    - Higher quality and increased reliability
    - Higher profit margins
    - Better internal coordination
    - Positive impact on meeting strategic goals
    - Higher worker morale

Project Attributes

- A project:
    - has a unique purpose.
    - is temporary.
    - drives change and enables value creation.
    - is developed using progressive elaboration.
    - requires resources, often from various areas.
    - involves uncertainty.
    - should have a primary customer or sponsor.
        - The project sponsor usually provides the direction and funding for the project.
- Project managers work with project sponsors, team, and other people involved in a project to achieve project goals.

Project constraints

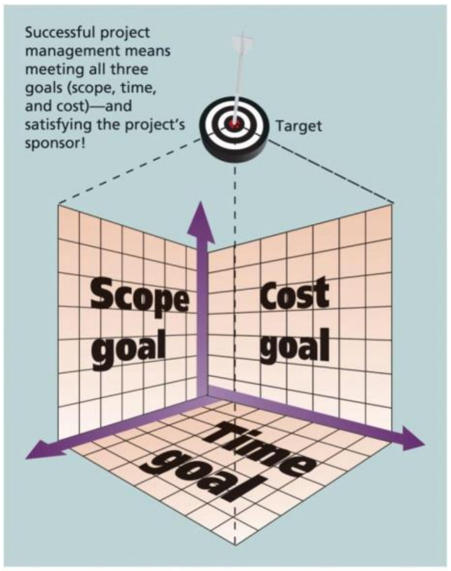

Successful project management means meeting all three goals (scope, time, and cost)—and satisfying the project's sponsor!
Target
Scope goal
Cost goal
Time goal

FIGURE 1-1 Project constraints

What is Project Management?

- "The application of knowledge, skills, tools and techniques to project activities to meet project requirements" (PMBOK® Guide, Sixth Edition, 2017)
- Project managers strive to meet the triple constraint (project scope, time, and cost goals) and facilitate the entire process to meet the needs and expectations of project stakeholders.

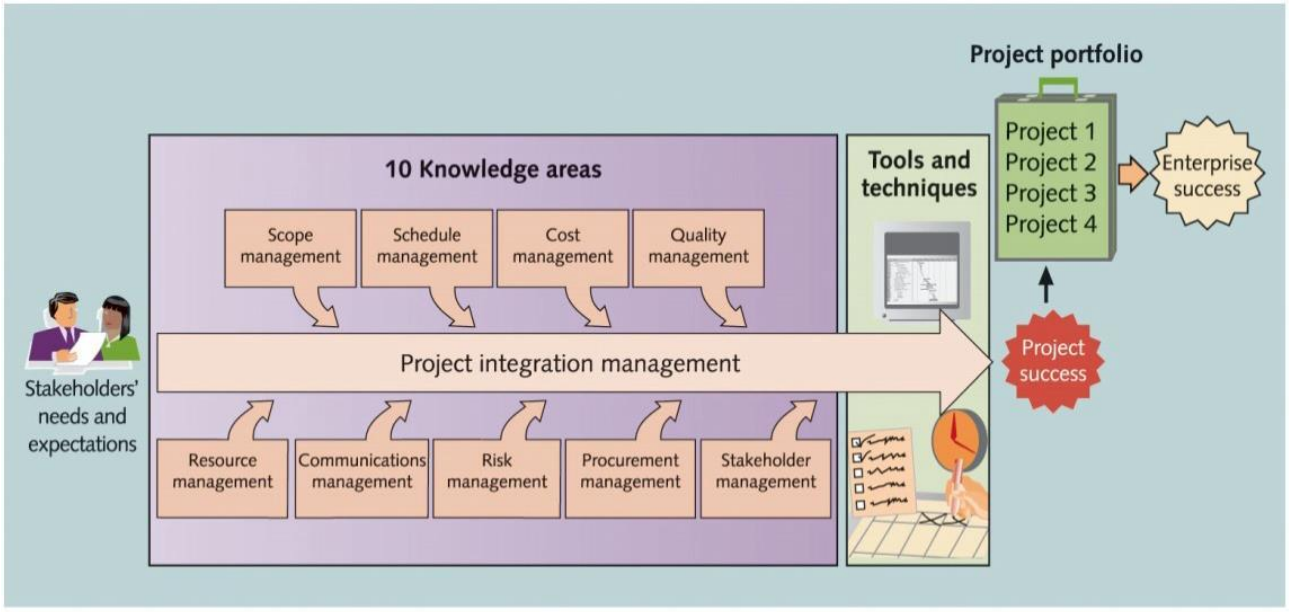


FIGURE 1-2 Project management framework

Aim of Project Management

- To complete a project:
    - On time
    - On budget
    - With required functionality
    - To the satisfaction of the client
    - Without exhausting the team
- To provide visibility about the progress of the project
- Clients wish to know:
    - Will the system do what was promised?
    - When will it be delivered? If late, how late?
    - How does the cost compare with the budget?
- Often the software is a part of a larger activity.
    - If the system is a product, marketing and development must be combined (e.g., Microsoft Office)
    - If the system has to work with other systems, developments must be coordinated (e.g., embedded systems in an automobile)

Aspects of Project Management

- Planning
    - Outline schedule during feasibility study (needed for SE)
    - Fuller schedule for each part of a project (e.g., each process step, iteration, or sprint)
- Contingency planning
    - Anticipation of possible problems (risk management)
- Progress tracking
    - Regular comparison of progress against plan
    - Regular modification of the plan
    - Changes of scope, etc. made jointly by client and developers.
- Final analysis
    - Analysis of project for improvements during next project

Terminology

- Deliverable
    - Work product that is provided to the client (mock-up, demonstration, prototype, report, presentation, documentation, code, etc.).
    - Release of a system or subsystem to customers or users
- Milestone
    - Completion of a specified set of activities (e.g., delivery of a deliverable, completion of a process step).
- Activity
    - Part of a project that takes place over time (also known as a task).
- Event
    - The end of a group of activities, e.g., agreement by all parties on the budget and plan.
- Dependency
    - An activity that cannot begin until some event is reached.
- Resource
    - Staff time, equipment, or other limited resources required by an activity.

Standard Approach to Project Management

- The scope of the project is defined early in the process.
- The development is divided into tasks and milestones.
- Estimates are made of the time and resources needed for each task.
- The estimates are combined to create a schedule and a plan.
- Progress is continually reviewed against the plan, perhaps weekly.
- The plan is modified by changes to scope, time, resources, etc.
- Typically, the plan is managed by a separate project management team, not by the software developers.

Agile Approach to Project Management

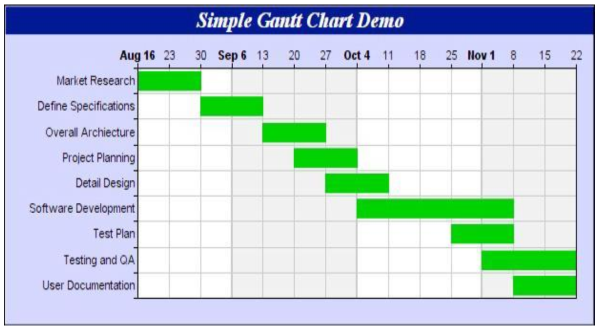
- Planning is divided into high level release forecasting and low-level detailed planning.
- Release planning is a best guess, high level view of what can be achieved in a sequence of time-boxes (sprint).
- Release plans are continually modified, perhaps daily.
- Clients and developers take joint control of the release plans and choice of sprints.

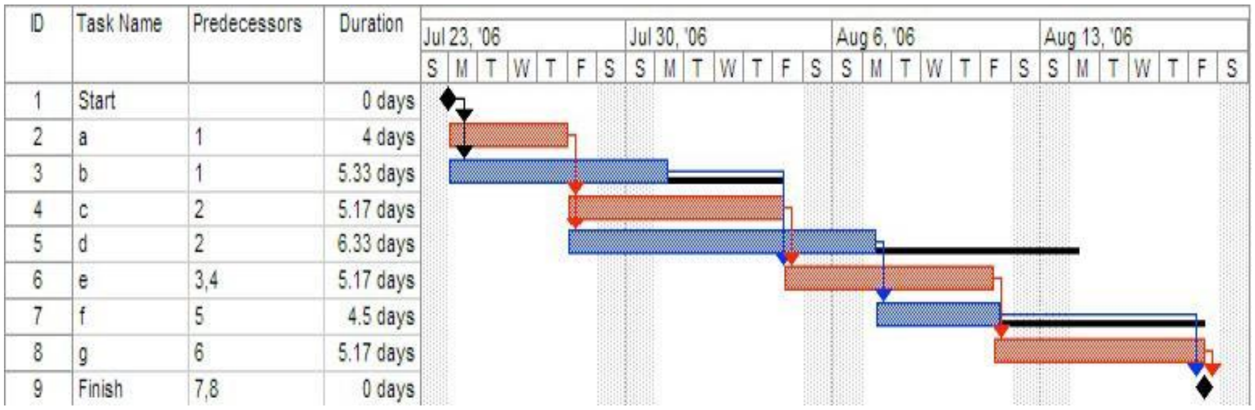
- For each time-box, the team plans what it can achieve. The team may use Gantt charts or other conventional planning tools.

Project Planning Tools

- Critical Path Method, Gantt charts, Activity bar charts, etc.
    - Build a work-plan from activity data.
    - Display work-plan in graphical or tabular form.
- Project planning software (e.g., Microsoft Project)
    - Maintain a database of activities and related data.
    - Calculate and display schedules.
    - Manage progress reports.

Gantt Chart





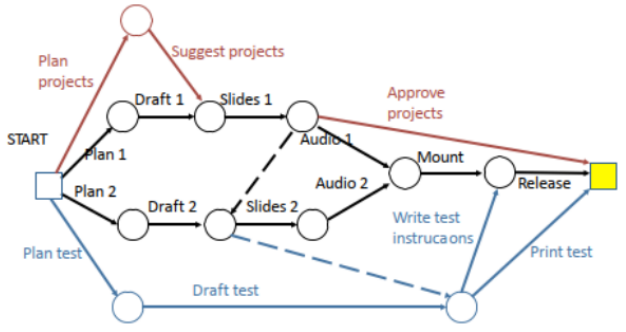
ID	Task Name	Predecessors	Duration
1	Start		0 days
2	a	1	4 days
3	b	1	5.33 days
4	c	2	5.17 days
5	d	2	6.33 days
6	e	3,4	5.17 days
7	f	5	4.5 days
8	g	6	5.17 days
9	Finish	7,8	0 days

Gantt Chart

- Used for small projects, single time-boxes, and sprints.
    - Dates run along the top (days, weeks, or months).
    - Each row represents an activity. Activities may be sequential, in parallel or overlapping.
    - The schedule for an activity is a horizontal bar. The left end marks the planned beginning of the task. The right end marks the expected end date.
    - The chart is updated by filling in each activity to a length proportional to the work accomplished.
    - Progress to date can be compared with the plan by drawing a vertical line through the chart at the current date.

Activity Graph

⟶	An activity (task)
⇢	A dummy activity (dependency)
○	An event
□	A milestone



Multi-disciplinary Project

Multi-disciplinary Project: a project comprising building work, together with its associated engineering work, where the engineer is subject to the authority of another professional acting as the Principal Agent while financial and administrative matters are dealt with by another professional.

Feasibility Study

- A feasibility study is a study made before committing to a project. A feasibility study leads to a decision:
    - go ahead.
    - do not go ahead.
    - think again.
- In production projects, the feasibility study often leads to a budget and resource request.
- A feasibility study may be in the form of a proposal.

Feasibility Study-Benefits

- Organization benefits
    - Create a marketable product.
    - Improve the efficiency of an organization (e.g., save staff)
    - Control a system that is too complex to control manually.
    - New or improved service (e.g., faster response to customers)
    - Safety or security
- Professional benefits are not the reason for doing a project.

Feasibility Study-Technical

- A feasibility study needs to demonstrate that the proposed system is technically feasible. This requires:
    - an outline of the requirements
    - a possible system design (e.g., database, distribution, etc.)
    - possible choices of software to be acquired or developed.
    - estimates of numbers of users, data, transactions, etc.
- These rough numbers are part of the provisional plan that is used to estimate the staffing, timetable, equipment needs, etc. The technical approach followed may be very different.

Feasibility Study-Planning and Resources

- The feasibility study must include an outline plan:
    - Estimate the staffing and equipment needs, and the preliminary timetable.
    -
    - Identify major milestones and decision points.
    - Identify interactions with and dependences on external systems.
- Provide a preliminary list of deliverables and delivery dates.

Feasibility Study-Alternatives and Risks

- A feasibility study should identify risks and alternatives.
    - What can go wrong?
    - How will progress be monitored and problems identified (visibility)?
    - What are the fall-back options?
- Alternatives
    - Continue with the current system, enhance it, or create new one?
    - Develop in-house, or contract out? (How will a contract be managed?)
    - Phases of delivery and possible points for revising plan.

Techniques for Feasibility Study

- The highest priority is to ensure that the client and development team have the same understanding of the goals of the system.
- For the development team to understand the goals:
    - Interviews with client and the staff of the client's organization.
    - Review of existing systems (including competitors')
- For the client to appreciate the proposed system:
    - Demonstration of key features or similar systems.
    - Walk through typical transactions or interactions.
- Outline budget:
    - n people for m months at \$$x$ per month.
    - equipment, buildings, etc.
    - contingency (at least 50% is needed)
- Phases/milestones:
    - specify deliverables and approximate dates.
    - planned releases.

Feasibility Study-Decision

- Different organizations and senior managers have different styles for feasibility studies, e.g., some decision makers:
    - Monitor the team and the process.
    - Rely on detailed reading of a written report.
    - Rely on face-to-face questioning of knowledgeable people.
- But they must understand the decision.

Feasibility Report

- Specific Requirements for the Feasibility Report
    - Outline plan, showing principal activities and milestones.
    - Discussion of Business Considerations.

    - Risk analysis. What can go wrong? What is your fall-back plan?

How to Minimize Risk?

- Techniques for managing risk:
    - Several target levels of functionality: required, desirable, optional phases.
    - Visible software process: intermediate deliverables
    - Good communication within the team, with the client, and with the experts.
    - Well defined development process
- The report is vague about the scope. Without a clear definition of scope, it is not clear that the project is feasible.
- The plan does not describe the activities in enough detail to estimate the effort convincingly.
- The projects are too ambitious. The report needs to describe how you will monitor the progress and adjust the scope if necessary.

Uncertainty Study

- Clients may be unsure of the scope of the project.
- Benefits are usually very hard to quantify.
- Approach is usually ill-defined. Estimates of resources and timetable are very rough.
- Organizational changes may be needed.

Feasibility Study Difficulties

- Feasibility studies rely heavily on the judgment of experienced people.
- Mistakes made at the beginning of a project are the most difficult to correct.
- **Advocacy** is needed to build enthusiasm for a project: to convince an organization to undertake an expensive, complex project with many risks.
- **Enthusiasm** is good, but enthusiasts usually emphasize potential benefits and downplay risks.

Personnel Performance

- In the project computing, not all people are equal.
    - The best are ten times more productive.
    - Some tasks are too difficult for everybody.
- Adding more people adds communications complexity.
    - Some activities need a single mind.
    - Sometimes, the elapsed time for an activity cannot be shortened.
- Adding more people may increase the time to complete a project.

Project Manager

- Create and maintain the schedule.
- Track progress against schedule.
- Keep some slack in the schedule (minimize risk).
- Continually make adjustments:
    - Start activities before previous activity complete.
    - Sub-contract activities.
    - Renegotiate deliverables.
- Keep senior management informed (visibility).
- The project manager needs the support of the head of the development team and the confidence of the team members.

Class 2: Requirements

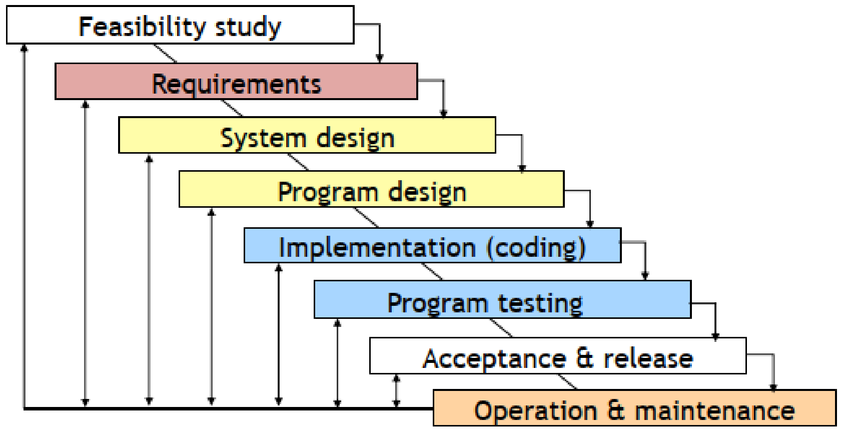
Requirements

- Requirements define the function of the system from the client's viewpoint.
    - The requirements establish the system's functionality, constraints, and goals by consultation with the client, customers, and users.
    - The requirements may be developed in a self-contained study or may emerge incrementally.
    - The requirements form the basis for acceptance testing.
- The development team and the client need to work together closely during the requirements phase of a software project.
    - The requirements must be developed in a manner that is understandable by both the client and the development staff.

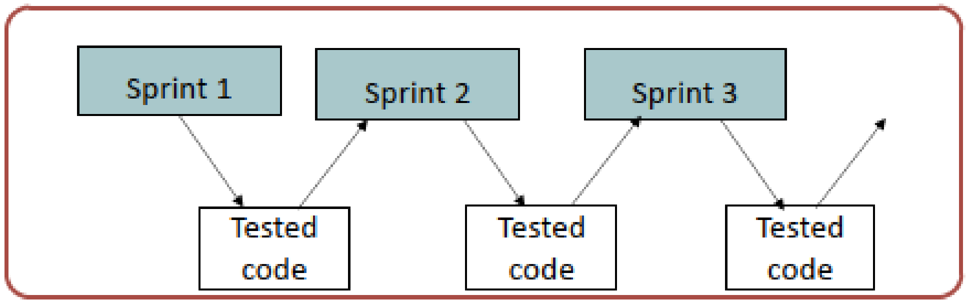
Why are Requirements Important?

- Causes of failed software projects
  - Incomplete requirements 13.1%
  - Lack of user involvement 12.4%
  - Lack of resources 10.6%
  - Unrealistic expectations 9.9%
  - Lack of executive support 9.3%
  - Changing requirements & specifications 8.8%
  - Lack of planning 8.1%
  - System no longer needed 7.5%
- Failure to understand the requirements led the developers to build the wrong system.

Requirements in the Modified Waterfall Model



Requirements with Agile Development



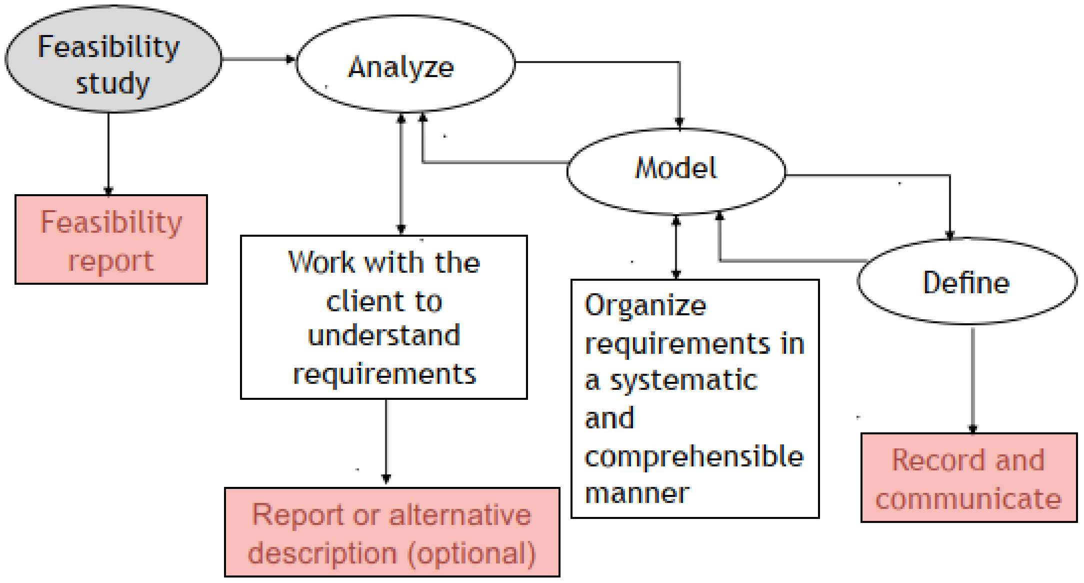
Requirement Goals

- Understand the requirements in appropriate detail.
- Define the requirements in a manner that is clear to the client. This may be a written specification, prototype system, or other form of communication.
- Define the requirements in a manner that is clear to the people who will design, implement, and maintain the system.
- Ensure that the client and developers understand the requirements and their implications.

Steps in the Requirements Phase

- The requirements part of a project can be divided into several stages:
    - Analysis to establish the system's services, constraints, and goals by consultation with client, customers, and users.
    - Modeling to organize the requirements in a systematic and comprehensible manner.
    - Define, record, and communicate the requirements.

The Requirements Process


Feasibility study
Analyze
Model
Define
Feasibility report
Work with the client to understand requirements
Organize requirements in a systematic and comprehensible manner
Report or alternative description (optional)
Record and communicate

Interviews with Clients

- Client interviews are the heart of the requirements analysis.
- Clients may have only a vague concept of requirements.
    - Allow plenty of time.
    - Prepare before you meet with the client.
    - Keep full note.
    - If you do not understand, delve further, again and again.
    - Repeat what you hear.
- For your projects you will need to schedule several meetings with your client to analyze the requirements.
- Small group meetings are often most effective.

Understand the Requirements

- Understand the requirements in depth:
    - Domain understanding
        - Example: Manufacturing light bulbs
    - Understanding of the real requirements of all stakeholders
        - Stakeholders may not have clear ideas about what they require, or they may not express requirements clearly. They are often too close to the old way of doing things.
    - Understanding the terminology
        - Clients often use specialized terminology. If you do not understand it, ask for an explanation.
- Keep asking questions, “Why do you do things this way?” “Is this essential?” “What are the alternatives?”

New and Old Systems

- A new system is when there is no existing system. This is rare.
- A replacement system is when a system is built to replace an existing system.
- A legacy system is an existing system that is not being replaced but must interface to the new system.
- Clients often confuse the current system with the underlying requirement.
- In requirements analysis it is important to distinguish:
    - features of the current system that are needed in the new system.
    - features of the current system that are not needed in the new system.
    - proposed new feature.

Unspoken Requirements

- Discovering the unspoken requirements is often the most difficult part of developing the requirements.
- Examples:
    - Resistance to change
    - Departmental friction (e.g., transfer of staff)
    - Management strengths and weaknesses

Stakeholders

- Identify the stakeholders: Who is affected by this system?
    - Client
    - Senior management
    - Production staff
    - Computing staff
    - Customers
    - Users (many categories)
    - *etc., etc., etc.,*
- Example: web shopping site (shoppers, administration, finance, warehouse). SE projects that build web applications often find that the administrative system that is not seen by the users is bigger than the part of the site that is visible to the users.

Viewpoint Analysis

- Analyze the requirements as seen by each group of stakeholders.
- Example: University Admissions System
    - Applicants
    - University administration
        - Admissions office
        - Financial aid office
        - Special offices (e.g., athletics, development)
    - Academic departments
    - Computing staff
    - Operations and maintenance

Special Studies

- Market research
    - focus groups, surveys, competitive analysis, etc.
    - Example: Windows XP T-shirt that highlighted Apple's strengths

- Technical evaluation
    - experiments, prototypes, etc.

        - Example: Windows XP boot faster

Defining and Communicating Requirements

- Record agreement between clients and developers
    - Provide a basis for acceptance testing.
    - Provide visibility.
    - Be a foundation for system and program design.
    - Communicate with other teams who may work on or rely on this system.
    - Inform future maintainers.

Defining and Communicating Requirements-Realism and Verifiability

- Requirements must be realistic, i.e., it must be possible to meet them.
    - Wrong: The system must be capable of x.
    - Right: The computer system can do x at a reasonable cost.
- Requirements must be verifiable, i.e., since the requirements are the basis for acceptance testing, it must be possible to test whether a requirement has been met.
    - Wrong: The system must be easy to use.
    - Right: After one day's training an operator should be able to input 50 orders per hour.

Defining and Communicating Requirements: <u>Option 1-Heavyweight Processes</u>

- Heavyweight software processes expect detailed specification.
    - Written documentation that specifies each requirement in detail.
    - Carefully checked by client and developers.
    - May be a contractual document.
- Difficulties
    - Time consuming and difficult to create.
    - Time consuming and difficult to maintain.
    - Checking a detailed specification is difficult and tedious.
    - Details may obscure the overview of the requirements.
    - Clients rarely understand the implications.
- The difficulty of creating and maintaining a detailed requirements specification is one of the reasons that many organizations are attracted to lighter weight development processes.

Defining and Communicating Requirements: <u>**Option 2-Lightweight Processes**</u>

- Lightweight processes use an outline specification + other tools.
  - Documentation describing key requirements in appropriate detail.
  - Reviewed by client and developers.
- Details provided by supplementary tools, e.g.,
  - User interface mock-up or demonstration.
  - Models, e.g., database schema, state machine, etc.
- Clients understand prototypes and models better than specification.
- Iterative or incremental (agile) development processes allow the client to appreciate what the final system will do.

<u>Option 2-Lightweight Processes</u>

- With lightweight processes, experience and judgment are needed to distinguish between details that can be left for later in the development process and key requirements that must be agreed with the client early in the process.
- Examples where detailed specifications are usually needed.
  - Business rules (e.g., reference to an accounting standard)
  - Legal restraint (e.g., laws on retention of data, privacy)
  - Data flow (e.g., sources of data, data validation)
- A common fault is to miss crucial details. This results in misunderstandings between the client and the developers.
- Yet the whole intent of lightweight processes is to have minimal intermediate documentation.

Functional Requirements

- Functional requirements describe the functions that the system must perform. They are identified by analyzing the use made of the system and include topics such as:
  - Functionality
  - Data
  - User interfaces

Non-functional Requirements

- Requirements that are not directly related to the functions that the system must perform.
  - Product requirements
    - performance, reliability, portability, etc...

    - Organizational requirements
        - delivery, training, standards, etc...
    - External requirements
        - legal, interoperability, etc...
    - Marketing and public relations

Functional vs. Nonfunctional Requirements

- A functional requirement describes **what** a software system should do, while non-functional requirements place constraints on **how** the system will do so.
- Functional requirement would be:
    - A system must send an email whenever a certain condition is met (e.g., an order is placed, a customer signs up, etc.).
- A related non-functional requirement for the system may be:
    - Emails should be sent with a latency of no greater than 5 hours after such an activity.

Requirements Confliction

- Some requests from the client may conflict with others:
- Recognize and resolve conflicts (e.g., functionality vs. cost vs. timeliness)
- Example:
    - Windows XP boot faster: shorter time-out vs. recognizes all equipment on bus.

Negotiation with the Client

- Sometimes the client will request functionality that is very expensive or impossible. What do you do?
    - Talk through the requirement with the client. Why is it wanted? Is there an alternative that is equivalent?
    - Explain the reasoning behind your concern. Explain the technical, organizational, and cost implications.
    - Be open to suggestions. Is there a gap in your understanding? Perhaps a second opinion might suggest other approaches.
- Before the requirements phase is completed the client and development team must resolve these questions.

Scenarios

- A scenario is a scene that illustrates some interaction with a proposed system.

- A scenario is a tool used during requirements analysis to describe a specific use of a proposed system. Scenarios capture the system, as viewed from the outside, e.g., by a user, using specific examples.
- Note on terminology:
  - Some authors restrict the word "scenario" to refer to a user's total interaction with the system.
  - Other authors use the word "scenario" to refer to parts of the interaction.
  - In this course, the term is used with both meanings.

Describing a Scenario

- Some organizations have complex documentation standards for describing a scenario.
- At the very least, the description should include:
  - A statement of the purpose of the scenario
  - The individual user or transaction that is being followed through the scenario.
  - Assumptions about equipment or software
  - The steps of the scenario

Developing a Scenario with a Client

- The requirements are being developed for a system that will enable university students to take exams online from their own rooms using a web browser.
- Create a scenario for how a typical student interacts with the system.

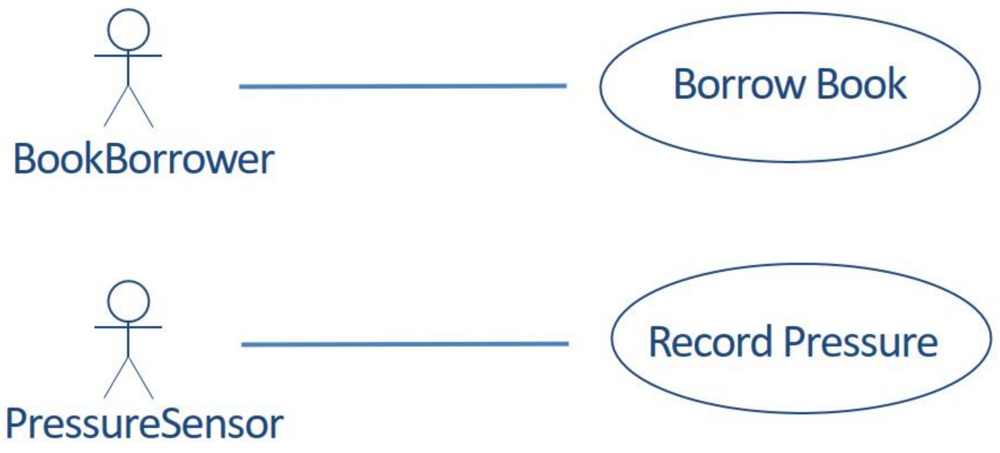
- Purpose: Scenario that describes the use of an online Exam system by a representative student
- Individual: [Who is a typical student?] Student A, senior at VGU, majoring in IT. [Where can the student be located? Do other universities differ?]
- Equipment: Any computer with a supported browser. [Is there a list of supported browsers? Are there any network restrictions?]
- 1. Student A authenticates. [How does an VGU student authenticate?]
- 2. Student A starts the browser and types the URL of Exam system. [How does the student know the URL?]
- 3. Exam system displays list of options. [Is the list tailored to the individual user?]
- 4. Student A selects Exam 1.
- 5. A list of questions is displayed, each marked to indicate whether completed or not. [Can the questions be answered in any order?]
- 6. Student A selects a question and chooses whether to submit a new answer or edit a previous answer. [Is it always possible to edit a previous answer? Are there other options?]
- 7. [What types of question are there: text, multiple choice, etc.?] The first question requires a written answer. Student A is submitting a new answer. The student has a choice whether to type the solution into the browser or to attach a separate file. Student A decides to attach a file. [What types of file are accepted?]
- 8. For the second question, the student chooses to edit a previous answer. Student A chooses to delete a solution previously typed into the browser, and to replace it with an attached file. [Can the student edit a previous answer, or must it always be replaced with a new answer?]
- 9. As an alternative to completing the entire exam in a single session, Student A decides to saves the completed questions to continue later. [Is this always permitted?]
- 10. Student A logs off.
- 11. Later Student A log in, finishes the exam, submits the answers, and logs out. [Is this process any different from the initial work on this exam?]
- 12. Student A has now completed the exam. The student selects an option that submits the exam to the grading system. [What if the student has not attempted every question? Is the grader notified?]

- 13. Student A now wishes to change a solution. The system does not permit changes once the solution has been submitted. [Can the student still see the solutions?]
- 14. Later Student A logins in to check the grades. [When are grades made available? How does the student know?]
- 15. Student A requests a regrade. [What are the policies? What are the procedures?]

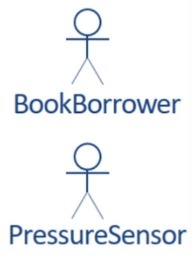
Modeling Scenarios as Use Cases

- Scenarios are useful in discussing a proposed system with a client, but requirements need to be made more precise before a system is fully understood.
- This is the purpose of requirements modeling.
- A use case provides such a model.

Two Simple Use Cases

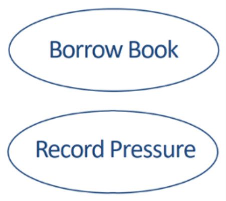


Actor and Use Case Diagram



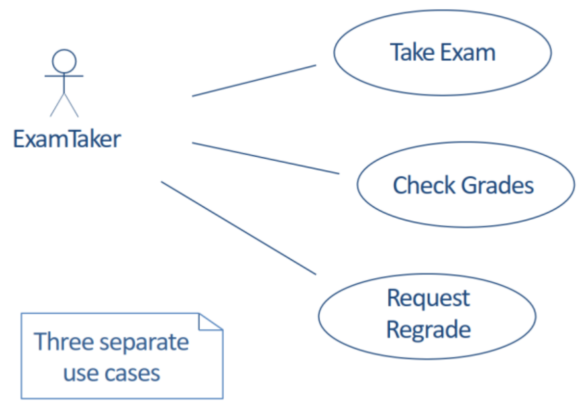
An **actor** is a user of a system in a particular role.

An **actor** can be human or an external system.



A **use case** is a task that an actor needs to perform with the help of the system

Use Cases for Exam System



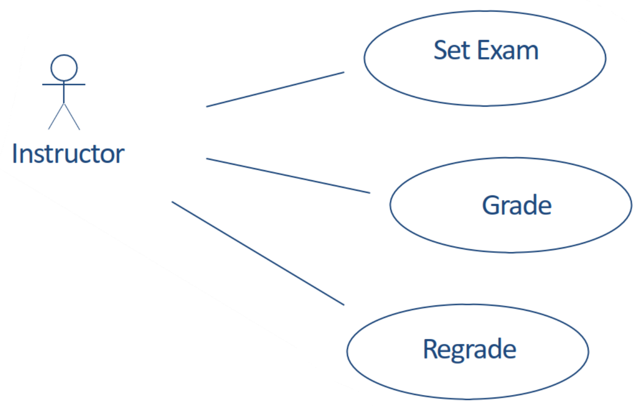
Describing a Use Case

- Some organizations have complex documentation standards for describing a use case.
- At the very least, the description should include:
    - The name of the use case, which should summarize its purpose
    - The actor or actors
    - The flow of events
    - Assumptions about entry conditions

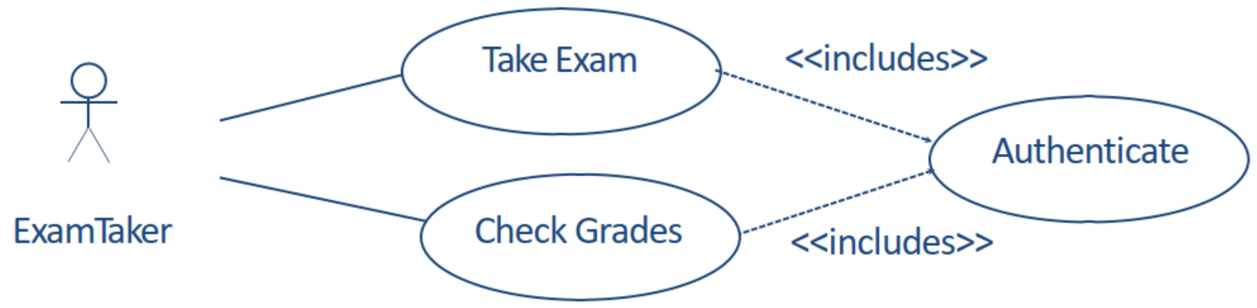
Outline of Take Exam Use Case

- Name of Use Case: Take Exam
- Actor(s): ExamTaker
- Flow of events:
- 1. ExamTaker connects to the Exam server.
- 2. Exam server checks whether ExamTaker is already authenticated and runs authentication process if necessary.
- 3. ExamTaker selects a exam from a list of options.
- 4. ExamTaker repeatedly selects a question and either types in a solution, attaches a file with a solution, edits a solution or attaches a replacement file.
- 5. ExamTaker either submits completed exam or saves current state.
- 6. When a completed exam is submitted, Exam server checks that all questions have been attempted and either sends acknowledgement to ExamTaker or saves current state and notifies ExamTaker of incomplete submission.
- 7. ExamTaker logs out.
- **Entry conditions:**

- 1. ExamTaker must have authentication credentials.
- 2. Computing requirements: supported browser.
- **Note:**
    - Actor is a role. An individual can be an ExamTaker
    - On one occasion and an Instructor at a different time.



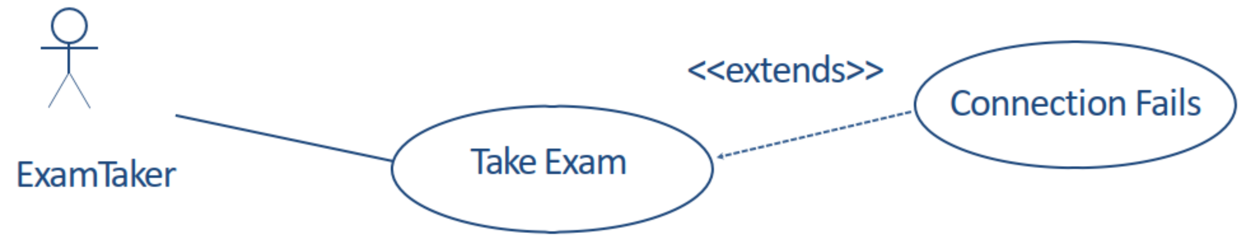
Relationships Between Use



Cases: <<includes>>

<<**includes**>> is used for use cases that are in the flow of events of the main use case.

Relationships Between Use Cases: <<extends>>



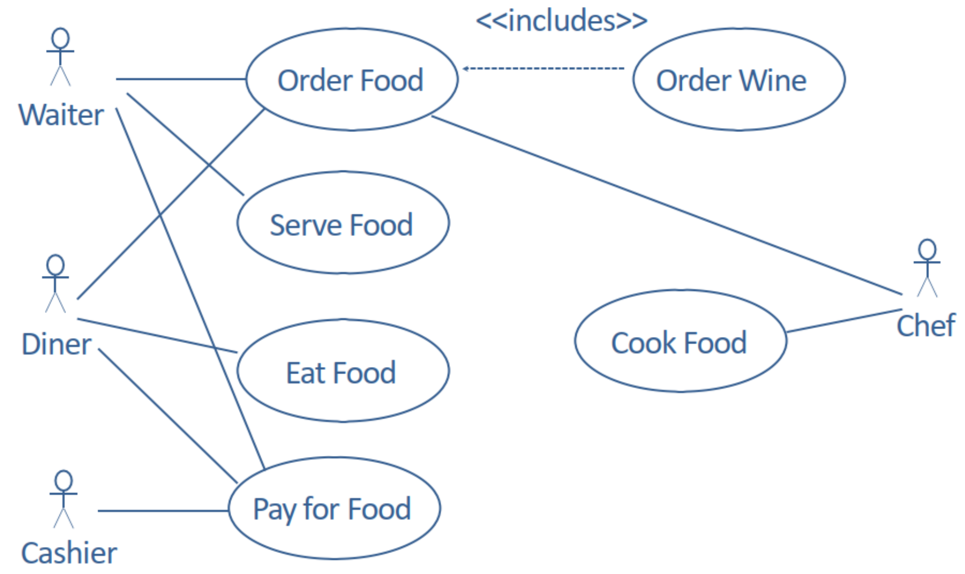
<<**extends**>> is used for exceptional conditions, especially those that can occur at any time.

Scenarios and Use Cases in the Development Cycle

- Scenarios and use cases are both intuitive -- easy to discuss with clients
- **Scenarios** are a tool for requirements analysis.
    - They are useful to validate use cases and in checking the design of a system.

    - They can be used as test cases for acceptance testing.
- **Use cases** are a tool for modeling requirements.
    - A set of use cases can provide a framework for the requirements specification.
    - Use cases are the basis for system and program design, but are often hard to translate into class models.

Use Cases with Several Actors



An Examination Question

The Pizza Ordering System

- The Pizza Ordering System allows the user of a web browser to order pizza for home delivery. To place an order, a shopper searches to find items to purchase, adds items one at a time to a shopping cart, and possibly searches again for more items. When all items have been chosen, the shopper provides a delivery address. If not paying with cash, the shopper also provides credit card information. The system has an option for shoppers to register with the pizza shop. They can then save their name and address information, so that they do not have to enter this information every time that they place an order.
- Develop a use case diagram, for a use case for placing an order, PlaceOrder. The use case should show a relationship to two previously specified use cases, IdentifyCustomer, which allows a user to register and log in, and PaybyCredit, which models credit card payments.

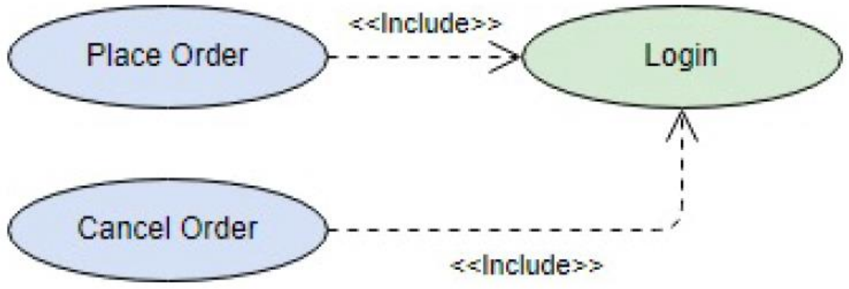
How to Draw a Use Case Diagram?

A Use Case model can be developed by following the steps below.

- 1. Identify the Actors (role of users) of the system.
- 2. For each category of users, identify all roles played by the users relevant to the system.
- 3. Identify what are the users required the system to be performed to achieve these goals.
- 4. Create use cases for every goal.
- 5. Structure the use cases.
- 6. Prioritize, review, estimate and validate the users.

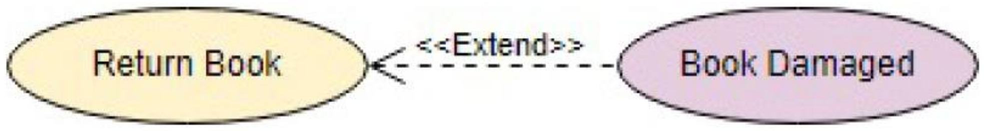
<<include>> Use Case

The time to use the <<include>> relationship is after you have completed the first cut description of all your main Use Cases. You can now look at the Use Cases and identify common sequences of user-system interaction.



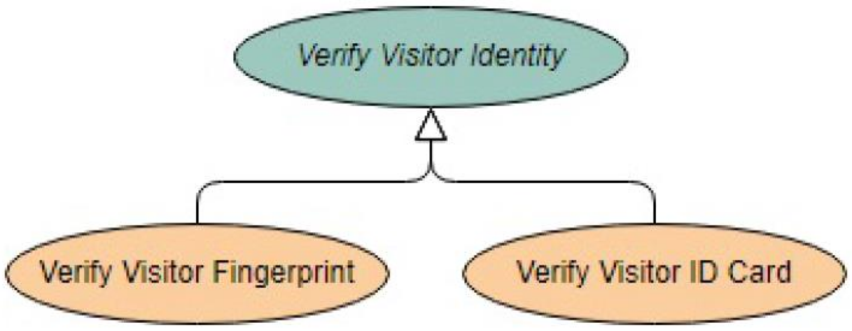
<<extend>> Use Case

An extending use case is, effectively, an alternate course of the base use case. The <<extend>> use case accomplishes this by conceptually inserting additional action sequences into the base use-case sequence.



Abstract and Generalized Use Case

The general use case is abstract. It cannot be instantiated, as it contains incomplete information. The title of an abstract use case is shown in italics.



Models for Requirements Analysis and Specification

- As you build understanding of the requirements through viewpoint analysis, scenarios, use cases, etc., use models to analyze and specify requirements. The models provide a bridge between the client's understanding and the developers'.
- The craft of requirements analysis and specification includes selecting the appropriate tool for the particular task.
  - A variety of tools and techniques.
  - Many familiar from other courses.
  - No correct technique that fits all situations.

Models

- A model is a simplification of reality.
  - We build models so that we can better understand the system we are developing.
  - We build models of complex system because we cannot comprehend such a system in its entirety.
- Models can be informal or formal. The more complex the project the more valuable a formal model becomes.

Principles of Modeling

- The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
- No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.
- Every model can be expressed at different levels of precision.
- Good models are connected to reality.

The Unified Modeling Language (UML)

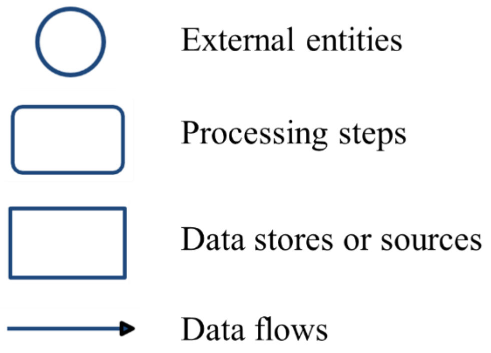
- UML is a standard language for modeling software systems
- Serves as a bridge between the requirements specification and the implementation.
- Provides a means to specify and document the design of a software system.
- Is process and programming language independent.
- Is particularly suited to object-oriented program development.

Models: Diagrams and Specification in UML

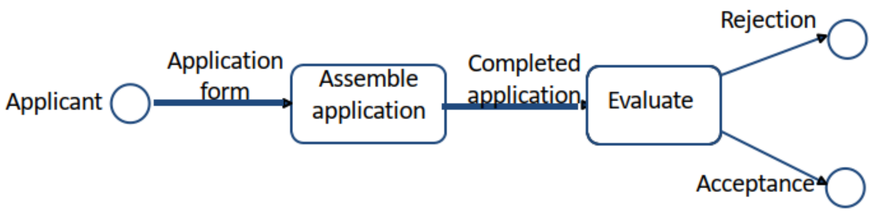
- In UML, a model consists of a diagram and a specification.
- A diagram is the graphical representation of a set of elements, usually rendered as a connected graph of vertices (things) and arcs (relationships).
- Each diagram is supported by technical documentation that specifies in more detail the model represented by the diagram.
- A diagram without a specification is of little value.

Data-Flow Models

An informal modeling technique to show the flow of data through a system.

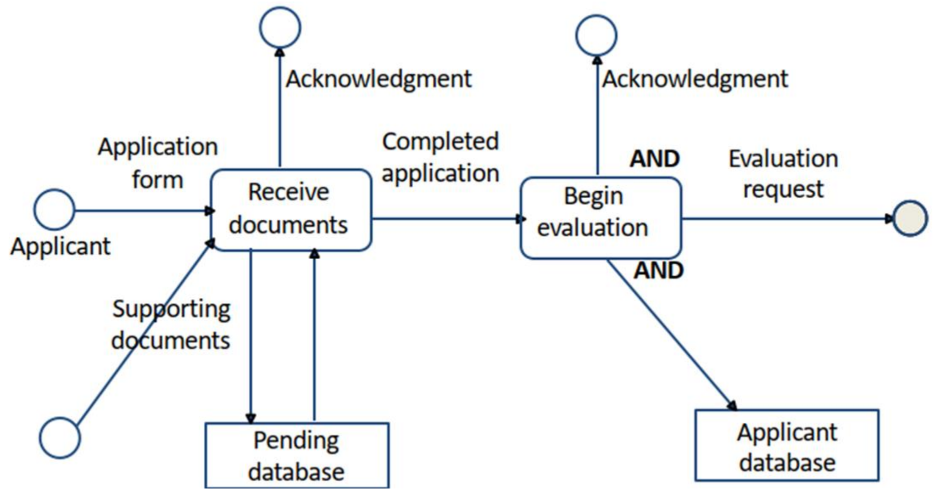


- Example: ***University Admissions (first attempt)***



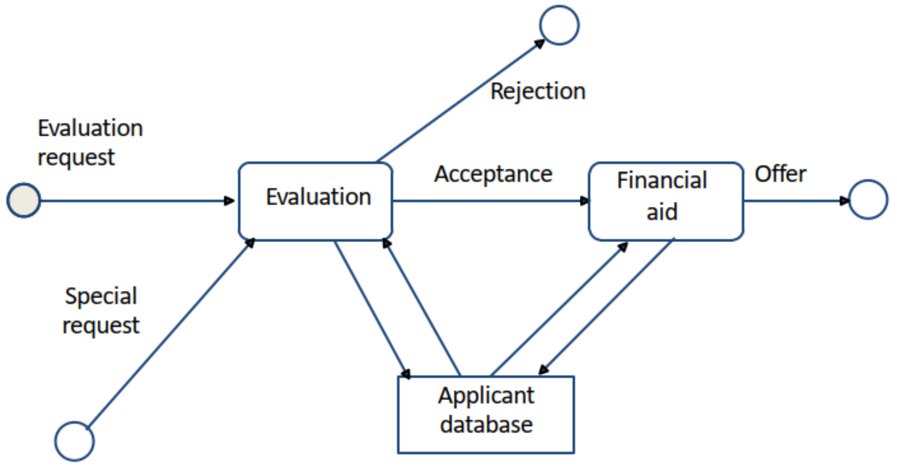
  - Shows the flow, but where is the data stored? Is there supporting information?

- Example: ***Assemble Application***



  - Does this model cover all situations? Are there special cases?\

- Example: ***Process Completed Application***



  - The requirements will need a description of the decision-making process.

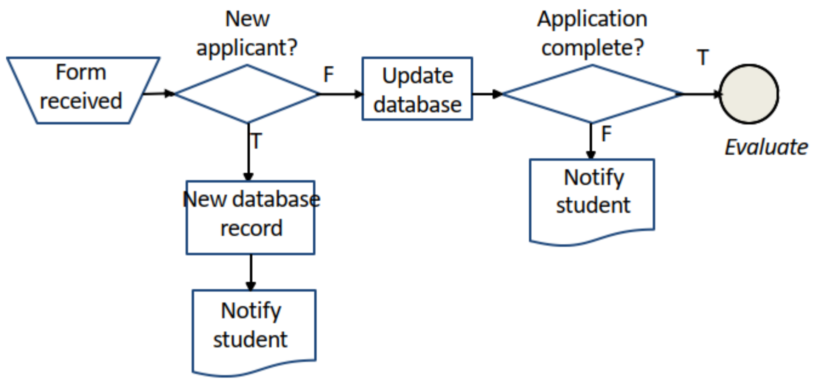
Decision Table Model

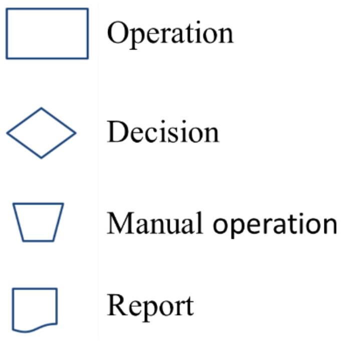
- University Admission Decision
- Each column is a separate decision case. The columns are processed from left to right.
- Note that the rules are specific and testable.

SAT > S1	T	F	F	F	F	F
GPA > G1	-	T	F	F	F	F
SAT between S1 and S2	-	-	T	T	F	F
GPA between G1 and G2	-	-	T	F	T	F
Accept	*X*	*X*	*X*			
Reject				*X*	*X*	*X*

Flowchart Model

- An informal modeling technique to show the logic of part of a system and paths that data takes through a system.
- Example: ***University Admissions Assemble Application***





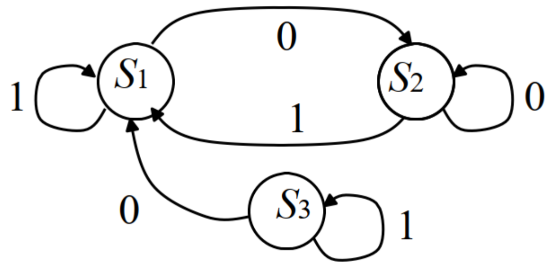
- Compare this example, which shows the logic, with the dataflow model, which shows the flow of data.

Modeling Tools: Pseudo-code

- The notation used for pseudo-code can be informal, or a standard used by a software development organization, or based on a regular programming language. What matters is that its interpretation is understood by everybody involved.
- Example: University Admission Decision

```
Admin_decision (application)
if application.SAT == null then error (incomplete)
if application.SAT > S1 then accept (application)
else if application.GPA > G1 then accept (application)
else if application.SAT > S2 and application.GPA > G2
        then accept (application)
else reject(application)
```

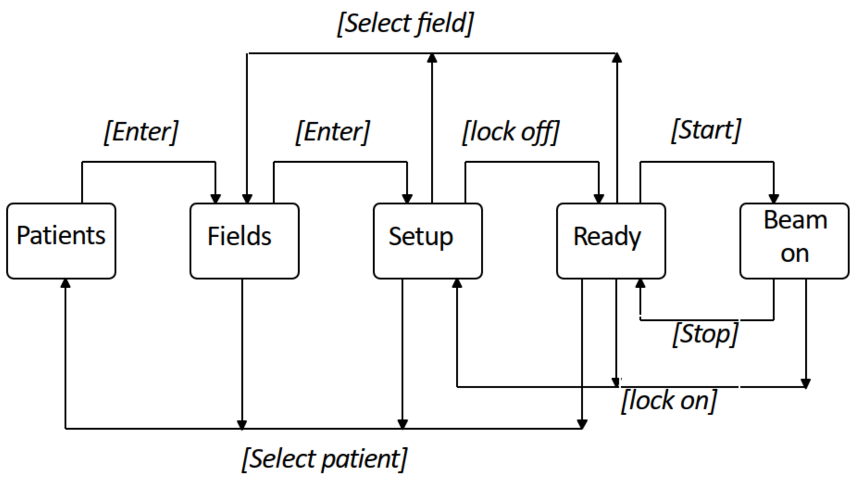
Modeling Tools: Transition Diagrams



- A system is modeled as a set of states, Si.
- A transition is a change from one state to another.
- The occurrence of a condition, Ci, causes the transition from one state to another.
- Transition function: $f(S_i, C_j) = S_k$

Finite State Machine Model, Therapy Control Console

- Example: Radiation Therapy Control Console
- You are developing requirements for the operator's control console. In an interview, the client describes the procedures that the operator must follow when operating the machine.
- You use a finite state machine model to specify the procedures.
- This shows the client that you understand the requirements and specifies the procedures for the developers.
- The client provides the following rough scenario.
- The set up is carried out before the patient is made ready. The operator selects the patient information from a database. This provides a list of radiation fields that are approved for this patient. The operator selects the first field. This completes the set up.
- The patient is now made ready. The lock is taken off the machine and the doses with this field are applied. The operator then returns to the field selection and chooses another field.

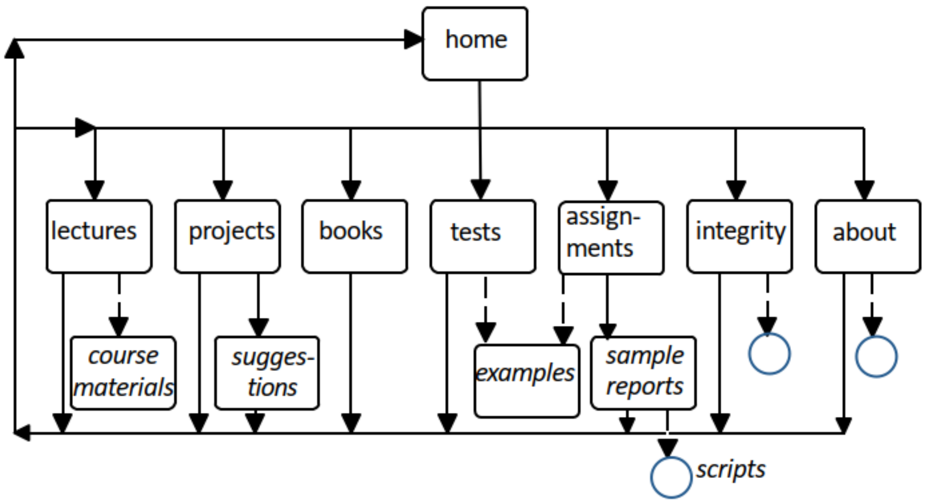


Finite State Machine Model State Transition Table

	Select Patient	Select Field	Enter	Lock of	Start	Stop	Lock on
Patients	__	__	Fields	__	__	__	__
Fields	Patients	__	Setup	__	__	__	__
Setup	Patients	Fields	__	Ready	__	__	__
Ready	Patients	__	__	Beam on	__	__	Setup
Beam on	__	__	__	__	__	Ready	Setup

This table can be used for requirements definition, program design, and acceptance testing.

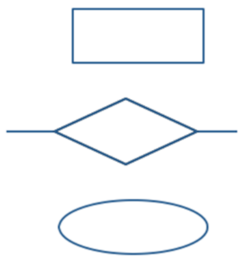
Finite State Machine Model



Example: ***Department Website***

Entity-Relation Model

- A requirements and design methodology for relational databases
  - A database of entities and relations
  - Tools for displaying and manipulating entity-relation diagrams
  - Tools for manipulating the database (e.g., as input to database design)
- Entity-relationship models can be used both for requirements specification and for the design specification.

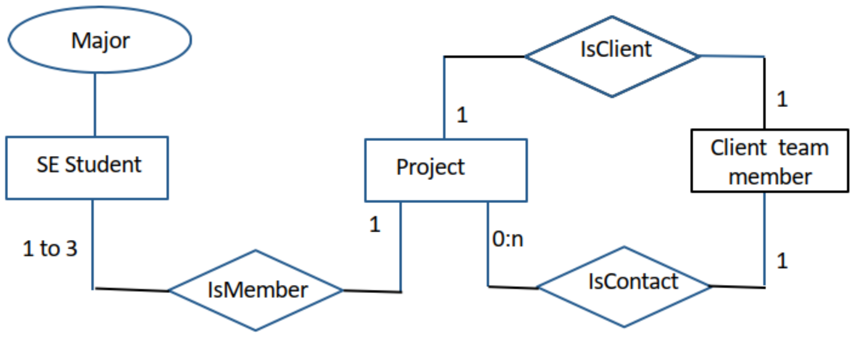


An entity (noun)

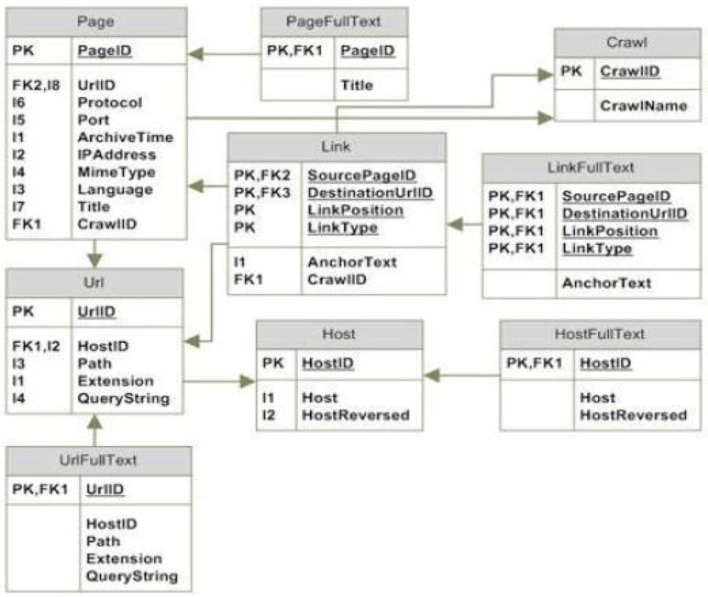
A relation between entities (verb)

An entity or relation attribute

Example: ***SE Project***



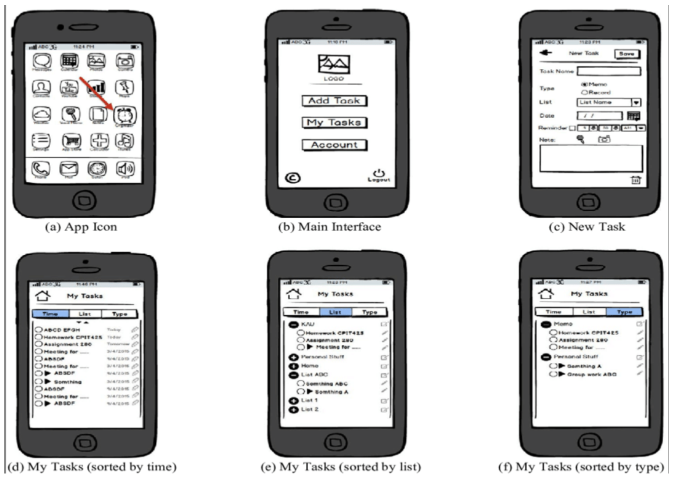
Example: ***Database Schema for Web Data***



Notation: Each table represents an entity. Each arrow represents a relation

Prototyping Requirements

- Rapid prototyping is the most comprehensive of all modeling methods
- A method for specifying requirements by building a system that demonstrates the functionality of key parts of the required system
- Particularly valuable for user interfaces


(a) App Icon
(b) Main Interface
(c) New Task
(d) My Tasks (sorted by time)
(e) My Tasks (sorted by list)
(f) My Tasks (sorted by type)

Class 3: User Experience

The Importance of the User Experience

- A computer system is only as good as the experience it provides to its users. If a system is hard to use:
    - users may fail to find important results or misinterpret what they do find.
    - users may give up in disgust.
- Appropriate functionality, easy navigation, elegant design, and fast response times make a measurable difference to a system's effectiveness.
- Usability is more than user interface design.
- Good support for users is more than a cosmetic flourish.
- Developing good user interfaces needs skill and time.

Terminology

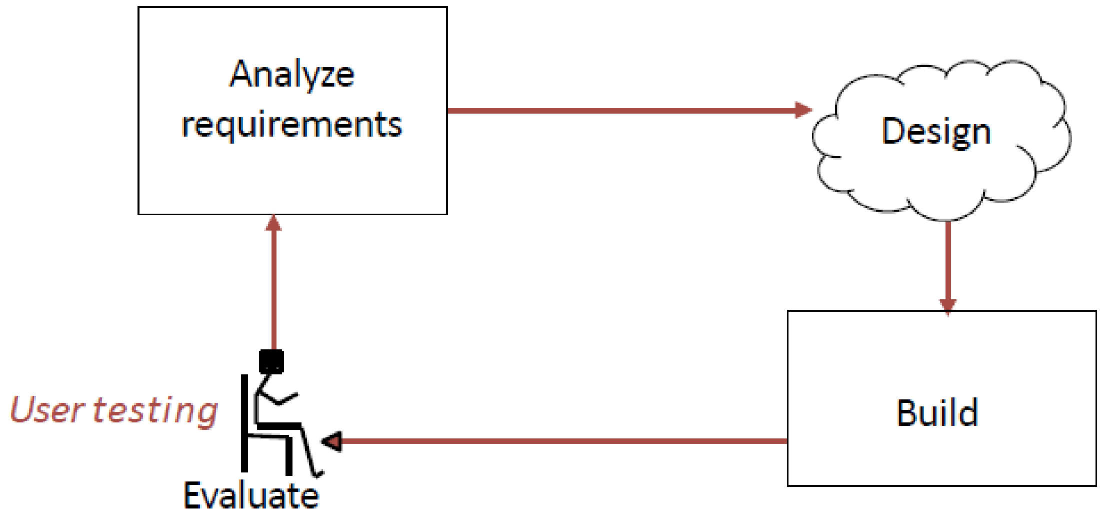
- The user experience is the total of all the factors that contribute to the usability (or otherwise) of a computer and its systems.
- Human Computer Interaction (HCI) is the academic discipline that studies how people interact with computers.

Development Processes for User Interfaces

- It is almost impossible to specify an interactive or graphical interface in a textual document.

  - Requirement benefit from sketches, comparison with existing systems, *etc*.
  - Designs should include graphical elements and benefit from a mock-up or other form of prototype.
  - User interfaces must be tested with users. Expect to change the requirements and design as the result of testing.
  - Schedules should include user testing and time to make changes.
- Whatever process you use to develop a software system, the development of the user interface is always iterative.

The Analyze/Design/Build/Evaluate Loop



Tools for Usability Requirements and Evaluation

	Initial	Mock-up	Prototype	Production
Client's opinions	√	√	√	
Competitive analysis	√			
Expert opinion	√	√	√	
Focus groups	√	√		
Observing users		√	√	√
Measurements			√	√

Tools for Usability Requirements: Focus Group

- A focus group is a group interview
  - Interviewer
- Potential users

    - Typically 5 to 12
    - Similar characteristics (e.g., same viewpoint)
- Structured set of questions
    - May show mock-ups
    - Group discussions
- Repeated with contrasting user groups

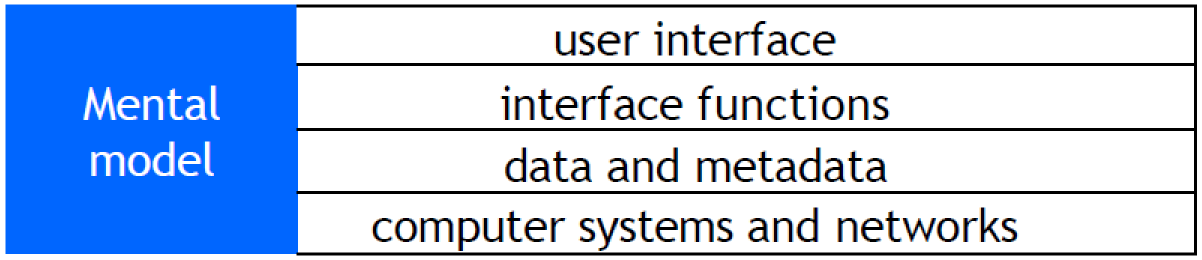
Accessibility Requirements

- Software designers must be prepared to users with disabilities (e.g., poor eyesight, lack of hearing, poor manual dexterity), or limited knowledge of English, *etc*.
- Requirements about accessibility are most likely to arise in the user interface.
- You may have a legal requirement to support people with disabilities.

Equipment Requirements

- There may also be requirements to support computers with poor performance, limited screen sizes, bad network connections, etc.
- Be explicit about the equipment assumptions that you make and how to handle failures. Do user testing with both good and bad equipment.
- Example: MacMail has a requirement that operations terminate cleanly if the network connection is lost, but its behavior is erratic if the network connection becomes extremely slow, e.g., it will not quit.

Design from a System Viewpoint



Mental Models

- A mental model is what a user thinks is true about a system, not necessarily what is actually true.
- A mental model should be similar in structure to the system that is represented.
- A mental model allows a user to predict the results of his/her actions.
- A mental model is simpler than the represented system. It includes only enough information to allow reasonable predictions.
- A mental model is also called a conceptual model.

- The mental model is the user's model of what the system provides.
- The computer model may be quite different from the user's mental model.
- Example: the desktop metaphor
    - mental model — one vast collection of pages, which are searched on request.
    - computer model — a central index, which is searched on request.
- Example: web search
    - mental model — one vast collection of pages, which are searched on request.
    - computer model — a central index, which is searched on request.

Mental Models vs. Computer Model

The mental model is that the photograph is embedded in the text of the document…

"All the News That's Fit to Print"

The New York Times

Late Edition

VOL. CLXVI ... No. 57,432 NEW YORK, WEDNESDAY, NOVEMBER 30, 2016 $2.50

Cabinet Choice Wrote Remake Of Health Law

Plan Could Aid Doctors but Limit Consumers

MOVIE FINANCIER IS SAID TO BE PICK FOR TREASURY JOB

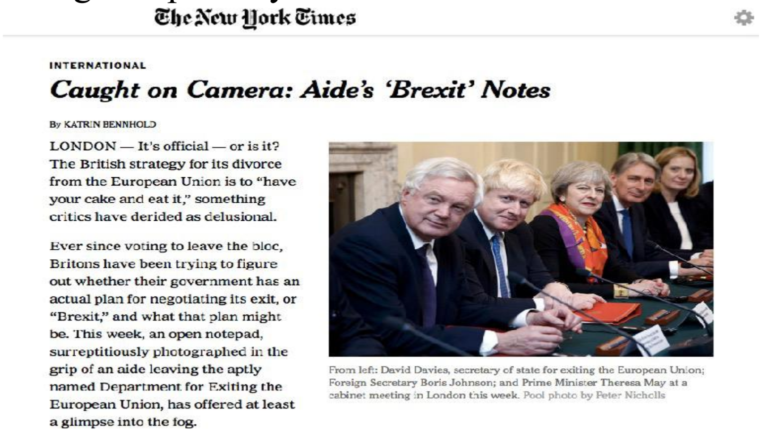
ROOTS ON WALL STREET

Trump Elevates Loyalist From Campaign to Pivotal Post

JOY, THEN GRIEF, OVER BRAZIL TEAM

Despite Climate Vow, China Scrambles for Coal

but in the computer model the photograph is an independent file, which could be changed separately.



The New York Times

INTERNATIONAL

Caught on Camera: Aide's 'Brexit' Notes

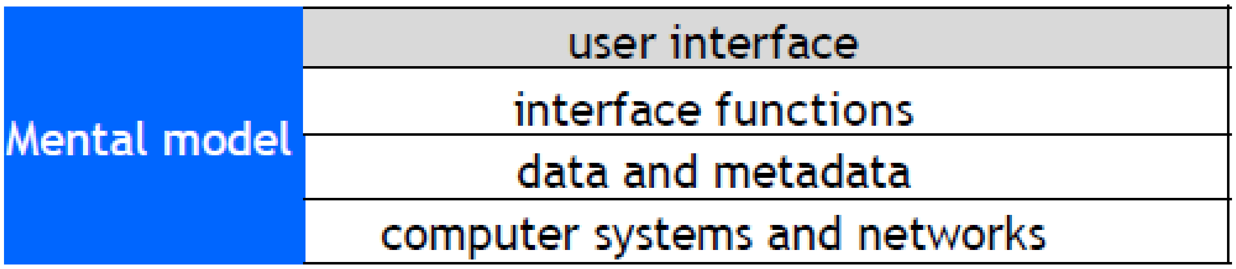
By KATRIN BENNHOLD

LONDON — It's official — or is it? The British strategy for its divorce from the European Union is to "have your cake and eat it," something critics have derided as delusional.

Ever since voting to leave the bloc, Britons have been trying to figure out whether their government has an actual plan for negotiating its exit, or "Brexit," and what that plan might be. This week, an open notepad, surreptitiously photographed in the grip of an aide leaving the aptly named Department for Exiting the European Union, has offered at least a glimpse into the fog.

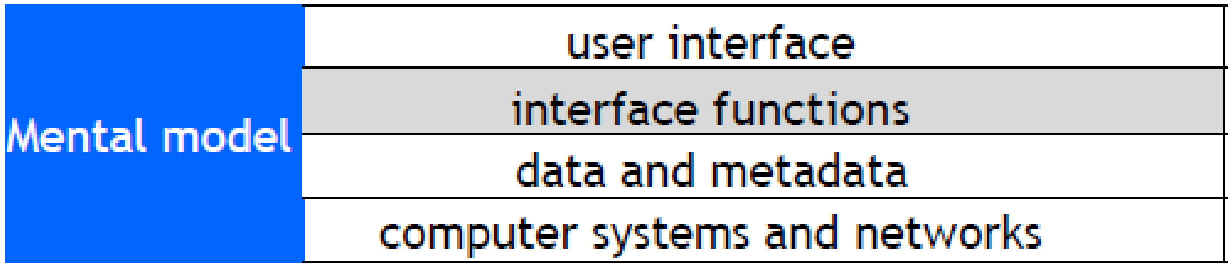
From left: David Davies, secretary of state for exiting the European Union; Foreign Secretary Boris Johnson; and Prime Minister Theresa May at a cabinet meeting in London this week. Pool photo by Peter Nicholls

User Interface Design



- The user interface is the appearance on the screen and the actual manipulation by the user.
  - Fonts, colors, logos, keyboard controls, menus, buttons
  - Mouse control, touch screen, or keyboard control
  - Conventions (e.g., "back", "help")
- Example: to leave full screen
  - Keyboard: escape key, control-F Mouse/touch:
- Examples of design choices
  - Screen space utilization in Adobe Reader.
  - Number of snippets per page in web search.

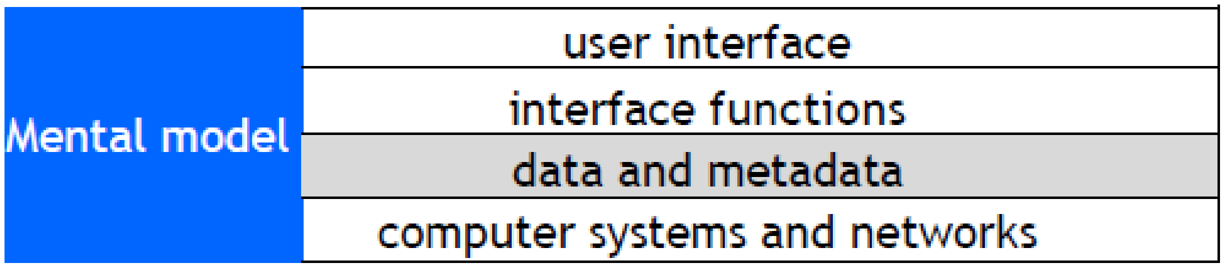
Interface Functions



- The interface functions determine the actions that are available to the user:
  - Select part of an object
  - Search a list or sort the results.
  - View help information
  - Manipulate objects on a screen.
  - Pan or zoom.
- There may be alternative user interface designs for the same interface functions, for example:
  - Different versions of the MS Windows desktop have most of the same interface functions, but different user interface designs.

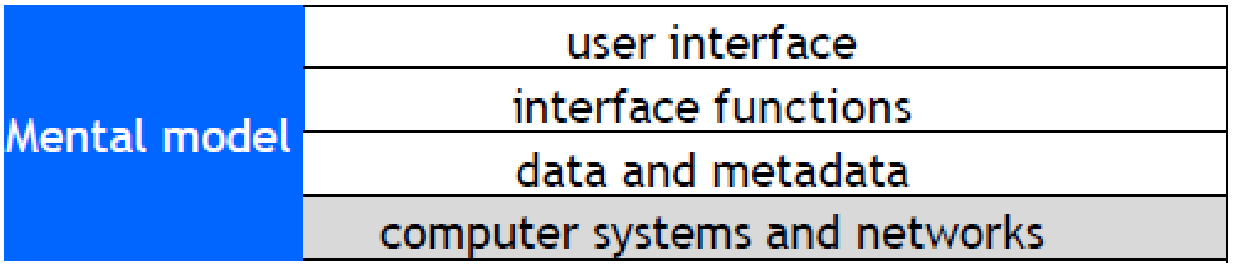
    - Applications that run on both Windows and Macintosh computers support a one-button mouse (Macintosh) o a two-button mouse (Windows).

Data and Metadata



- The interface functions and the interface design provide an interface to the data and metadata stored by the computer system.
- The desktop metaphor has the concept of associating a file with an application. This requires a file type to be stored with each file:
    - extension to filename (e.g., .txt, .pdf)
- Inexperienced clients sometimes ask for interface features that require additional data or metadata.

Computer Systems and Networks



- The performance, reliability and predictability of computer systems and networks are crucial factors.
- Examples
    - Instantaneous response time is essential for mouse tracking.
    - Response time for transactions may determine the action taken, e.g., approve credit card if no reply within five seconds.
- Response time requirements
    - 0.1 sec – the user feels that the system is reacting instantaneously.
    - 1 sec – the user will notice the delay, but his/her flow of thought stays uninterrupted.

    - 10 sec – the limit for keeping the user's attention focused on the dialogue.
- As computer systems improve, users have got more demanding. A response time that is good enough today, may not be good enough five years from now.

Principles of User Interface Design

- User interface design is partly an art, but there are general principles
    - Consistency -- in appearance, controls, and function.
    - Feedback -- what is the computer system doing? Why does the user see certain results?
    - Users should be able to interrupt or reverse actions.
    - Error handling should be simple and easy to comprehend.
    - Skilled users should be offered shortcuts; beginners should have simple, well-defined options.
    - The user should feel in control.

- Avoid the Fancy

- Simple is Good

Interface Design: Menus

- Easy for users to learn and use
- Certain categories of error are avoided
- Enables context-sensitive help
- Major difficulty is structure of large number of choices
    - Scrolling menus (e.g., states of USA)
    - Hierarchical
    - Associated control panels
    - Menus plus command line
- Users prefer broad and shallow to deep menu systems

Help System Design

- Help system design is difficult
    - Must prototype with mixed users
    - Must have many routes to same information
- Categories of help:
    - Overview and general information
    - Specific or context information
    - Tutorials (general)
    - Cook books and wizards
    - Emergency ("I am in trouble ...")

- Help systems need experienced designers. Schedule plenty of time for development and user testing.

Information Presentation

- Simple is often better than fancy
- Text
    - precise, unambiguous
    - fast to compute and transmit
- Graphical interface
    - simple to comprehend/learn, but icons can be difficult to recognize
    - uses of color
    - variations show different cases

Command Line Interfaces

- Problems with graphical interfaces
    - Not suitable for some complex interactions
    - May be slow for skilled users
    - Difficult to build scripts
    - Only suitable for human users
- User interacts with computer by typing commands (e.g., Linux shell script)
    - Allows complex instructions to be given to computer
    - Facilitates formal methods of specification & implementation
    - Skilled users can input commands quickly
    - Unless very simple, requires learning or training
    - Can be adapted for people with disabilities
    - Can be multi-lingual
    - Suitable for scripting/ non-human clients

Device-Aware Interfaces

- Interfaces must take into account physical constraints of computers and networks:
    - How does a desk-top computer differ from a laptop?
    - What is special about a smartphone?
    - How do you make use of a touch-sensitive screen?
    - What works well with a digital camera?
- Constraints that the interface must allow for:
    - performance of device (e.g., fast or slow graphics)
    - limited form factor (e.g., small display, no keyboard)
    - connectivity (e.g., intermittent)

Networks

- Operations that transfer data over the network have unpredictable response times and are subject to delay.
    - Large data transfers should run asynchronously in a separate thread.
    - Provide visual feedback to indicate that the operation is in progress.

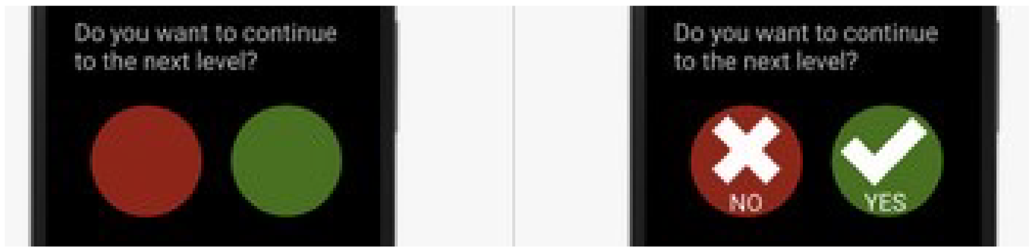
    - Provide a way for users to cancel long running data transfers

Smartphones and Tablets: Screen Size

- Smartphones and tablets have small screens. Every small area is important.
    - There is big difference in screen size between a small smartphone and a large tablet.
    - There may be different layouts or even different version for different screens.
    - Apps need to have different layouts for portrait and landscape.
- Apps need to be tested with a full range of screen sizes and orientations.
- Some apps may be optimized for a large screen.
- Use a simulator to see how your app looks on various devices.

Smartphones: Accessibility

- Smartphones have small displays and small virtual keyboards. Some apps rely on speech or other sound signals.
    - People with poor eyesight, color blindness, hearing loss, or clumsy fingers may have difficulty using your applications.
    - Android and iOS provide numerous accessibility features and provide online advice about how to build accessible apps.
    - For your user testing try to find people who do not have perfect eyesight, hearing, etc. Have testers of various ages. Older people are often less able to use touch sensitive screens.


Do you want to continue to the next level?
Do you want to continue to the next level?
NO
YES

Screen Size: Responsive Web Design

- Responsive web design adjusts how web pages are viewed based on the size of the screen, or other characteristics of the device or browser used to view the page.
- Media queries in CSS (Cascading Style Sheet) allow the page to use different CSS style rules based on characteristics of the device the site is being displayed on, e.g., the width of the browser window.
- For example. You develop a web site using a laptop computer. How will it look on a smartphone?

Responsive Web Design

- Browsers such as Chrome, Firefox, and Safari have options that allow you to see how your site would appear on other devices.
- The ideal:
    - A single web site adapts to any device by using a mix of flexible grids and layouts, and careful use of CSS media queries.
- In practice:
    - Mobile devices, such as smartphones, are so different from regular computers that it is extremely difficult to create a web site that works well on all devices.
- For example:
    - Less content can be displayed on a small screen than on a large screen.
    - Smartphones are usually used in portrait; computer screens are landscape.
    - A touch screen has different characteristics from a mouse, e.g., the user cannot hover over a menu item.
    - A virtual keyboard needs screen space.

- Major web sites have several versions of the site for different devices.
- They identify the user's device and automatically provide the version designed for that type of device.
- This Safari example shows the New York Times mobile version of the web site on a small iPhone.
- The next slide, from Firefox, shows the same page on a laptop computer.

User Interface Design: Navigation

- Getting the navigation right is central to the user experience. Applications consist of one or more pages (web) or screens (mobile apps).
  - Web: loading a new page uses the network and is therefore slow
  - Smartphone or tablet: loading a new screen is usually instantaneous
- The first step in user interface design is to choose what pages or screens to use.
  - What functionality is provided on each screen?
  - How are the screens organized?
  - How does the user know what screens are available?
  - How does the user move between screens?

User Interface Design: Organization of Pages and Screens

- The organization of the pages must match the user's mental model. Keep it simple.
- Organization of pages in a web application.
  - The basic building block of the web is a hyperlink. This allows any page to link to any other page. This is flexible, but can lead to confusing applications.
  - Many web sites use a hierarchical tree structure.
  - When a user leaves a page, the state is lost unless explicitly saved.

Organization of screens in a mobile app

- Both Android and iOS encourage the use of a stack based architecture.
- When a user leave a screen, the state is pushed onto a stack and is available when the screen is next used.
- Indicate to the user what pages are available and how to reach them.

Navigation: User Testing

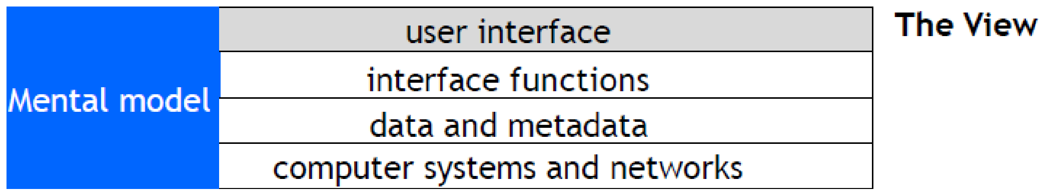
- Before building an application, test the navigation.
  - Create simple mock-ups, e.g., rough sketches.
  - Create a large number of simple scenarios. They should show all the main paths through the application, including user mistakes, system problems, *etc*.
  - Step through the scenarios with the client and potential users.
- Later in the development process, these scenarios can be used for user interface design, program design, and all forms of testing.
- It is much easier to make changes in the basic navigation before beginning the detailed user interface design and implementation.

Mobile Apps: Development Environments

- App development is complex. Most apps are built using development environments such as Android Studio and Apple's Xcode.
- There are good tutorials and good documentation online. I strongly recommend that you work through a tutorial before attempting to build a real app.
- If you are new to Android or iOS development allow plenty of learning time.
- For your project try to have several team members with previous experience of the development environment that you are using.
- These environments are moderately flexible for user interface design. They support a variety of ways to navigate, user interface objects, etc. But they have limitations (e.g., iOS does not have a radio button).
- To build a good user experience, understand and adopt the style of the development environment.
    - Users will be familiar with the style of interface from other apps.
    - The interface objects are well designed with good technical implementation.
    - The objects are intended to be used on a variety of devices.
    - When new versions of the operating systems are released, the designs will need a few modifications.

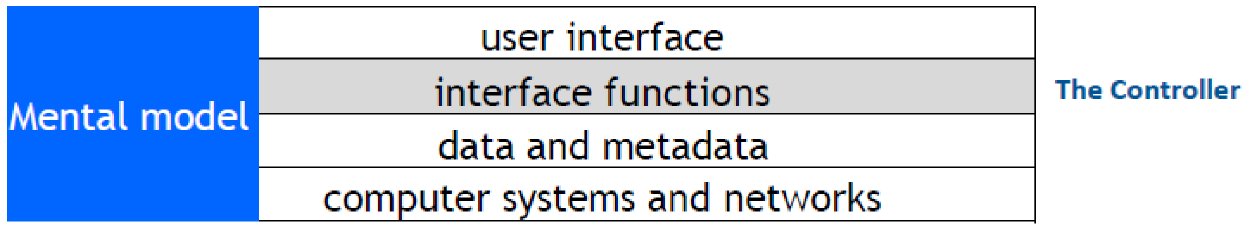
Model-View-Controller: The View

- Most modern development environments use one of the many variations of the Model-View-Controller architecture.



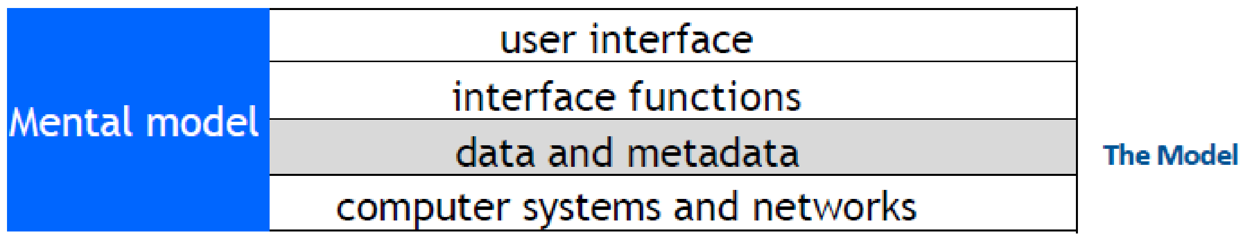
- In the Model-View-Controller architecture, the user interface is called the View.
    - Many of the interface functions are implemented by user interface widgets that are provided by the development environment.
    - The development environment may also provide recommended styles for the user interface, e.g., fonts, graphical elements.

Model-View-Controller: The Controller



- In the Model-View-Controller architecture, the interface functions are invoked by the Controller.
- The Controller manages the flow of control of the whole user interface and connects the View to the Model.

Model-View-Controller: The Model



- In the Model-View-Controller architecture, the data and metadata are maintained by the Model.
- The Model also manages the logic of all parts of the system that are not part of the user interface.

Mobile Apps: the Design Challenge

- How do you design a user interface with no instructions, no user manual, no training?
- Look: Characteristics of the appearance that convey information.
- Feel: Interaction techniques that are intuitive and provide satisfactory experience.
- Metaphors and mental models: Mental models and metaphors. But there may not be an intuitive model.
- Navigation rules: How to move among data, functions, and activities in a large space.
- Conventions: Familiar aspects that do not need extra training – good for users, good for designers, e.g., scroll bars, buttons, gestures, help systems, sliders.

Mobile Apps: Development Environments

- Development environments include standard classes that provide templates for navigation and the layout of screens.
- Designs that incorporate these classes can use services, APIs, etc. that the operating environment provides.
- This example uses an iOS TableViewController.
- Because the length of tables can be very long, iOS implements a TableViewController in a manner that makes efficient use of storage and processing power.

Mobile Apps: Conventions

- Development environments encourage consistent apps.
- You may never have used this app, but look at how many aspects feel familiar.

Design Tensions in Web and Mobile Applications

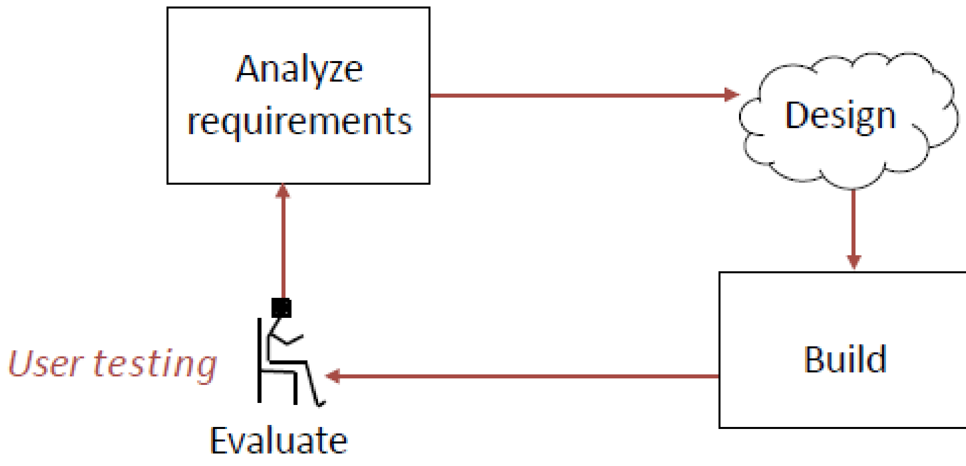
- Designers wish to control what the user sees, but users wish to configure their own environments.
    - Client computers and network connections vary greatly in capacity.
    - Client software may run on various operating systems, which may not be the current version.
    - Accessibility requires that designers do not take control of parameters such as font size.
- Be explicit about the assumptions you make about the user's computer, web browser, etc. In using style sheets, such as CSS, avoid over-riding user preferences.

Usability and Cost

- User interface development may be a major part of a software development project.
    - Good usability may be expensive in hardware or special software development.
    - Costs are multiplied if a user interface has to be used on different computers or migrate to different versions of systems.
- Design users interfaces that can be built with standard tools:
    - Web browsers provide a general-purpose user interface where others maintain the user interface software.
    - Use the standard classes that the development environment provides.

The Analyze/Design/Build/Evaluate Loop

- Whenever possible, the design and evaluation should be done by different people.

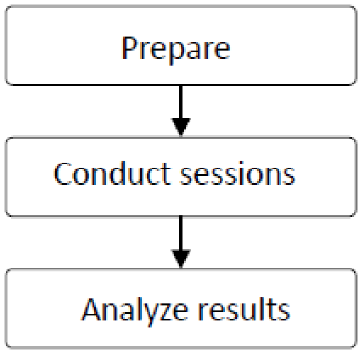


Evaluation

- If your system has users, the schedule should include time for user testing and time to make changes after the user testing is completed.
- When to do evaluation
    - Iterative improvements during development.
    - Making sure that a system is usable before launching it.
    - Iterative improvements after launch.
- Methods of evaluation
    - Empirical evaluation with users (user testing)
    - Measurements on operational systems
    - Analytical evaluation: without users
- How do you measure usability? Usability comprises the following aspects:
- Effectiveness
    - The accuracy and completeness with which users achieve certain goals
    - Measures: quality of solution, error rates
- Efficiency
    - The relation between the effectiveness and the resources expended in achieving them
    - Measures: task completion time, learning time, number of clicks
- Satisfaction
    - The users' comfort with and positive attitudes towards the use of the system

Evaluation with Users

- Stages of evaluation with users:



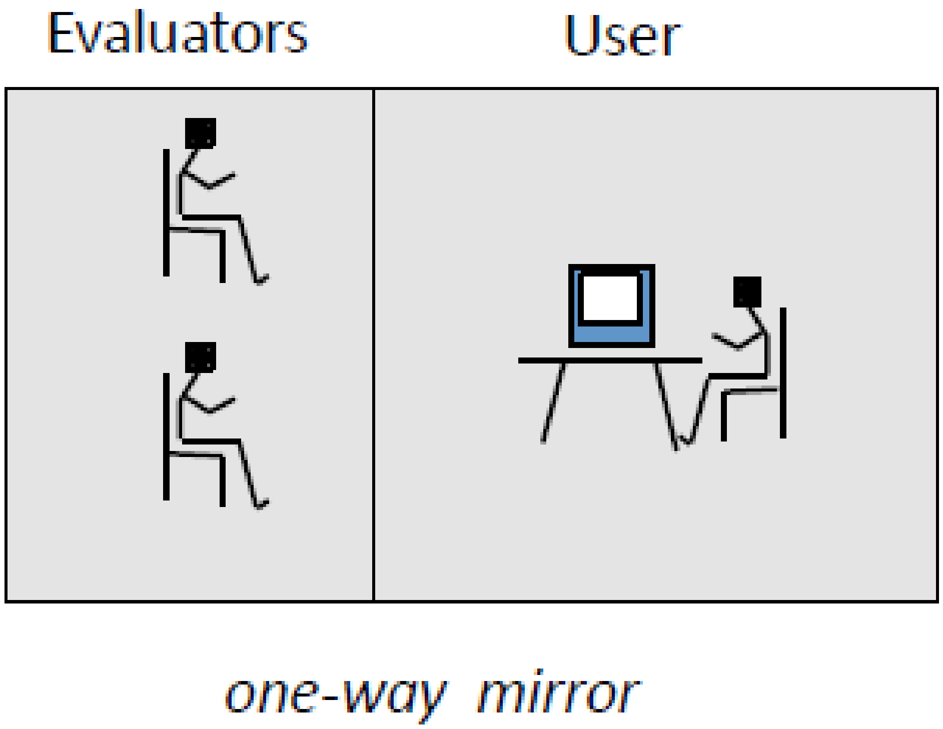
- User testing is time-consuming, expensive, and essential

Evaluation with Users: Preparation

- Determine goals of the usability testing
    - “Can a user find the required information in no more than 2 minutes?”
- Write the user tasks
    - “Given a new customer application form, add a new customer to the customer database.”
- Recruit participants
    - Use the descriptions of users from the requirements phase to determine categories of potential users and user tasks

Usability Laboratory

- Concept: monitor users while they use system



Evaluation with Users: Sessions

- Conduct the session:
    - Usability Lab
    - Simulated working environment.
- Observe the user:
    - Human observer(s)
    - Video camera
    - Audio recording
- Inquire satisfaction data.

Evaluation: Number of Users

- A great deal can be learned from user testing with a small number of users, even as few as five people.
    - Try to find different types of user (young/old, experienced/beginners, etc.).
    - Prepare carefully.
    - Combine structured tests with free form interviews.
    - Have at least two evaluators for every test.

Results Analysis

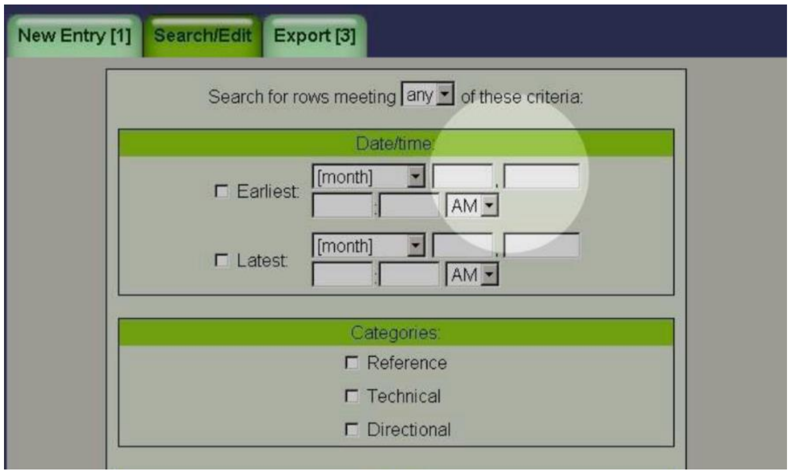
- Test the system, not the users
- Respect the data and users' responses. Do not make excuses for designs that failed.
- If possible, use statistical summaries.
- Pay close attention to areas where users:
    - were frustrated
    - took a long time
    - could not complete tasks
- Note aspects of the design that worked and make sure they are incorporated in the final product.

Project: Methodology

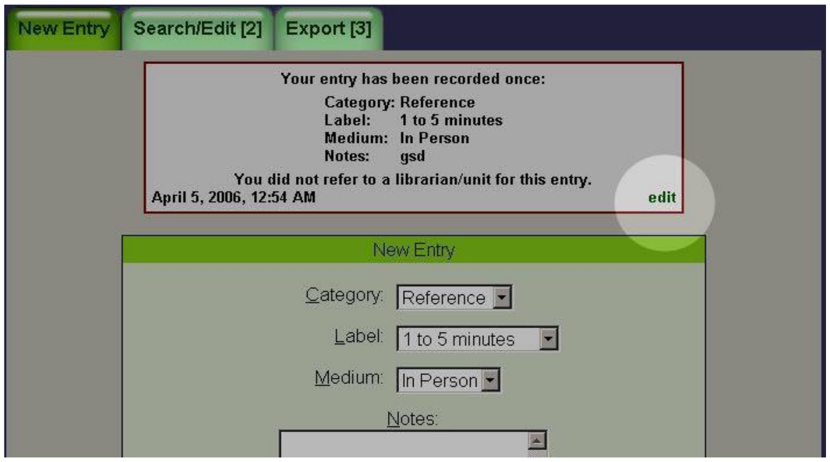
- The next few slides are from a presentation
- How we're user testing:
    - One-on-one, 30-45 min user tests with staff levels Specific tasks to complete
    - No prior demonstration or training
    - Pre-planned questions designed to stimulate feedback
    - Emphasis on testing system, not the stakeholder!
    - Standardized tasks /questions among all testers
- Types of questions we asked:
    - Which labels, keywords were confusing? What was the hardest task?
    - What did you like, that should not be changed? If you were us, what would you change?
    - How does this system compare to your paper based system
    - How useful do you find the new report layout? (admin)
    - Do you have any other comments or questions about the system? (open ended)

A Project: Results

- What we've found: Issue #1, Search Form Confusion!

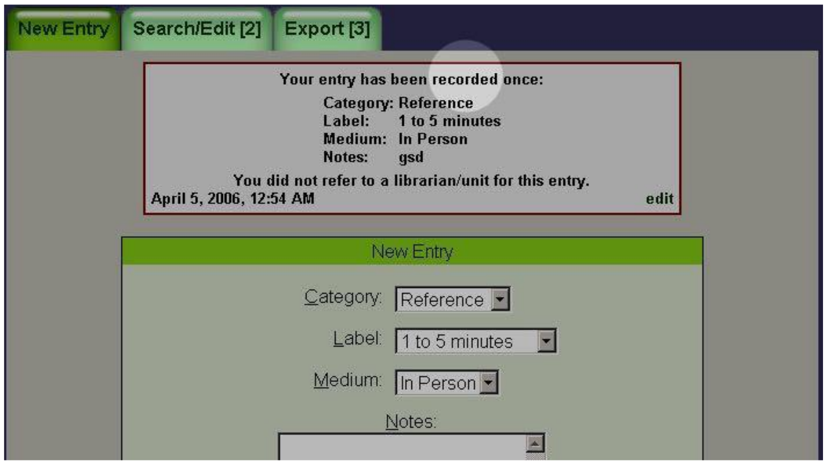


- What we've found: Issue #2, Inconspicuous Edit/ Confirmations!



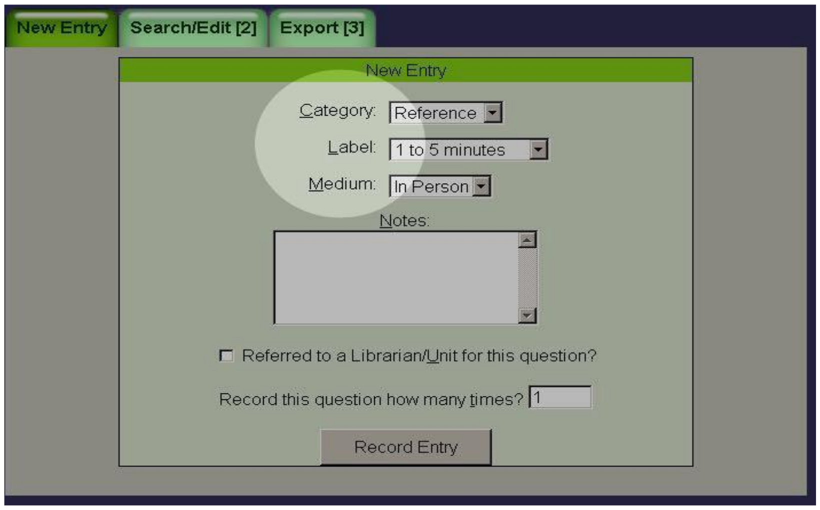
Your entry has been recorded once:
Category: Reference
Label: 1 to 5 minutes
Medium: In Person
Notes: gsd
You did not refer to a librarian/unit for this entry.
April 5, 2006, 12:54 AM edit

- What we've found: Issue #3, Confirmation Terms



Your entry has been recorded once:
Category: Reference
Label: 1 to 5 minutes
Medium: In Person
Notes: gsd
You did not refer to a librarian/unit for this entry.
April 5, 2006, 12:54 AM edit

- What we've found: Issue #4, Entry Semantics



New Entry

Category: Reference

Label: 1 to 5 minutes

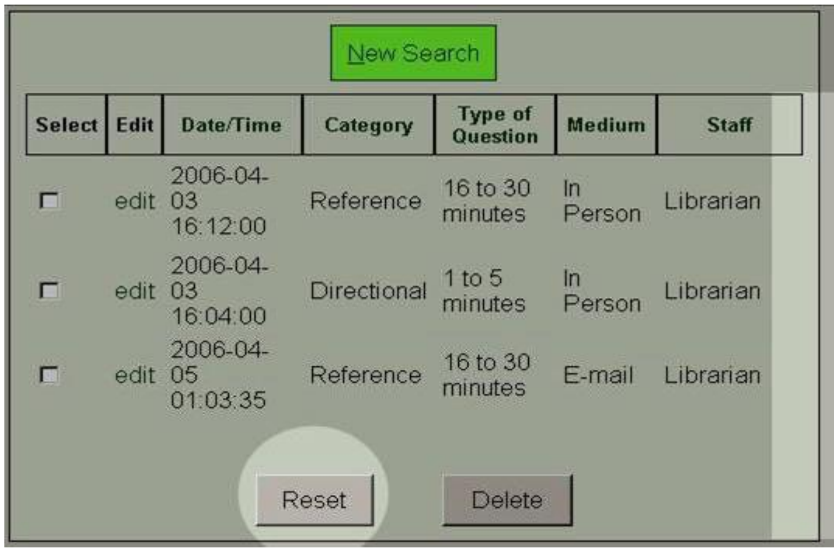
Medium: In Person

Notes:

☐ Referred to a Librarian/Unit for this question?

Record this question how many times? 1

- What we've found: #5, Search Results Disambiguation & Semantics



Select	Edit	Date/Time	Category	Type of Question	Medium	Staff
☐	edit	2006-04-03 16:12:00	Reference	16 to 30 minutes	In Person	Librarian
☐	edit	2006-04-03 16:04:00	Directional	1 to 5 minutes	In Person	Librarian
☐	edit	2006-04-05 01:03:35	Reference	16 to 30 minutes	E-mail	Librarian

Evaluation based on Measurement

- Basic concept: log events in the users' interactions with a system
- Examples from a Web system
  - Clicks (when, where on screen, etc.)
  - Navigation (from page to page)
  - Keystrokes (e.g., input typed on keyboard)
  - Use of help system
  - Errors May be used for statistical analysis or for detailed tracking of individual user.
- Analysis of system logs
  - Which user interface options were used?
  - When was the help system used?
  - What errors occurred and how often?
  - Which hyperlinks were followed (click through data)?
- Human feedback
  - Complaints and praise
  - Bug reports
  - Requests made to customer service

Refining the Design based on Evaluation

- Do not allow evaluators to become designers
- Designers are poor evaluators of their own work, but know the requirements, constraints, and context of the design:
  - Some user problems can be addressed with small changes
  - Some user problems require major changes
  - Some user requests (e.g., lots of options) are incompatible with other requests (e.g., simplicity)
- Designers and evaluators need to work as a team

Class 4a: Design

Design

- The requirements describe the function of a system as seen by the client.
- For a given set of requirements, the software development team must design a system that will meet those requirements.
- In practice requirements and design are interrelated. In particular, working on the design often clarifies the requirements. This feedback is a strength of the iterative and agile methods of software development.
- We have already looked at user interface design.
- The next few lectures look at the following aspects of design:

    - system architecture
    - program design
    - security
    - performance

Creativity and Design

- Software development is a craft.
- Software developers have a variety of tools that can be applied in different situations.
- Part of the art of software development is to select the appropriate tool for a given implementation.
- System and program design are a particularly creative part of software development, as are user interfaces. You hope that people will describe your designs as “elegant”, “easy to implement, test, and maintain.”
- Above all strive for simplicity. The aim is find simple ways to implement complex requirements.
- System architecture is the overall design of a system
    - Computers and networks (e.g., monolithic, distributed)
    - Interfaces and protocols (e.g., http, ODBC)
    - Databases (e.g., relational, distributed)
    - Security (e.g., smart card authentication)
    - Operations (e.g., backup, archiving, audit trails)
- At this stage of the development process, you should also be selecting:
    - Software environments (e.g., languages, database systems, class frameworks)
    - Testing frameworks

Models for System Architecture

- Our models for systems architecture are based on UML
- For every system, there is a choice of models
    - Choose the models that best model the system and are clearest to everybody.
- When developing a system, every diagram must have supporting specification.
    - The diagrams shows the relationships among parts of the system, but much, much more detail is needed to specify a system explicitly.

Subsystems

- A subsystem is a grouping of elements that form part of a system.

- Coupling is a measure of the dependencies between two subsystems. If two systems are strongly coupled, it is hard to modify one without modifying the other.
- Cohesion is a measure of dependencies within a subsystem. If a subsystem contains many closely related functions its cohesion is high.
- An ideal division of a complex system into subsystems has low coupling between subsystems and high cohesion within subsystems.

Component

- A component is a replaceable part of a system that conforms to and provides the realization of a set of interfaces.
- A component can be thought of as an implementation of a subsystem.
- UML definition of a component: "a distributable piece of implementation of a system, including software code (source, binary, or executable), but also including business documents, etc., in a human system."

Components as Replaceable Elements

- Components allow systems to be assembled from binary replaceable elements
- A component is bits not concepts
- A component can be replaced by any other component(s) that conforms to the interfaces
- A component is part of a system
- A component provides the realization of a set of interfaces

Components and Classes

- Classes represent logical abstractions. They have attributes (data) and operations (methods).
- Components have operations that are reachable only through interfaces.

Package

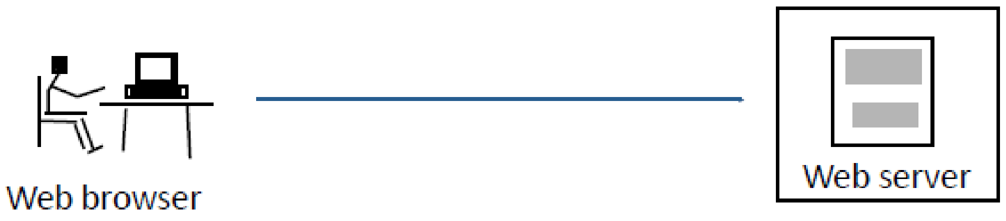
- A package is a general-purpose mechanism for organizing elements into groups.

Node

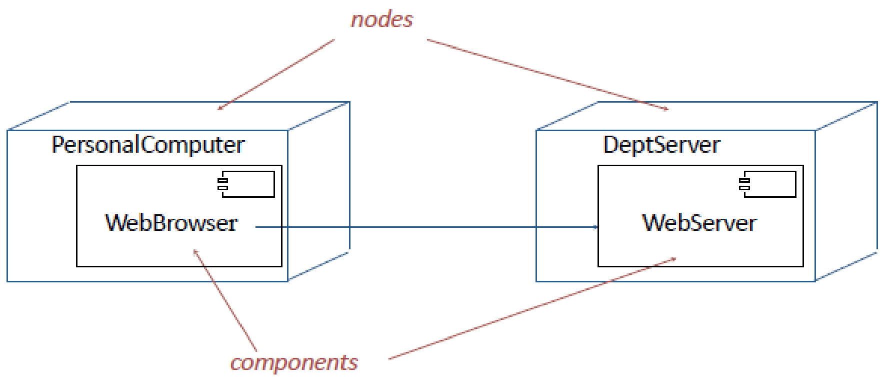
- A node is a physical element that exists at run time and provides a computational resource, e.g., a computer, a smartphone, a router.
- Components may live on nodes.

Example: Simple Web System

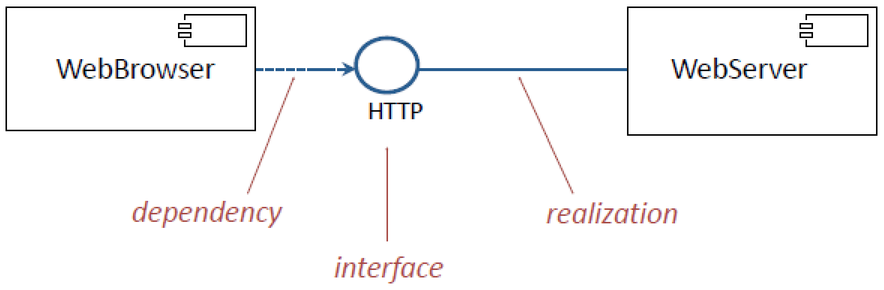
- Static pages from serve
- All interaction requires communication with server



Deployment Diagram

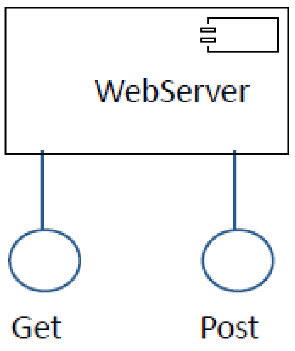


Component Diagram: Interfaces



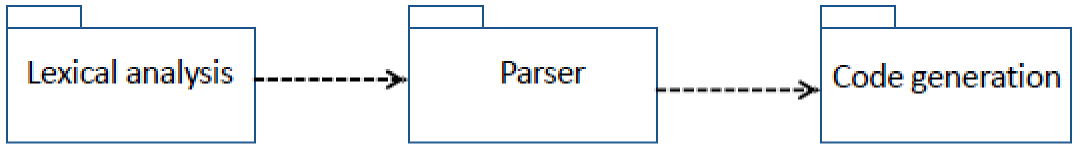
Application Programming Interface (API)

- An API is an interface that is realized by one or more components.



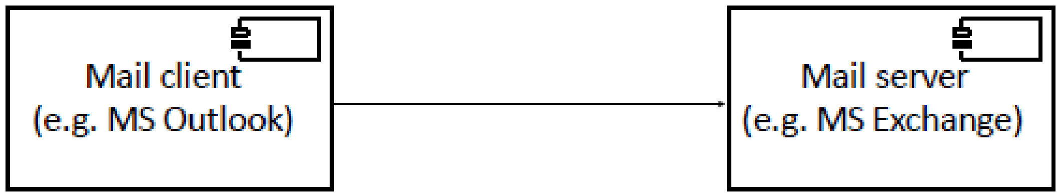
Architectural Style: Pipe

- Example: A three-pass compiler
- Output from one subsystem is the input to the next.



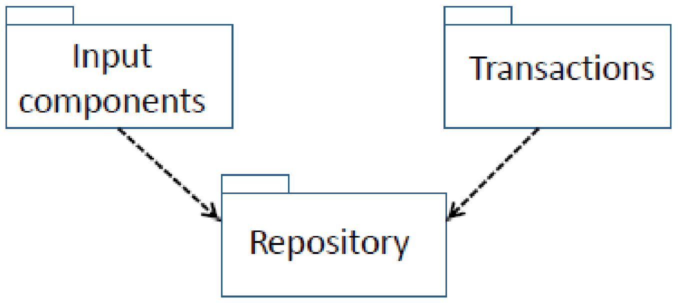
Architectural Style: Client/Server

- Example: A mail system
- The control flows in the client and the server are independent.
- Communication between client and server follows a protocol.
- In a peer-to-peer architecture, the same component acts as both a client and a server.



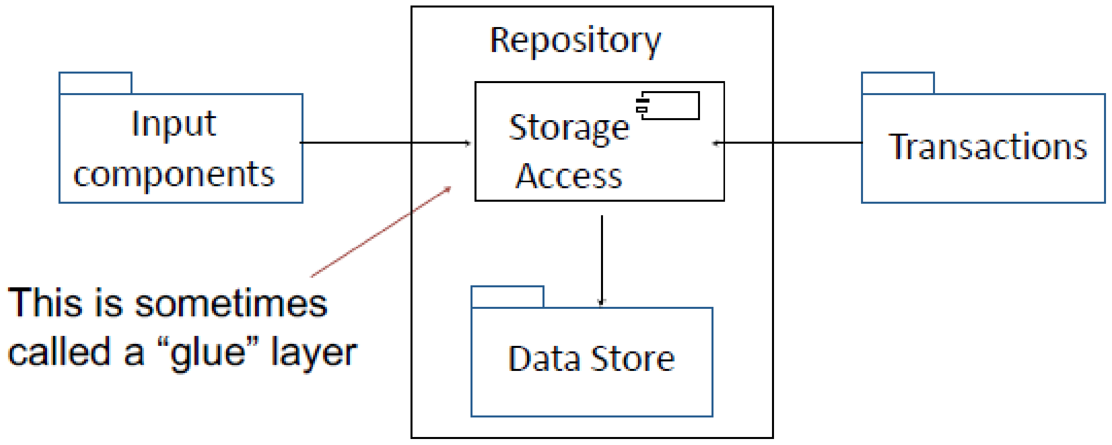
Architectural Style: Repository

- Advantages: Flexible architecture for data-intensive systems.
- Disadvantages: Difficult to modify repository since all other components are coupled to it.



Architectural Style: Repository with Storage Access Layer

- Advantages: Data Store subsystem can be changed without modifying any component except the Storage Access.

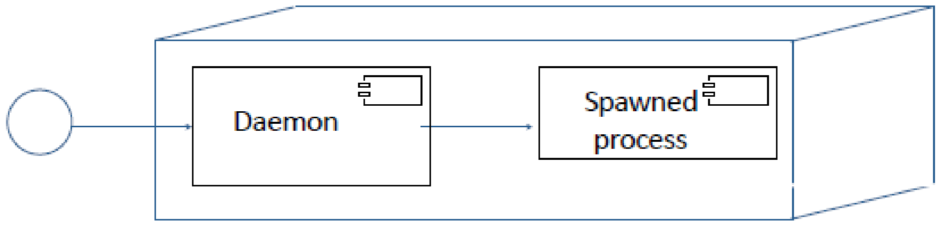


Time-Critical Systems

- A time-critical (real time) system is a software system whose correct functioning depends upon the results produced and the time at which they are produced.
- A hard real time system fails if the results are not produced within required time constraints e.g., a fly-by-wire control system for an airplane must respond within specified time limits
- A soft real time system is degraded if the results are not produced within required time constraints e.g., a network router is permitted to time out or lose a packet

Time Critical System: Architectural Style – Daemon

- A daemon is used when messages might arrive at closer intervals than the time to process them
- Example: Web server
    - The daemon listens at port 80
    - When a message arrives it: spawns a processes to handle the message returns to listening at port 80



Architectural Styles for Distributed Data

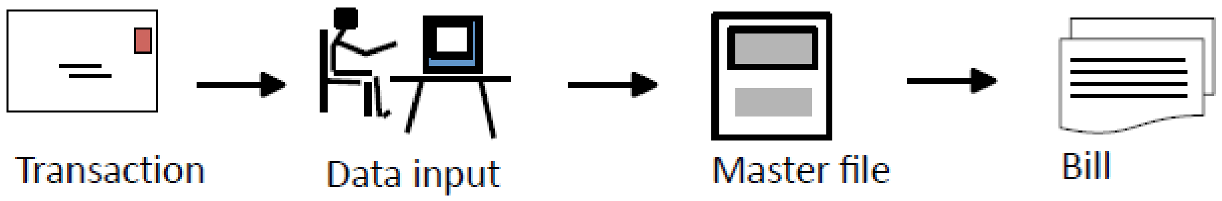
- Replication: Several copies of the data are held in different locations.
    - Mirror: Complete data set is replicated
    - Cache: Dynamic set of data is replicated (e.g., most recently used)
- With replicated data, the biggest problems are concurrency and consistency

Batch Processing with Master File Update

- Electricity utility customer billing (e.g., NYSEG)
- Telephone call recording and billing (e.g., Verizon)
- Car rental reservations (e.g., Hertz)
- Bank (e.g., Tompkins Trust)
- University grade registration (e.g., IU)
- Example: Electricity Utility Billing
- Requirements analysis identifies several transaction types:
    - Create account/ close account
    - Meter reading
    - Payment received
    - Other credits/ debits
    - Check cleared/ check bounced
    - Account query
    - Correction of error
    - etc., etc., etc.,

First Attempt

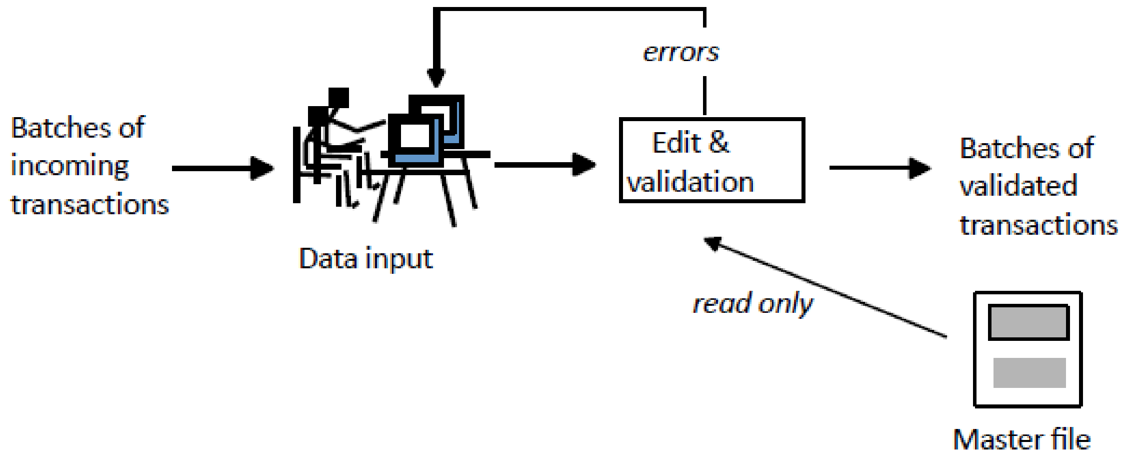
- Each transaction is handled as it arrives.



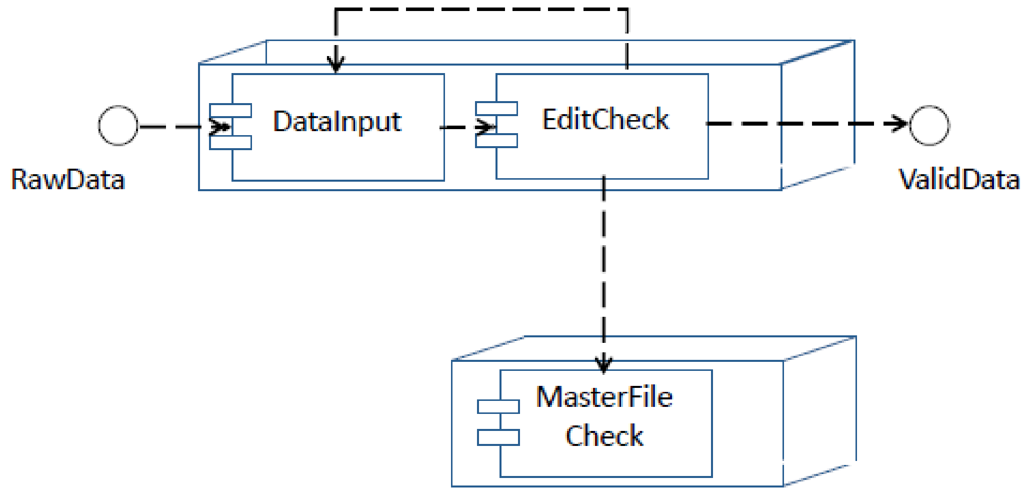
Criticisms of First Attempt

- Where is this first attempt weak?
    - All activities are triggered by a transaction.
    - A bill is sent out for each transaction, even if there are several per day.
    - Bills are not sent out on a monthly cycle.
    - Awkward to answer customer queries.
    - No process for error checking and correction.
    - Inefficient in staff time.

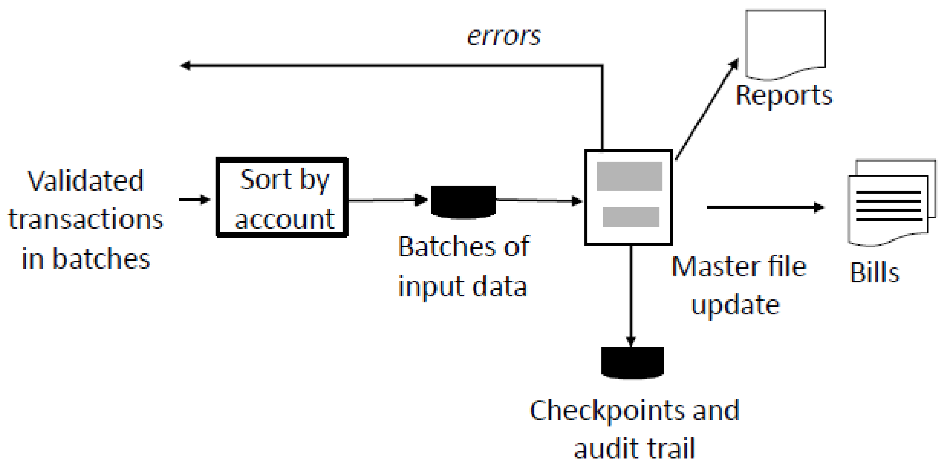
Batch Processing: Edit and Validation


errors
Batches of incoming transactions
Data input
Edit & validation
Batches of validated transactions
read only
Master file

Deployment Diagram: Validation



Batch Processing: Master File Update-MFU

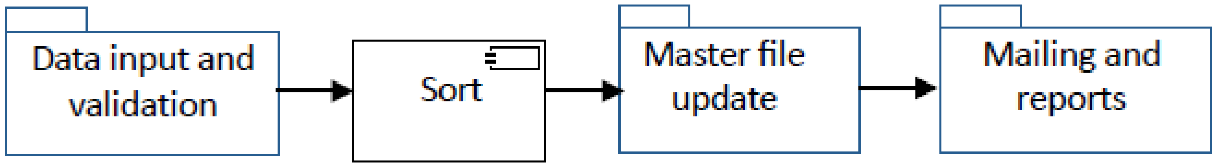


Benefits of Batch Processing with MFU

- All transactions for an account are processed together at appropriate intervals, e.g., monthly.
- Backup and recovery have fixed checkpoints.
- Better management control of operations.
- Efficient use of staff and hardware.
- Error detection and correction is simplified.

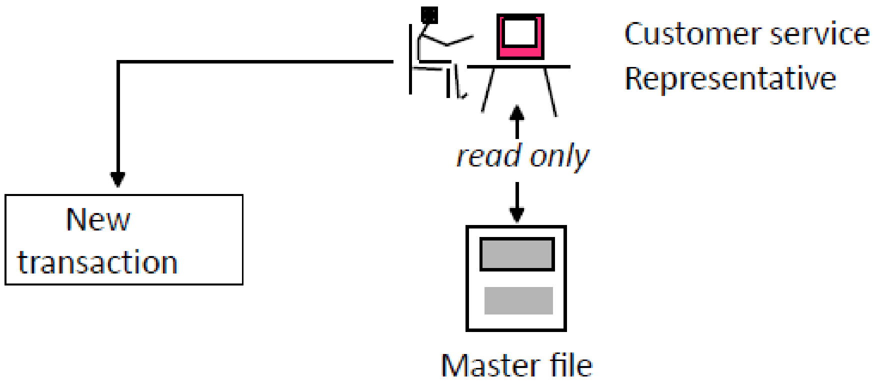
Architectural Style: MFU (Basic Version)

- Advantages:
    - Efficient way to process batches of transactions.
- Disadvantages:
    - Information in master file is not updated immediately.
    - No good way to answer customer inquiries.



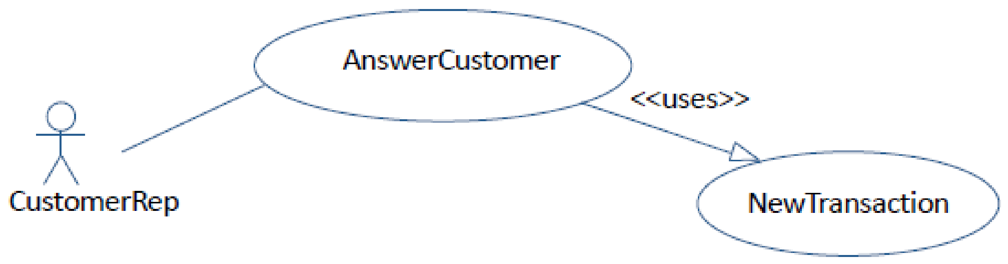
Online Inquiry

- A customer calls the utility and speaks to a customer service representative.
- Customer service department can read the master file, make annotations, and create transactions, but cannot change the master file.



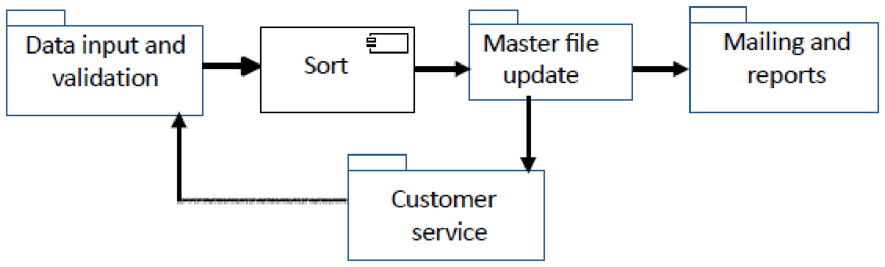
Online Inquiry: Use Case

- The representative can read the master file, but not make changes to it.
- If the representative wishes to change information in the master file, a new transaction is created as input to the master file update system.



Architectural Style: Master File Update (Full)

- Advantage:
  - Efficient way to answer customer inquiries.
- Disadvantage:
  - Information in master file is not updated immediately.

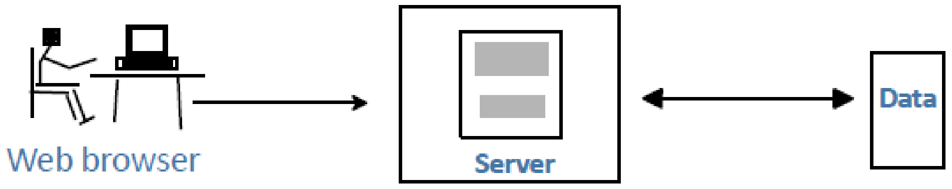


Example 2: Three Tier Architecture-TTA

- The basic client/server architecture of the web has:
  - a server that delivers static pages in HTML format
  - a client (known as a browser) that renders HTML pages
  - Both client and server implement the HTTP interface.
- Extend the architecture of the server so that it can configure HTML pages dynamically.

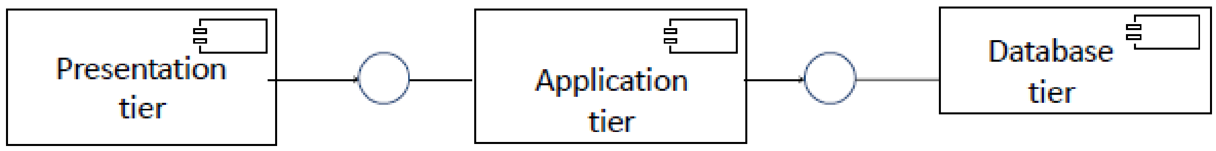
Web Server with Data Store

- Advantage:
  - Server-side code can configure pages, access data, validate information, etc.
- Disadvantage:
  - All interaction requires communication with server

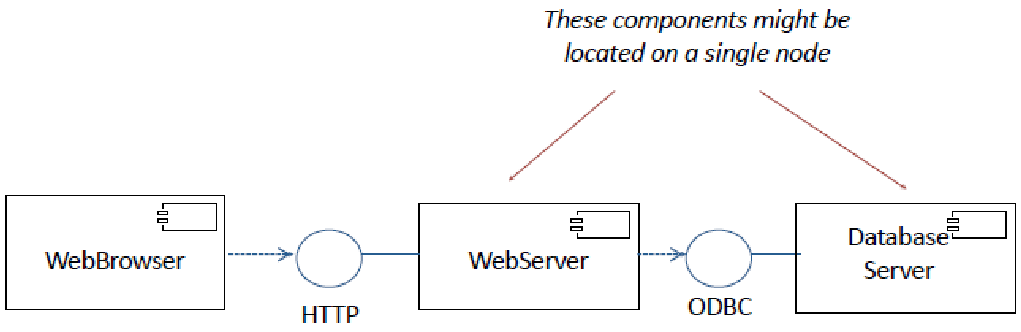


Architectural Style: TTA

- Each of the tiers can be replaced by other components that implement the same interfaces.

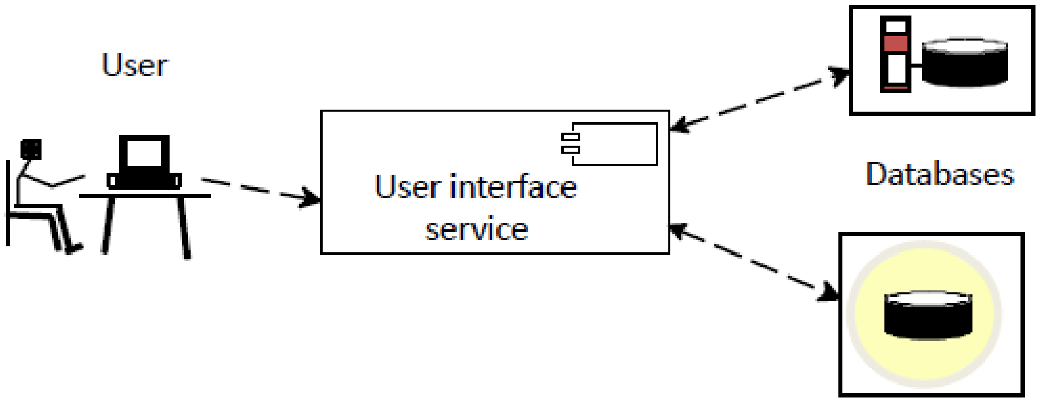


Component Diagram



TTA: Broadcast Searching

- This is an example of a multicast protocol.
- The primary difficulty is to avoid troubles at one site degrading the entire system (e.g., every transaction cannot wait for a system to time out).



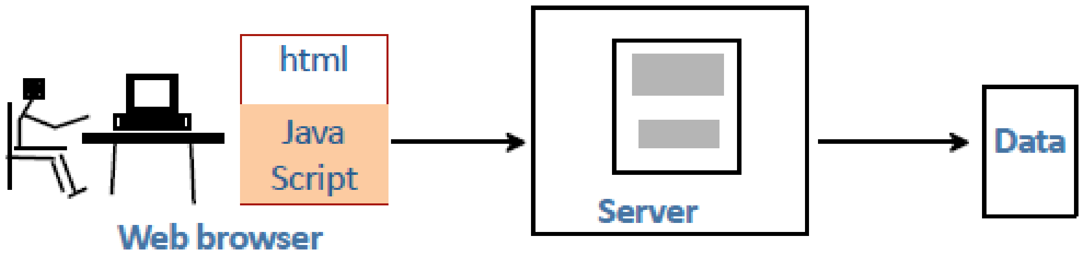
Extending the Architecture of the Web

- Using a three tier architecture, the web has:
- a server that delivers dynamic pages in HTML format
- a client (known as a browser) that renders HTML pages
- Both server and client implement the HTTP interface.
- Every interaction with the user requires communication between the client and the server.

- Problem 2 Extend the architecture so that simple user interactions do not need messages to be passed between the client and the server.

Extending the Web with Executable Code that can be Downloaded

- Executable code in a scripting language such as JavaScript can be downloaded from the server
- Advantage:
    - Scripts can interact with user and process information locally.
- Disadvantage:
    - All interactions are constrained by web protocol



Extending the Three Tier Architecture

- In the three tier architecture, a web site has:
    - a client that renders HTML pages and executes scripts
    - a server that delivers dynamic pages in HTML format
    - a data store
- The three tier architecture with downloadable scripts is one the ways in which the basic architecture has been extended. There are some more:
    - Protocols: e.g., HTTPS, FTP, proxies
    - Data types: e.g., helper applications, plug-ins
    - Executable code: e.g., applets, servlets
    - Style sheets: e.g., CSS

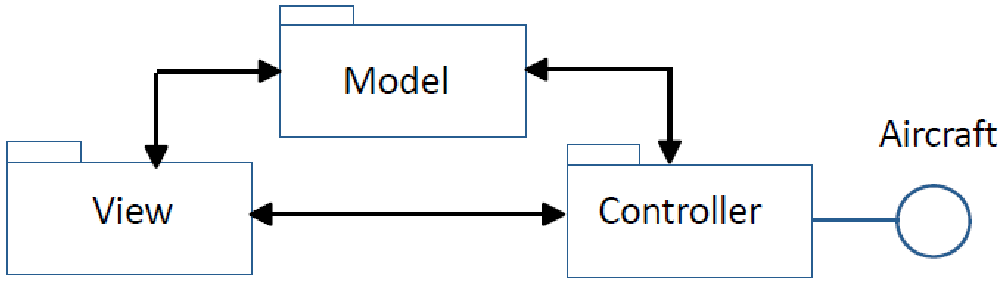
Example 3: Model/View/Controller (MVC)

- The definition of Model/View/Controller (MVC) is in a state of flux. The term is used to describe a range of architectures and designs.
    - Some are system architectures, where the model, view, and controller are separate components.
    - Some are program designs, with classes called model, view, and controller.
- We will look at three variants:
    - An MVC system architecture used in robotics.

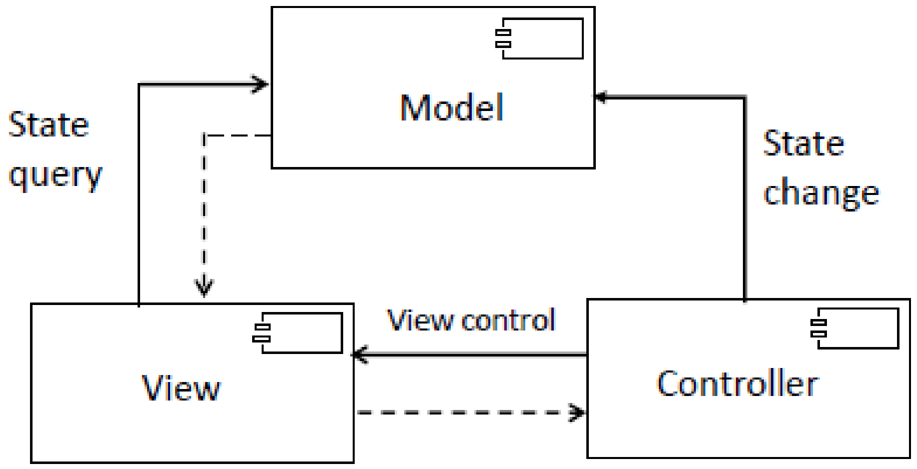
    - A general purpose MVC system architecture used for interactive systems.
    - Apple’s version of MVC as a program design for mobile apps.

Model/View/Controller in Robotics

- Controller: Receives instrument readings from the aircraft, updates the view, and sends controls signals to the aircraft.
- Model: Translates data received from and sent to the aircraft, and instructions from the user into a model of flight performance. Uses domain knowledge about the aircraft and flight.
- View: Displays information about the aircraft to the user on the ground and transmits instructions to the model via the controller.



Example 3: MVC for Mobile Apps



Model

- The model records the state of the application and notifies subscribers. It does not depend on the controller or the view.
    - stores the state of the application in suitable data structures or databases
    - responds to instructions to change the state information
    - notifies subscribers of events that change the state
    - may be responsible for validation of information

View

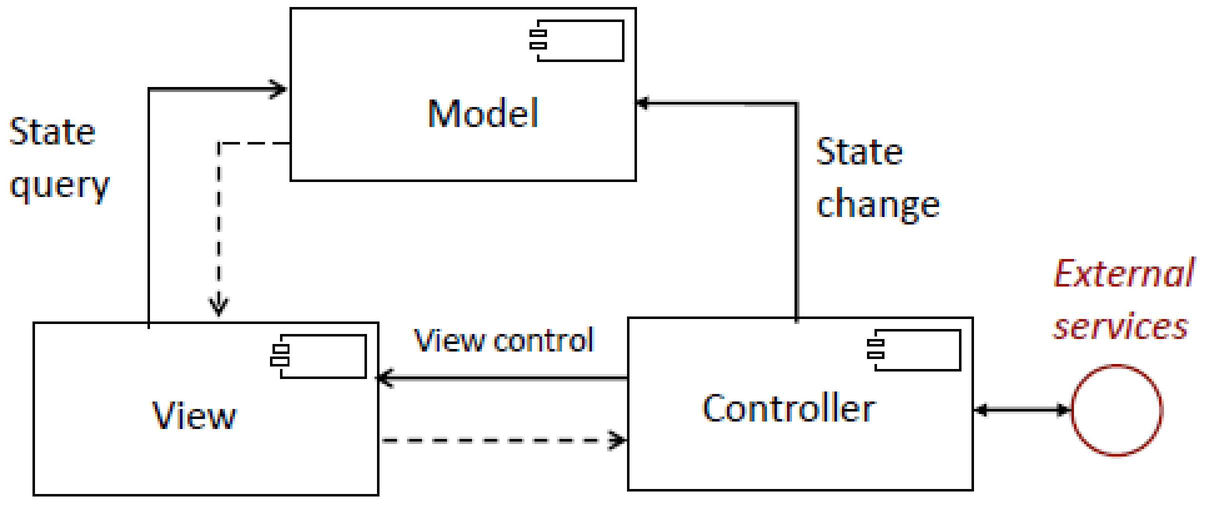
- The view is the part of the user interface that presents the state of the interface to the user. It subscribes to the model, which notifies it of events that change the state.
    - renders data from the model for the user interface
    - provides editors for properties, such as text fields, etc.
    - receives updates from the model
    - sends user input to the controller
- A given model may support a choice of alternative views.

Controller

- The controller is the part of the user interface that manages user input and navigation within the application.
    - defines the application behavior
    - maps user actions to changes in the state of the model
    - interacts with external services via APIs
    - may be responsible for validation of information
- Different frameworks handle controllers in different ways. In particular there are several ways to divide responsibilities between the model and the controller, e.g., data validation, external APIs.

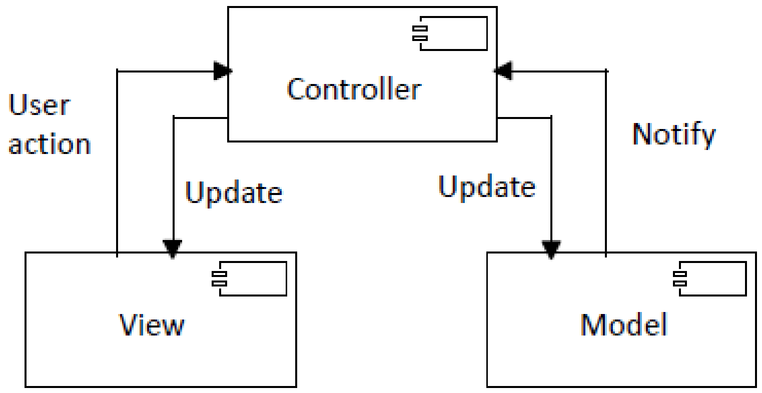
External Services for Mobile Apps

- Mobile apps often make extensive use of cloud-based external services, each with an API (e.g., location, validation). These are usually managed by the controller.



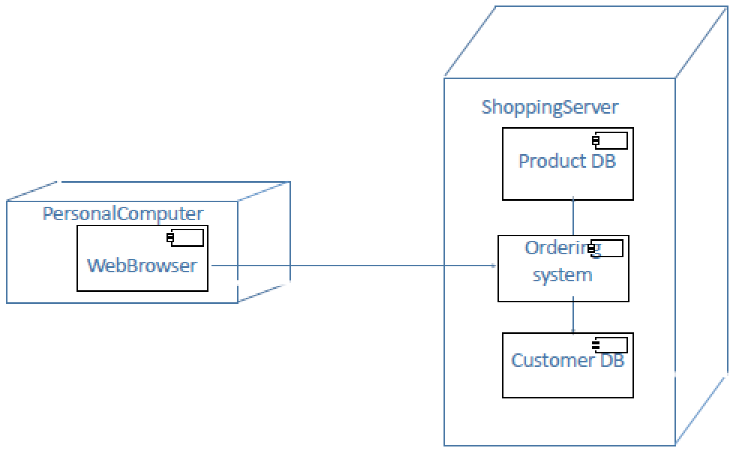
Apple's Version of MVC

- The diagram shows the model, view, and controller as components. In practice the Model-View-Controller is a program design with three major classes.

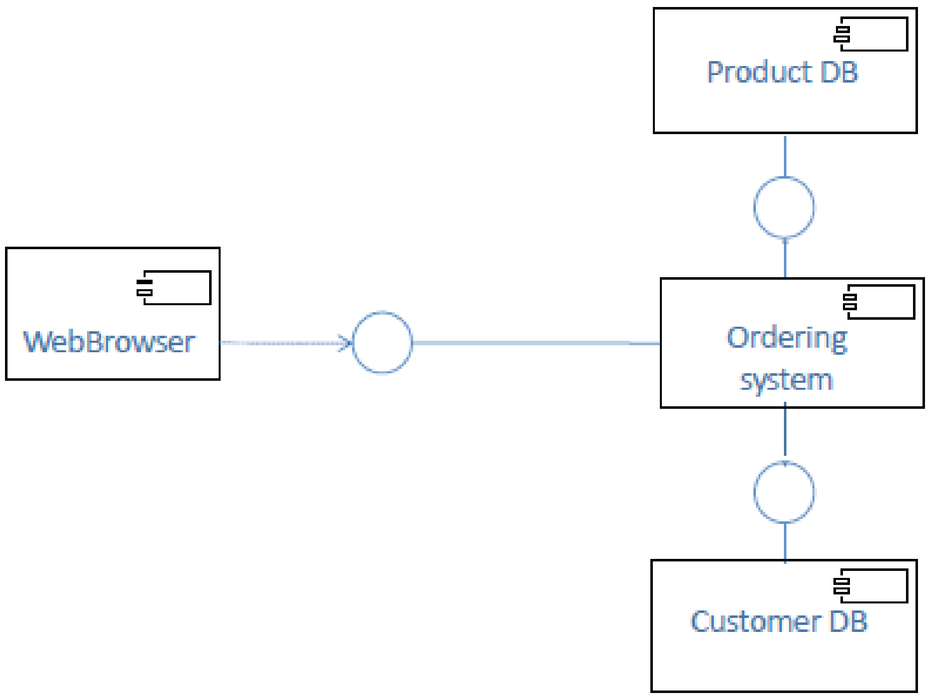


An Exam Question

- A company that makes sports equipment decides to create a system for selling sports equipment online. The company already has a product database with description, marketing information, and prices of the equipment that it manufactures.
- To sell equipment online the company will need to create: a customer database, and an ordering system for online customers.
- The plan is to develop the system in two phases. During Phase 1, simple versions of the customer database and ordering system will be brought into production. In Phase 2, major enhancements will be made to these components.
- a) For the system architecture of Phase 1:
    - i). Draw a UML deployment diagram.



- a) For the system architecture of Phase 1:
  - i). Draw a UML deployment diagram.



- (b) For Phase 1:
  - i). What architectural style would you use for the customer database? Repository with Storage Access Layer
  - ii). Why would you choose this style? It allows the database to be replaced without changing the applications that use the database.

Class 4b: Implementation

Program Design

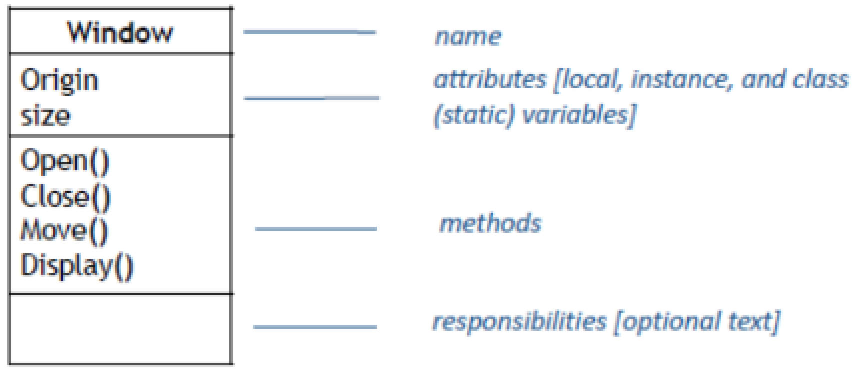
- The task of program design is to represent the software architecture in a form that can be implemented as one or more executable programs. Given a system architecture, the program design specifies:
  - programs, components, packages, classes, class hierarchies, etc.
  - interfaces, protocols (where not part of the system architecture)
  - algorithms, data structures, security mechanisms, operational procedures
- If the program design is done properly, all significant design decisions should be made before implementation. Implementation should focus on the actual coding.

UML Models

- UML models (diagrams and specifications) can be used for almost all aspects of program design.
    - Diagrams give a general overview of the design, showing the principal elements and how they relate to each other.
    - Specifications provides details about each element of the design. The specification should have sufficient detail that they can be used to write code from.
- In heavyweight software development processes, the entire specification is completed before coding begins.
- In lightweight software development processes, an outline specification is made before coding, but the details are completed as part of the coding process, using language based tools such as Javadocs.
- Models used mainly for requirements
    - Use case diagram shows a set of use cases and actors, and their relationships.
- Models used mainly for systems architecture
    - Component diagram shows the organization and dependencies among a set of components.
    - Deployment diagram shows the configuration of processing nodes and the components that live on them.
- Models used mainly for program design
    - Class diagram shows a set of classes, interfaces, and collaborations with their relationships.
    - Object diagram or sequence diagram show a set of objects and their relationships.

Class Diagram

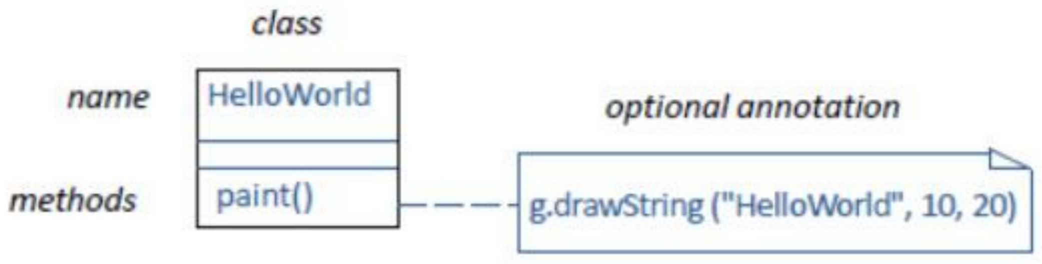
- A class is a description of a set of objects that share the same attributes, methods, relationships, and semantics.


Window
Origin
size
Open()
Close()
Move()
Display()
name
attributes [local, instance, and class (static) variables]
methods
responsibilities [optional text]

The "Hello, World!" Applet

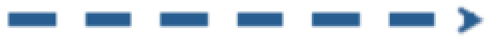
```
import java.awt.Graphics;
class HelloWorld extends java.applet.Applet {
        public void paint (Graphics g) {
        g.drawString ("Hello, World!", 10, 20);
        }
}
```

The HelloWorld Class

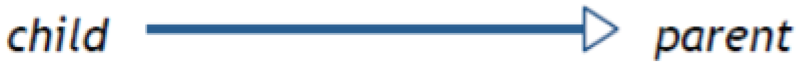


Notation: Relationships

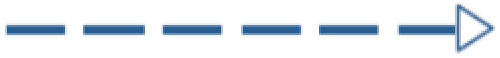
- A dependency is a semantic relationship between two things in which a change to one may effect the semantics of the other.



- A generalization is a relationship is which objects of the specialized element (child) are substitutable for objects of the generalized element (parent).

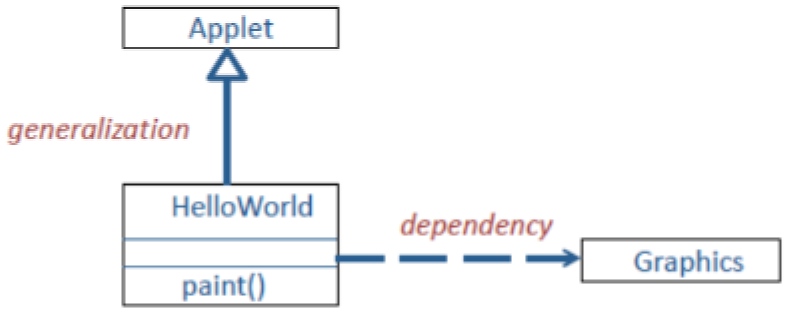


- A realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.



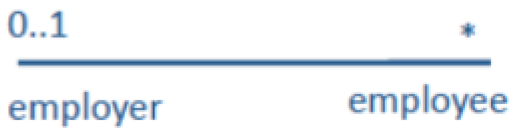
The HelloWorld Class

- Note that the Applet and Graphics classes are shown briefly, i.e., just the name is shown, not the attributes or operations.

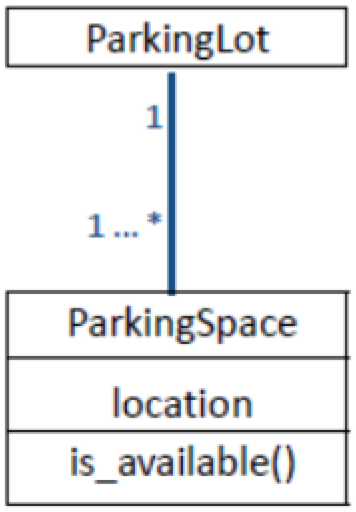


Notation: Association

- An association is a structural relationship that describes a set of links, a link being a connection among objects.



Association



Deciding which Classes to Use

- Given a real-life system, how do you decide what classes to use?

- Step 1. Identify a set of candidate classes that represent the system design.
    - What terms do the users and implementers use to describe the system? These terms are candidates for classes.
    - Is each candidate class crisply defined?
    - For each class, what is its set of responsibilities? Are the responsibilities evenly balanced among the classes?
    - What attributes and methods does each class need to carry out its responsibilities?
- Step 2. Modify the set of classes
- Goals:
    - Improve the clarity of the design
        - If the purpose of each class is clear, with easily understood methods and relationships, developers are likely to write simple code, which future maintainers can understand and modify.
    - Increase coherence within classes, and lower coupling between classes.
    - Aim for high cohesion within classes and weak coupling between them.

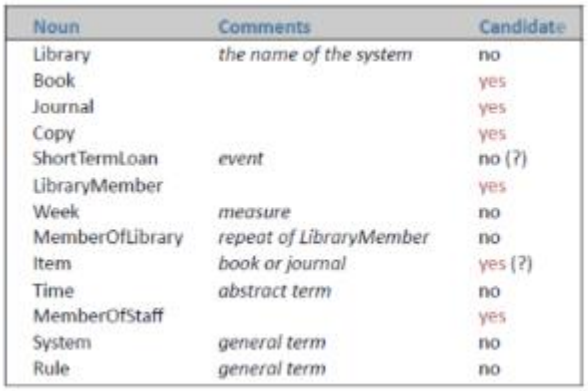
Application Classes and Solution Classes

- A good design is often a combination of application classes and solution classes.
- Application classes represent application concepts. Noun identification is an effective technique to generate candidate application classes.
- Solution classes represent system concepts, e.g., user interface objects, databases, etc.

Noun Identification: a Library Example

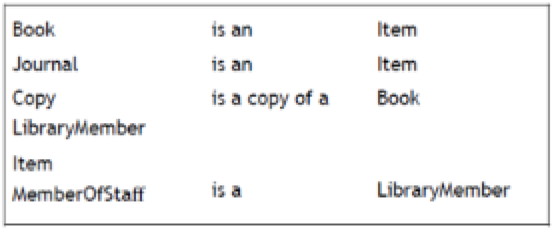
- The library contains books and journals. It may have several copies of a given book. Some of the books are reserved for short-term loans only.
- All others may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.
- The system must keep track of when books and journals are borrowed and returned, and enforce the rules.

Candidate Classes



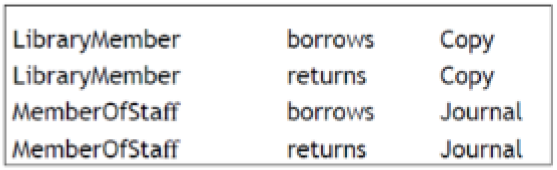
Noun	Comments	Candidate
Library	*the name of the system*	no
Book		yes
Journal		yes
Copy		yes
ShortTermLoan	*event*	no (?)
LibraryMember		yes
Week	*measure*	no
MemberOfLibrary	*repeat of LibraryMember*	no
Item	*book or journal*	yes (?)
Time	*abstract term*	no
MemberOfStaff		yes
System	*general term*	no
Rule	*general term*	no

Relations between Classes



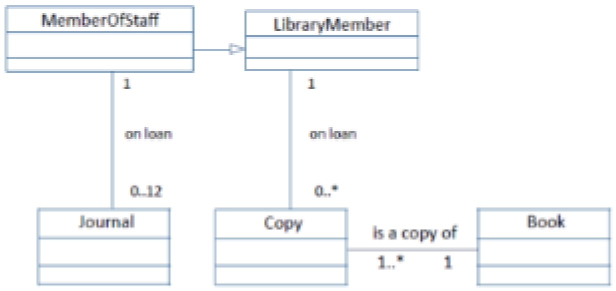
Book	is an	Item
Journal	is an	Item
Copy	is a copy of a	Book
LibraryMember		
Item		
MemberOfStaff	is a	LibraryMember

Methods



LibraryMember	borrows	Copy
LibraryMember	returns	Copy
MemberOfStaff	borrows	Journal
MemberOfStaff	returns	Journal

A Possible Class Diagram

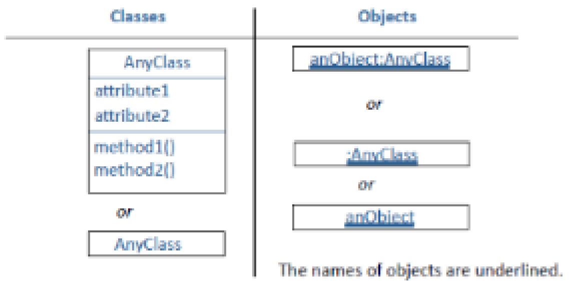


From Candidate Classes to Completed Design

- Methods used to move to final design
- Reuse: Wherever possible use existing components, or class libraries. They may need extensions.
- Restructuring: Change the design to improve understandability, maintainability, etc. Techniques include merging similar classes, splitting complex classes, etc.

- Optimization: Ensure that the system meets anticipated performance requirements, e.g., by changed algorithms or restructuring.
- Completion: Fill all gaps, specify interfaces, etc.
- Design is iterative
    - As the process moves from preliminary design to specification, implementation, and testing it is common to find weaknesses in the program design. Be prepared to make major modifications.

UML Notation for Classes and Objects

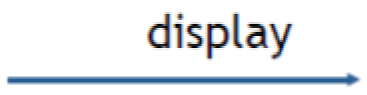


Modeling Dynamic Aspects of Systems

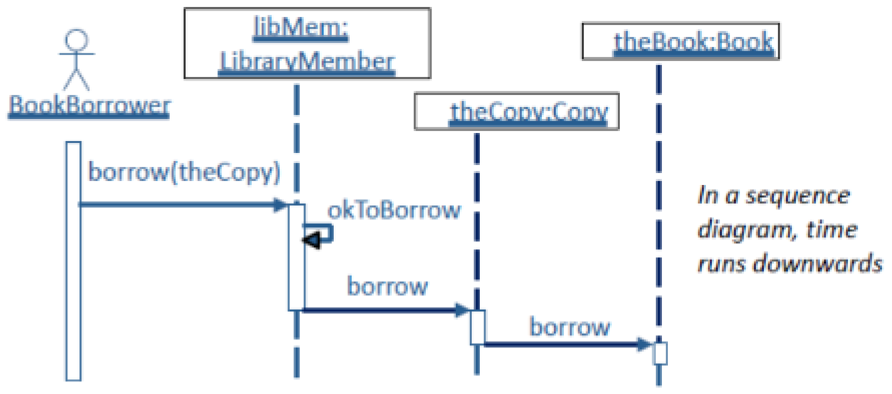
- Interaction diagram: shows set of objects and their relationships including messages that may be dispatched among them.
    - Sequence diagrams: Time ordering of messages

Notation: Interaction

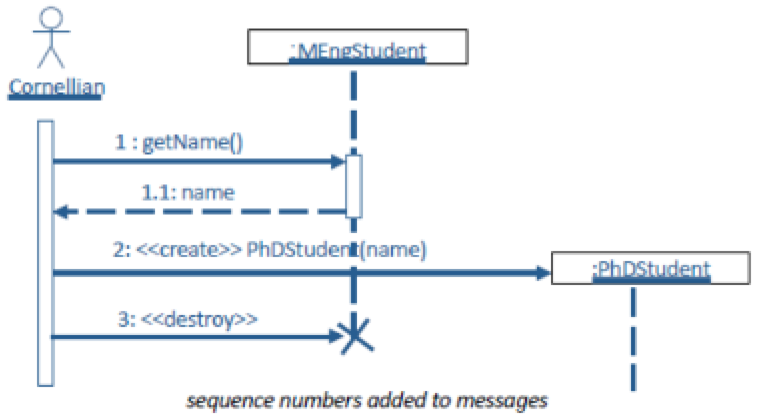
- An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.



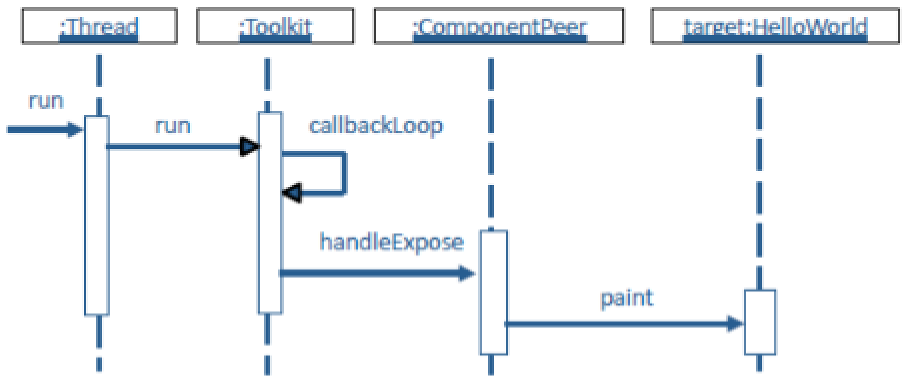
Sequence Diagram: Borrow Copy of a Book



Sequence Diagram: Change in Cornell Program

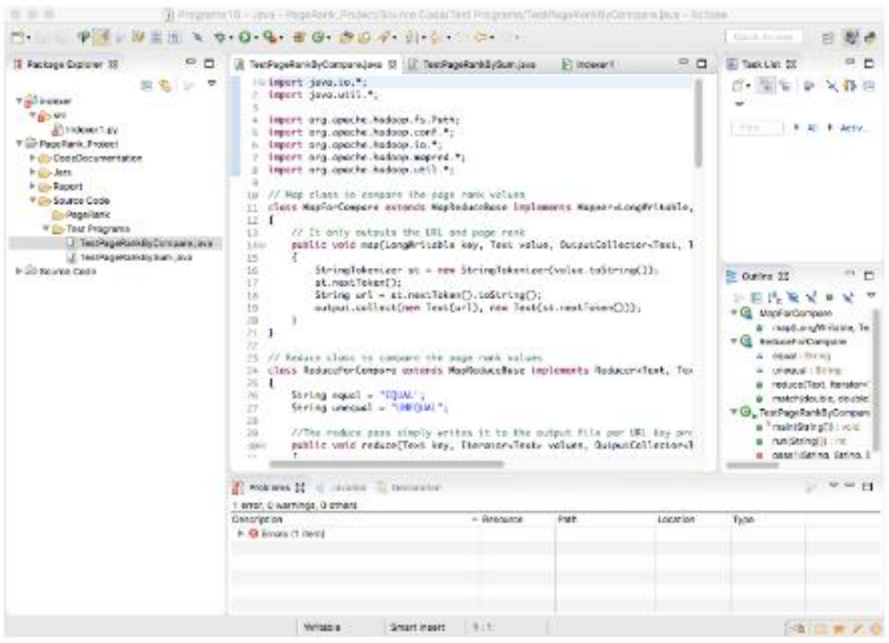


Sequence Diagram: Painting Mechanism



Integrated Development Environments

- Basic software development requires:
  - text editor (e.g., vi editor for Linux)
  - compiler for individual files
  - build system (e.g., make for Linux)
- Integrated development environments combine:
  - source code editor
  - incremental compiler
  - build automation tools
  - a debugger



```
import java.io.*;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

// Map class to compare the page rank values
class MapForCompare extends MapReduceBase implements Mapper<LongWritable,
{
    // It only outputs the URL and page rank
    public void map(LongWritable key, Text value, OutputCollector<Text, T
    {
        StringTokenizer st = new StringTokenizer(value.toString());
        st.nextToken();
        String url = st.nextToken().toString();
        output.collect(new Text(url), new Text(st.nextToken()));
    }
}

// Reduce class to compare the page rank values
class ReduceForCompare extends MapReduceBase implements Reducer<Text, Te
{
    String equal = "EQUAL";
    String unequal = "UNEQUAL";

    //The reduce pass simply writes it to the output file per URL key pro
    public void reduce(Text key, Iterator<Text> values, OutputCollector<T
```

```
1 error, 0 warnings, 0 others
Description          Resource    Path    Location    Type
▶ Errors (1 item)
```

- Eclipse is a modern integrated development environment. It was originally created by IBM's Rational division. There are versions for many languages including Java, C/C++, Python, etc. The Java system provides:
  - source code editor
  - debugger
  - incremental compiler
  - programming documentation
  - build automation tools
  - version control
  - XML editor and tools
  - web development tools Much more is available via plug-ins

Program Design: Integrated Development Environment

- Integrated development environments provide little help in designing a program.
- They assume that you have already have a design:
  - classes
  - methods
  - data structures
  - interfaces
- Options for program design:
  - program design using modeling tools, such as UML
  - design while coding: design — code — redesign loop (small programs only)
  - existing frameworks
  - advanced environments that combine frameworks and development tools
- It is often good to combine aspects of these different approaches

The Design — Code — Redesign Loop

- If the class structure is straightforward it may be possible to use the integrated development environment to:
  - create an outline of the class structure and interfaces
  - write code
  - modify the class structure as needed and rework the code as necessary
- This is only possible with small teams with close communication.
- The maximum size of program depends on experience of programmer(s) and complexity of the program. It may be possible to complete a single agile sprint.
- Eventually the amount of rework becomes overwhelming.

Class Hierarchies

- Since the design of class hierarchies is difficult it is good practice to use existing frameworks.
- Often many of the classes will have been written for you, or abstract classes are provided that you can use as a basis for your own subclasses.
- Examples:
  - class hierarchies that are part of programming languages
  - toolkits (e.g., for graphical user interfaces)

    - design patterns
    - frameworks for web development and mobile apps
- Example: Java
- Java is a relatively straightforward language with a very rich set of class hierarchies.
    - Java programs derive much of their functionality from standard classes.
    - Learning and understanding the classes is difficult.
    - Experienced Java programmers can write complex systems quickly.
    - Inexperienced Java programmers write inelegant and buggy programs.
- Languages such as Java and Python steadily change their class hierarchies over time. Commonly the changes replace special purpose functionality with more general frameworks.
- If you design your programs to use the class hierarchies in the style intended by the language developers, it is likely to help with long term maintenance.

Web Development Frameworks

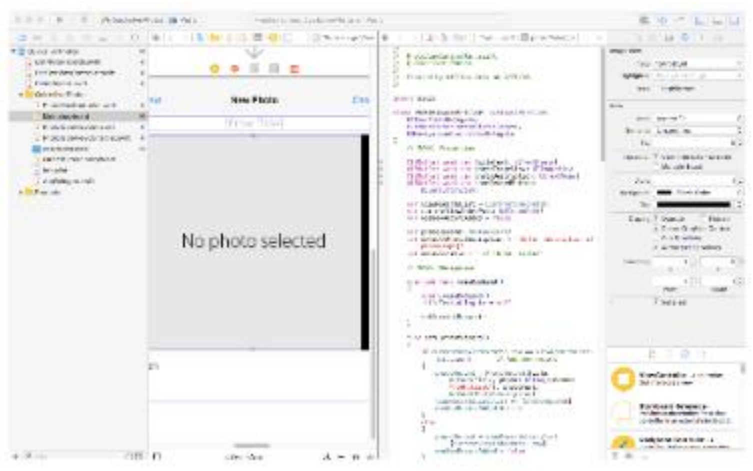
- A web development framework provides a skeleton for building web applications.
- An early example was Cold Fusion, which implements a three tier architecture. Modern example, such as Ruby on Rails and Django, often use a MVC architecture. For example, Ruby on Rails provides:
    - a database
    - a web server
    - web pages
- It is intended to be used with web standards, e.g., XML , HTML, CSS, and JavaScript.

Web Development Frameworks: Django

- Django is a Python framework for developing web sites
    - loosely based on MVC architecture
    - supports a variety of web and database servers
    - web template system
    - authentication system
    - administrative interface
    - mitigation of web attacks
- Django is a complex framework. Teams should allow plenty of time for learning

Advanced Development Environments

- Application frameworks can be used with any program development environment, e.g., Django and Eclipse (Python version)
- An advanced development environment combines:
    - integrated development environment (IDE)
    - application framework
    - user interface layout manager and more
- Example: Apple's Xcode for iOS



- An advanced development environment is intended to provide everything that a developer needs.
- The developer is expected to follow the program choices that are provided.
- For example, when Xcode is used with iOS it has a very specific purpose: mobile apps for Apple devices such as iPhones, iPads.
    - Special programming language (Swift or Objective C)
    - MVC framework (Apple version)
- If you accept the overall program design it is very powerful:
    - Auto layout of graphical interfaces
    - Comprehensive set of classes for user interfaces and navigation
    - Simulators for all Apple devices

Using Development Frameworks

- Development frameworks are powerful and flexible.
- If your application fits the framework, they do much of the program design and provide high quality code for many of the standard parts of any application.
- Some parts of the application may need be designed separately.

- But beware:
    - You are forced to build your application within the framework that is provided.
    - The frameworks are continually modified.
    - These frameworks are complex and take a long time to learn.

Production Programming

- Murphy's Law:
    - If anything can go wrong, it will.
- Challenges:
    - Code has to be maintained over the long term, with different system software.
    - Interfaces will be used in new and unexpected ways.
    - Every possible error will eventually occur at the worst possible time (bad data, failures of hardware and system software).
    - There are likely to be security attacks.
- Robust programming
    - Write simple code.
    - Avoid risky programming constructs.
    - If code is difficult to read, rewrite it.
    - Incorporate redundant code to check system state after modifications.
    - Test implicit assumptions explicitly, e.g., check all parameters received from other routines.
    - Eliminate all warnings from source code.
    - Have a thorough set of test cases for all your code.
- In a production environment, expect to spend longer on coding and testing than in an academic setting.

Software Reuse

- It is often good to design a program to reuse existing components. This can lead to better software at lower cost.
- Potential benefits of reuse
    - Reduced development time and cost
    - Improved reliability of mature components
    - Shared maintenance cost
- Potential disadvantages of reuse
    - Difficulty in finding appropriate components
    - Components may be a poor fit for application

    - Quality control and security may be unknown

Evaluating Software

- Software from well established developers is likely to be well written and tested, but still will have bugs and security weaknesses, especially when incorporated in unusual applications.
- The software is likely to be much better than a new development team would write.
- But sometimes it is sensible to write code for a narrowly defined purpose rather than use general purpose software.
- Maintenance: When evaluating software, both commercial and open source, pay attention to maintenance. Is the software supported by an organization that will continue maintenance over the long term?

Reuse: Open Source Software

- Open source software varies enormously in quality.
- Because of the processes for reporting and fixing problems, major systems such as Linux, Apache, Python, Lucene, etc. tend to be very robust and free from problems. They are often better than the commercial equivalents.
- More experimental systems, such as Hadoop, have solid cores, but their lesser used features have not been subject to the rigorous quality control of the best software products.
- Other open source software is of poor quality and should not be incorporated in production systems.

Evaluating Applications Packages

- Applications packages for business functions are provided by companies such as SAP and Oracle. They provide enormous capabilities and relieve an organization from such tasks as updating financial systems when laws change.
- They are very expensive:
    - License fees to the vendor.
    - Modifications to existing systems and special code from the vendor.
    - Disruption to the organization when installing them.
    - Long term maintenance costs.
    - The costs of changing to a different vendor are huge.
- Cornell's decision (about 1990) to move to PeopleSoft (now part of Oracle) has cost the university several hundred millions of dollars.

- If you are involved in such a decision insist on a very thorough feasibility study. Be prepared to take a least a year and spend several million dollars before making the decision.

Design for Change: Replacement of Components

- The software design should anticipate possible changes in the system over its life-cycle.
- New vendor or new technology
    - Components are replaced because its supplier goes out of business, ceases to provide adequate support, increases its price, etc., or because software from another source provides better functionality, support, pricing, etc.
- This can apply to either open source or vendor-supplied components.
- New implementation
    - The original implementation may be problematic, e.g., poor performance, inadequate back-up and recovery, difficult to trouble - shoot, or unable to support growth and new features added to the system.
- Example.
    - The portal nsdl.org was originally implemented using uPortal. This did not support important extensions that were requested and proved awkward to maintain. It was reimplemented using PHP/MySQL
- Additions to the requirements
    - When a system goes into production, it is usual to reveal both weaknesses and opportunities for extra functionality and enhancement to the user interface design. For example, in a data-intensive system it is almost certain that there will be requests for extra reports and ways of analyzing the data.
- Requests for enhancements are often the sign of a successful system. Clients recognize latent possibilities.
- Changes in the application domain
    - Most application domains change continually, e.g., because of business opportunities, external changes (such as new laws), mergers and take-overs, new groups of users, new technology, etc., etc.,
- It is rarely feasible to implement a completely new system when the application domain changes. Therefore existing systems must be modified.

This may involve extensive restructuring, but it is important to reuse existing code as much as possible.

Design Patterns

- Design patterns are template designs that can be used in a variety of systems. They are particularly appropriate in situations where classes are likely to be reused in a system that evolves over time.

Structural

- These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.
- Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- Bridge decouples an abstraction from its implementation so that the two can vary independently.
- Composite composes zero-or-more similar objects so that they can be manipulated as one object.
- Decorator dynamically adds/overrides behavior in an existing method of an object.
- Facade provides a simplified interface to a large body of code.
- Flyweight reduces the cost of creating and manipulating a large number of similar objects.
- Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Legacy Systems

- Many data intensive systems, e.g., those used by banks, universities, etc. are legacy systems. They may have been developed forty years ago as batch processing, master file update systems and been continually modified.
    - Recent modifications might include customer interfaces for the web, smartphones, etc.
    - The systems will have migrated from computer to computer, across operating systems, to different database systems, etc.
    - The organizations may have changed through mergers, etc.
- Maintaining a coherent system architecture for such legacy systems is an enormous challenge, yet the complexity of building new systems is so great that it is rarely attempted.

- The Worst Case A large, complex system that was developed several decades ago:
    - Widely used either within a big organization or by an unknown number of customers.
    - All the developers have retired or left.
    - No list of requirements. It is uncertain what functionality the system provides and who uses which functions.
    - System and program documentation incomplete and not kept up to date.
    - Written in out-of-date versions of programming languages using system software that is also out of date.
    - Numerous patches over the years that have ignored the original system architecture and program design.
    - Extensive code duplication and redundancy.
    - The source code libraries and production binaries may be incompatible.

Legacy Requirements

- Planning
    - In conjunction with the client develop a plan for rebuilding the system.
- Requirements as seen by the customers and users
    - Who are the users?
    - What do they actually use the system for?
    - Does the system have undocumented features that are important or bugs that users rely on?
    - How many people use the fringe parts of the system? Where are they flexible?
- Requirements as implied by the system design
    - If there is any system documentation, what does it say about the requirements?
    - Does the source code include any hints about the requirements?
    - Is there code to support obsolete hardware or services? If so, does anybody still use them?

Legacy Code

- Source code management
    - Use a source code management system to establish a starting version of the source code and binaries that are built from this source code.
    - Create a test environment so that the rebuilt system can be compared with the current system. Begin to collect test cases.
    - Check the licenses for all vendor software.
- Rebuilding the software
    - An incremental software development process is often appropriate, with each increment released when completed.
    - The following tasks may be tackled in any appropriate order, based on the condition of the code. Usually the strategy will be to work on different parts of the system in a series of phases.
        - Understand the original systems architecture and program design.
        - Establish a component architecture, with defined interfaces, even if much of the code violates the architecture and needs adapters.
- Move to current versions of programming languages and systems software.
- If there are any subsystems that do not have source code, carry out a development cycle to create new code that implements the requirements.
- If there is duplicate code, replace with a single version.
- Remove redundant code and obsolete requirements.
- Clean up as you go along.

Testing: System and Sub-System Testing

- Tests on components or complete system, combining units that have already been thoroughly tested
- Emphasis on integration and interfaces
- Trial data that is typical of the actual data, and/or stresses the boundaries of the system, e.g., failures, restart
- Carried out systematically, adding components until the entire system is assembled
- Can be open or closed box: by development team or by special testers
- System testing is finished fastest if each component is completely debugged before assembling the next

Class 5: Testing

Bugs, Faults, and Failures

- Bug (fault):
  - Programming or design error whereby the delivered system does not conform to specification (e.g., coding error, protocol error)
- Failure:
  - Software does not deliver the service expected by the user (e.g., mistake in requirements, confusing user interface)

Failure of Requirements

- An actual example
  - The head of an organization is not paid his salary because it is greater than the maximum allowed by the program. (Requirements problem)

Bugs and Features

- That's not a bug. That's a feature!
  - Users will often report that a program behaves in a manner that they consider wrong, even though it is behaving as intended. That's not a bug.
- That's a failure!
  - The decision whether this needs to be changed should be made by the client not by the developers.

Terminology

- Fault avoidance
  - Build systems with the objective of creating fault-free (bug - free) software.
- Fault detection (testing and verification)
  - Detect faults (bugs) before the system is put into operation or when discovered after release.
- Fault tolerance
  - Build systems that continue to operate when problems (bugs, overloads, bad data, etc.) occur.

Failures: A Case Study

- A passenger ship with 1,509 persons on board grounded on a shoal near Nantucket Island, Massachusetts. At the time the vessel was about 17 miles from where the officer thought they were. The vessel was enroute from Bermuda to Boston.

Case Study: Analysis

- From the report of the National Transportation Safety Board
    - The ship was steered by an autopilot that relied on position information from the Global Positioning System (GPS).
    - If the GPS could not obtain a position from satellites, it provided an estimated position based on Dead Reckoning (distance and direction traveled from a known point).
    - The GPS failed one hour after leaving Bermuda.
    - The crew failed to see the warning message on the display (or to check the instruments).
    - 34 hours and 600 miles later, the Dead Reckoning error was 17 miles.

Case Study: Software Lessons

- All the software worked as specified(no bugs), but ...
    - After the GPS software was specified, the requirements changed (standalone system now part of integrated system).
    - The manufacturers of the autopilot and GPS adopted different design philosophies about the communication of mode changes.
    - The autopilot was not programmed to recognize valid/invalid status bits in messages from the GPS.
    - The warnings provided by the user interface were not sufficiently conspicuous to alert the crew.
    - The officers had not been properly trained on this equipment. Reliable software needs all parts of the software development process to be carried out well.

Building Reliable Software: Quality Management Processes

- Assumption:
    - Good software is impossible without good processes
- The importance of routine:
    - Standard terminology (requirements, design, acceptance, etc.)
    - Software standards (coding standards, naming conventions, etc.)
    - Regular builds of complete system (often daily)
    - Internal and external documentation
    - Reporting procedures
- This routine is important for both heavyweight and lightweight development processes.
- When time is short...

    - Pay extra attention to the early stages of the process: feasibility, requirements, design.
    - If mistakes are made in the requirements process, there will be little time to fix them later.
    - Experience shows that taking extra time on the early stages will usually reduce the total time to release.

Building Reliable Software: Communication with the Client

- A system is no use if it does not meet the client's needs
    - The client must understand and review the agreed requirements in detail. It is not sufficient to present the client with a specification document and ask him/her to sign off.
    - Appropriate members of the client's staff must review relevant areas of the design (including operations, training materials, system administration).
    - The acceptance tests must belong to the client.

Building Reliable Software: Complexity

- The human mind can encompass only limited complexity
    - Comprehensibility
    - Simplicity
    - Partitioning of complexity
- A simple component is easier to get right than a complex one

Building Reliable Software: Change

- Changes can easily introduce problems
    - Change management
    - Source code management and version control
    - Tracking of change requests and bug reports
    - Procedures for changing requirements specifications, designs and other documentation
    - Regression testing
    - Release control
- When adding new functions or fixing bugs it is easy to write patches that violate the systems architecture or overall program design. This should be avoided as much as possible.
- Be prepared to modify the architecture to keep a high quality system.

Building Reliable Software: Fault Tolerance

- Aim:
    - A system that continues to operate when problems occur.

- Examples:
    - Invalid input data (e.g., in a data processing application)
    - Overload (e.g., in a networked system)
    - Hardware failure (e.g., in a control system)
- General Approach:
    - Failure detection
    - Damage assessment
    - Fault recovery
    - Fault repair

Fault Tolerance: Recovery

- Backward recovery
    - Record system state at specific events (checkpoints). After failure, recreate state at last checkpoint.
    - Combine checkpoints with system log (audit trail of transactions) that allows transactions from last checkpoint to be repeated automatically.
- Recovery software is difficult to test
    - Example: After an entire network is hit by lightning, the restart crashes because of overload. (Problem of incremental growth.)

Building Reliable Software: Small Teams and Small Projects

- Small teams and small projects have advantages for reliability
    - Small group communication cuts need for intermediate documentation, yet reduces misunderstanding.
    - Small projects are easier to test and make reliable.
    - Small projects have shorter development cycles. Mistakes in requirements are less likely and less expensive to fix.
    - When one project is completed it is easier to plan for the next.
- Improved reliability is one of the reasons that agile development has become popular over the past few years.

Reliability Metrics

- Reliability
    - Probability of a failure occurring in operational use.
- Traditional measures for online systems
    - Mean time between failures
    - Availability (up time)
    - Mean time to repair
- Market measures
    - Complaints

- Customer retention

Reliability Metrics for Distributed Systems

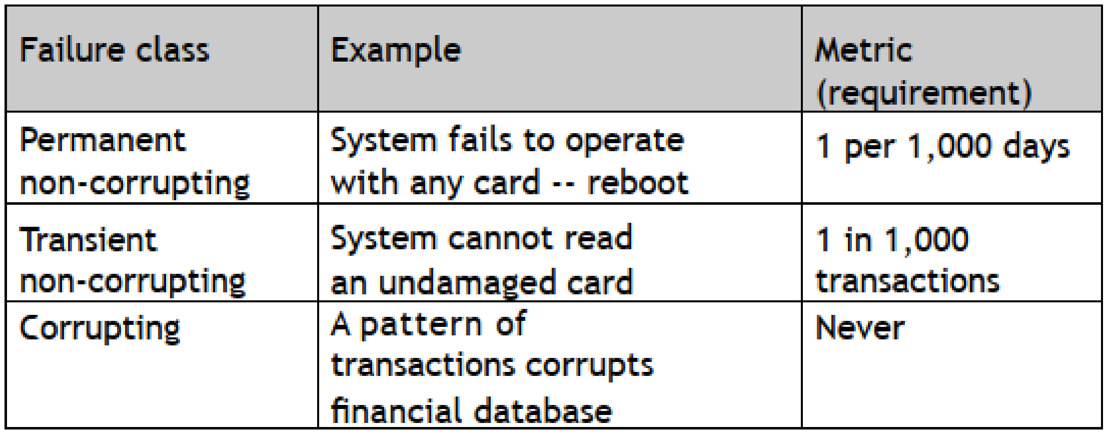
- Traditional metrics are hard to apply in multi- component systems
    - A system that has excellent average reliability might give terrible service to certain users.
    - In a big network, at any given moment something will be giving trouble, but very few users will see it.
    - When there are many components, system administrators rely on automatic reporting systems to identify problem areas.

Metrics: User Perception of Reliability

- Perceived reliability depends upon:
    - user behavior
    - set of inputs
    - pain of failure
- User perception is influenced by the distribution of failures
    - A personal computer that crashes frequently, or a machine that is out of service for two days every few years.
    - A database system that crashes frequently but comes back quickly with no loss of data, or a system that fails once in three years but data has to be restored from backup.
    - A system that does not fail but has unpredictable periods when it runs very slowly.

Reliability Metrics for Requirements

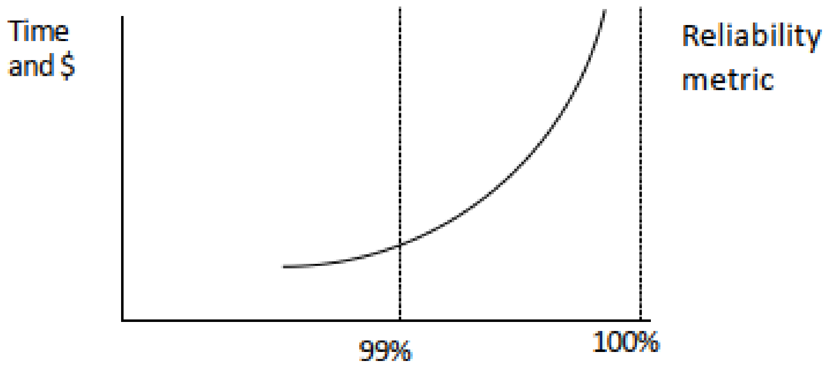
- Example: ATM card reader



Failure class	Example	Metric (requirement)
Permanent non-corrupting	System fails to operate with any card -- reboot	1 per 1,000 days
Transient non-corrupting	System cannot read an undamaged card	1 in 1,000 transactions
Corrupting	A pattern of transactions corrupts financial database	Never

Metrics: Cost of Improved Reliability

- Example. Many supercomputers average 10 hours productive work per day. How do you spend your money to improve reliability?



Example: Central Computing System

- A central computer system (e.g., a server farm) is vital to an entire organization (e.g., an Internet shopping site). Any failure is serious.
- Step 1: Gather data on every failure
  - Create a database that records every failure
  - Analyze every failure:
    - hardware software (default)
    - environment (e.g., power, air conditioning)
    - human (e.g., operator error)
- Step 3: Invest resources where benefit will be maximum, e.g.,
  - Priority order for software improvements
  - Changed procedures for operators
  - Replacement hardware
  - Orderly restart after power failure

Building Reliable Systems: Two Principles

- For a software system to be reliable:
  - Each stage of development must be done well, with incremental verification and testing.
  - Testing and correction do not ensure quality, but reliable systems are not possible without thorough testing.

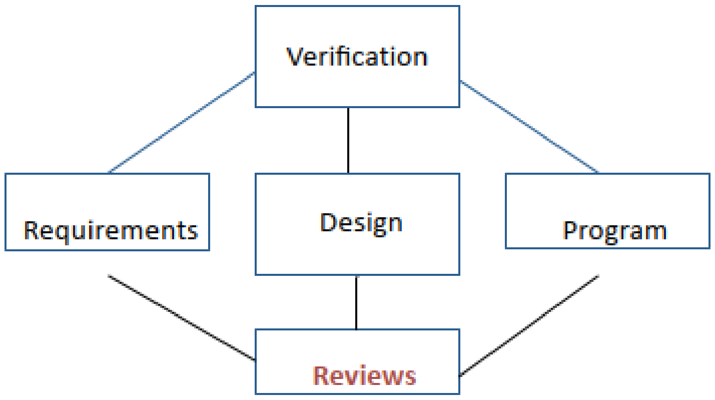
Static and Dynamic Verification

- Static verification:
  - Techniques of verification that do not include execution of the software.
  - May be manual or use computer tools.

- Dynamic verification :
    - Testing the software with trial data.
    - Debugging to remove errors.

Static Verification: Reviews

- Reviews are a form of static verification that is carried out throughout the software development process.



Reviews

- Reviews are a fundamental part of good software development
- Concept
    - Colleagues review each other's work:
        - can be applied to any stage of software development, but particularly valuable to review program design or code
        - can be formal or informal
- Preparation
    - The developer(s) provides colleagues with documentation (e.g., models, specifications, or design), or code listing.
    - Participants study the materials in advance.
- Meeting
    - The developer leads the reviewers through the materials, describing what each section does and encouraging questions.

The Review Meeting

- A review is a structured meeting
- Participants and their roles:
    - Developer(s): person(s) whose work is being reviewed
    - Moderator: ensures that the meeting moves ahead steadily
    - Scribe: records discussion in a constructive manner Interested parties: other developers on the same project

    - Outside experts: knowledgeable people who are not working on this project
    - Client: representatives of the client who are knowledgeable about this part of the process

Benefits of Reviews

- Extra eyes spot mistakes, suggest improvements
- Colleagues share expertise; helps with training
- Incompatibilities between components can be identified
- Gives developers an incentive to tidy loose ends
- Helps scheduling and management control

Successful Reviews

- To make a review a success:
    - Senior team members must show leadership
    - Good reviews require good preparation by everybody
    - Everybody must be helpful, not threatening Allow plenty of time and be prepared to continue on another day

Static Verification: Pair Design and Pair Programming

- Concept: achieve benefits of review by shared development
- Two people work together as a team:
    - design and/or coding testing and system integration documentation and hand-over
- Benefits include:
    - two people create better software with fewer mistakes
    - cross training Many software houses report excellent productivity

Static Verification: Program Inspections

- Formal program reviews whose objective is to detect faults
    - Code is read or reviewed line by line
    - 150 to 250 lines of code in 2 hour meeting
    - Use checklist of common errors
    - Requires team commitment, e.g., trained leaders
- So effective that it is claimed that it can replace unit testing

Static Verification: Analysis Tools

- Program analyzers scan the source of a program for possible errors and anomalies.
    - Control flow: Loops with multiple exit or entry points
    - Data use: Undeclared or uninitialized variables, unused variables, multiple assignments, array bounds Interface faults:

    - Parameter mismatches, non-use of functions results, uncalled procedures
    - Storage management: Unassigned pointers, pointer arithmetic
- Static analysis tools
    - Cross-reference table: Shows every use of a variable, procedure, object, etc.
    - Information flow analysis: Identifies input variables on which an output depends.
    - Path analysis: Identifies all possible paths through the program.

Dynamic Verification: Stages of Testing

- Testing is most effective if divided into stages
    - User interface testing
    - Unit testing
        - unit test
    - System testing
        - integration test
        - function test
        - performance test
        - installation test
    - Acceptance testing (carried out separately)

Testing Strategies

- Bottom-up testing
    - Each unit is tested with its own test environment. Used for all systems.
- Top-down testing
    - Large components are tested with dummy stubs. Particularly useful for: user interfaces work-flow client and management demonstrations
- Stress testing
    - Tests the system at and beyond its limits. Particularly useful for:
        - real-time systems
        - transaction processing
    - Most systems require a combination of all three strategies.

Methods of Testing

- Open box testing
    - Testing is carried out by people who know the internals of what they are testing.
    - Example: Tick marks in a graphing package

- Closed box testing
  - Testing is carried out by people who do not know the internals of what they are testing.
  - Example: Educational demonstration that was not foolproof

Testing: Unit Testing

- Tests on small sections of a system, e.g., a single class
- Emphasis is on accuracy of actual code against specification
- Test data is usually chosen by developer(s) based on their understanding of specification and knowledge of the unit
- Can be at various levels of granularity
- Can be open box or closed box: by the developer(s) of the unit or by special testers
- If unit testing is not thorough, system testing becomes almost impossible. If you are working on a project that is behind schedule, do not rush the unit testing.

Testing: System and Sub-System Testing

- Tests on components or complete system, combining units that have already been thoroughly tested
- Emphasis on integration and interfaces
- Trial data that is typical of the actual data, and/or stresses the boundaries of the system, e.g., failures, restart
- Carried out systematically, adding components until the entire system is assembled
- Can be open or closed box: by development team or by special testers
- System testing is finished fastest if each component is completely debugged before assembling the next

Dynamic Verification: Test Design

- Testing can never prove that a system is correct. It can only show that either (a) a system is correct in a single case, or (b) that it has an error.
  - The objective of testing is to find errors or demonstrate that program is correct in specific instances.
  - Testing is never comprehensive.
  - Testing is expensive.

Test Cases

- Test cases are specific tests that are chosen because they are likely to find specific problems. Test cases are chosen to balance expense against chance of finding serious errors.

    - Cases chosen by the development team are effective in testing known vulnerable areas.
    - Cases chosen by experienced outsiders and clients will be effective in finding gaps left by the developers.
    - Cases chosen by inexperienced users will find other errors.

Variations in Test Sets

- A test suite is the set of all test cases that apply to a system or component of a system.
- When running tests, there are some errors that occur only under certain circumstances, e.g., when certain other software is running on the same machine, when tasks are scheduled in specific sequences, or with unusual data sets, etc.
- Therefore it is customary for each test run to vary some of the test cases systematically, and to change the order in which the tests are made, etc.

Incremental Testing (e.g., Daily Testing)

- Spiral development and incremental testing
    - Create a first iteration that has the structure of the final system and some basic functionality.
    - Create an initial set of test cases.
    - Check-in changes to the system on a daily basis, rebuild entire system daily.
    - Run a comprehensive set of test cases daily, identify and deal with any new errors.
    - Add new test cases continually.
- Many large software houses, e.g., Microsoft, follow this procedure with a daily build of the entire system and comprehensive sets of test cases. For a really big system this may require hundreds or even thousands of test computers and a very large staff of testers.

Dynamic Verification: Regression Testing

- When software is modified, regression testing is used to check that modifications behave as intended and do not adversely affect the behavior of unmodified code.
- After every change, however small, rerun the entire testing suite.

Regression Testing: Program Testing

- Collect a suite of test cases, each with its expected behavior.
- Create scripts to run all test cases and compare with expected behavior. (Scripts may be automatic or have human interaction)

- When a change is made to the system, however small (e.g., a bug is fixed), add a new test case that illustrates the change (e.g., a test case that revealed the bug).
- Before releasing the changed code, rerun the entire test suite.

Incremental Testing: an Example

- Example
    - A graphics package consisting of a pre-processor, a runtime package (set of classes), and several device drivers.
- Starting point
    - A prototype with a good overall structure, and most of the functionality, but hastily coded and not robust.
- Approach
    - On a daily cycle:
        - Design and code one small part of the package (e.g., an interface, a class, a dummy stub, an algorithm within a class)
        - Integrate into prototype.
        - Create additional test cases if needed.
        - Regression test.

Documentation of Testing

- Every project needs a test plan that documents the testing procedures for thoroughness, visibility, and for future maintenance.
- The test plan should include:
    - Description of testing approach.
    - List of test cases and related bugs.
    - Procedures for running the tests.
    - Test analysis report.

Fixing Bugs

- Isolate the bug
    - Intermittent → repeatable
    - Complex example → simple example
- Understand the bug and its context
    - Root cause
    - Dependencies
    - Structural interactions
- Fix the bug
    - Design changes
    - Documentation changes

    - Code changes
    - Create new test case

Moving the Bugs Around

- Fixing bugs is an error-prone process
    - When you fix a bug, fix its environment. Bug fix need static and dynamic testing.
    - Repeat all tests that have the slightest relevance (regression testing).
- Bugs have a habit of returning
    - When a bug is fixed, add the failure case to the test suite for future regression testing.
- Persistence
    - Most people work around a problem. The best people track down the root cause and ?x it forever!

Difficult Bugs

- Some bugs may be extremely difficult to track down and isolate. This is particularly true of intermittent failures.
    - A large central computer stops a few times every month with no dump or other diagnostic.
    - A database load dies after running for several days with no diagnostics.
    - An image processing system runs correctly, but uses huge amounts of memory.
- Such problems may require months of effort to track down.

Bugs in Hardware

- Three times in my career I have encountered hardware bugs:
    - The film plotter with the missing byte (1:1023)
    - Microcode for virtual memory management
    - The Sun page fault Each problem was actually a bug in embedded software/firmware

When Fixing a Bug Creates a Problem for Customers

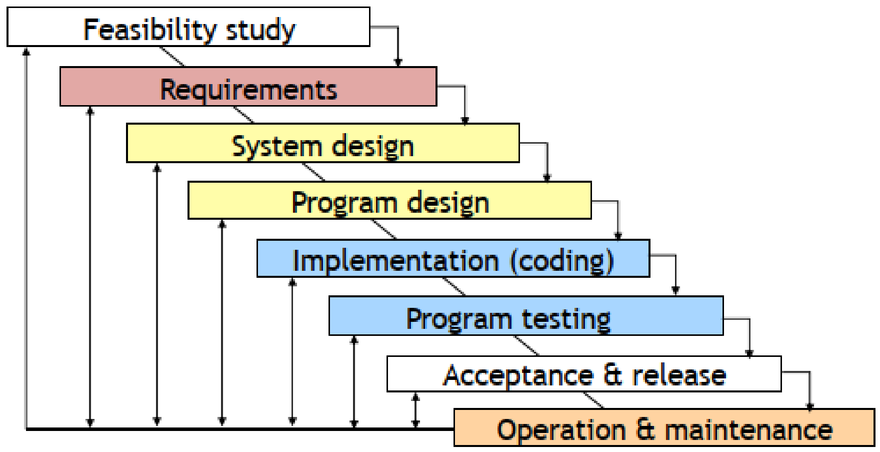
- Sometimes customers will build applications that rely upon a bug. Fixing the bug will break the applications.
    - The graphics package with rotation about the Z-axis in the wrong direction.
    - An application crashes with an emulator, even though the emulator is bug free. (Compensating bug problem)
    - The 3-pixel rendering problem with Internet Explorer.

- With each of these bugs the code was easy to fix, but releasing it would have caused problems for existing programs.

Acceptance Testing

- The complete system, including documentation, training materials, installation scripts, etc. is tested against the requirements by the client, assisted by the developers.
    - Each requirement is tested separately.
    - Scenarios are used to compare the expected outcomes with what the system does.
    - Emphasis is placed on how the system handles problems, errors, restarts, and other difficulties.
    - Is the system we have built, the system that you wanted? Does it meet your requirements?

Acceptance Testing in the Modified Waterfall Model



Acceptance Testing with Iterative Development

- Iterative development If the client is properly involved in the development cycle:
    - The client will have tested many parts of the system, e.g., → At the end of each iteration → During user testing
    - Problems and suggestions for improvement will have been incorporated into the system.
- BUT: There must still be an acceptance test of the final system before it is released.

Acceptance Testing with Agile Development

- Acceptance testing is particularly important with agile development, since each sprint should end with fully tested code.

- Each sprint should be a complete development process, ending with acceptance testing by the client.
- If several sprints build on each other, each sprint may need to repeat the acceptance tests for earlier sprints to check that they are still met.

Resources for Acceptance Testing

- Acceptance Testing is a important part of a software project
  - It requires time on the schedule
  - It may require substantial investment in test data, equipment, and test software.
  - Good testing requires good people.
  - Help systems and training materials are important parts of acceptance testing.

Acceptance Tests

- Closed box by the client without knowledge of the internals
- The entire system is tested as a whole
- The emphasis is on whether the system meets the requirements
- The tests should use real data in realistic situations, with actual users, administrators, and operators
- The acceptance tests must be successfully completed before the new system can go live or replace a legacy system. Completion of the acceptance tests may be a contractual requirement before the system is paid for.

Techniques for Release

- The transition from the previous version of a production system to a new release is challenging.
- Parallel Testing: Clients operate the new system alongside the old production system with same data and compare results
- Alpha Testing: Clients operate the system in a realistic but non- production environment
- Beta Testing: Clients operate the system in a carefully monitored production environment

Release: Parallel Testing

- For data processing systems, such as financial systems, payroll, etc., the old and the new systems are run together for several productions cycles to check that the new system replicates the functionality of the old.
  - Requires two sets of everything (staff, equipment, etc.).
  - Requires software to control changeover (e.g., do not mail two sets of payments).

    - Requires automated scripts to compare results. Parallel testing may take several months. Often, the new system will be brought into production in phases.

Release: Alpha and Beta Testing

- Alpha testing can be done with software that lacks some functionality.
- Beta testing requires fully functional system.
- Alpha and Beta testing must be managed
- If you simply make versions of the software available to clients:
    - Many clients will never use it.
    - Other clients will use a few features.
    - Only a few clients will test it systematically.
    - Only a few clients will report problems systematically. What incentives can you give clients to test your software for you?
    - Financial (e.g., discounts on products)
    - Prestige (e.g., recognition, publicity)

Delivery: Summary

- A good delivery package results in
    - happy client
    - happy users
    - less expense in support and maintenance
- But many projects rush the packaging, help systems, and training materials, give them to the least experienced members of the team, do not test them properly, and generally neglect this part of the software process.

Training

- Time and money spent on training is usually well spent:
    - one-on-one
    - in-house training
    - training courses
    - distance education
    - online tutorials
- Development team provides information for training materials:
    - users (perhaps several categories)
    - system administrators
    - system maintainers
    - trainers

Training and Usability

- A well-designed system needs less training
    - good conceptual model
    - intuitive interfaces
- Different skill levels need different types of training
    - skilled users work from the conceptual model
    - less-skilled users prefer cookbook sets of instructions
    - occasional users will forget complex details, but remember general structure

Help Systems

- Resources
    - A good help system is a major sub-project (time- consuming, expensive)
    - A good help system saves user time and support sta? (time-saving, cost-saving)
- Help system design
    - Users need many routes to find information (index by many terms, examples, mini-tutorials, etc.)
    - Help systems need to be tested with real users.

Categories of Documentation

- Software development
    - Requirements, design
    - Source code, test plans and results
- User
    - Introductory (various audiences)
    - User manual
    - Web site of known problems, FAQ, etc.
- System administrator and operator
    - System manuals Business
    - License, contract, etc.