INTRODUCTION

- What is software?
  - Computer programs +
  - Configuration data and files +
  - User and system documentation.
- What is software engineering?
  - An engineering discipline which is concerned with all aspect of software production.*.

SE vs. CS

- Difference between software engineering and computer science:
  - Computer science is concerned with theory and fundamentals
  - Software engineering is concerned with practical aspects of developing and delivering software
- Software engineering challenges:
  - Coping with legacy systems*
  - Coping with increasing diversity (heterogeneous systems-many types of hardware/software)
  - Coping with faster, cheaper

Software Process

- What is a software process?
  - A set of activities and associated results which produce a software product
- Four fundamental process activities:
  1. Software specification
  2. Software development
  3. Software validation (i.e. It does what it intended to do based on software spec.)
  4. Software evolution (i.e. Change/enhancement/maintenance)

Software Process Models

- What is a software process model?*
  - A representation of software process from a specific perspective.
- Examples of software process models:
  - Workflow model: Sequence of activities in the process along with their inputs, outputs and dependencies.
  - Data-flow or activity model: A set of activities that carry out some data transformation (input→output).
  - Role/action model: Represents roles of people involved in the software process and activities for which they are responsible.

Software Development Models

- Different models (paradigms) of software development:
  1. The Waterfall approach: Complete one phase (e.g. req., design, code, test) before going to next.
  2. Evolutionary development: Build quick, modify, and redo component until completion (Prototyping often used).*.
  3. Formal transformation: Transform Specifications, using mathematical methods, to a program; guarantee correctness.
  4. System assembly from reusable components: Assemble already existing parts.
  5. Incremental method: Design and deliver parts as they become available (i.e. many small deliverables).

Attributes of Good Software

- Maintainability (more than 40% of software cost is due to maintenance)
- Dependability (Reliability, security, safety)*
- Efficiency (memory, CPU time)
- Usability (Good UI and Documentation)

Ethical Responsibility

- Confidentiality (respect confidentiality of employer/client)
- Competence (avoid misrepresenting the level of competence)*
- Intellectual property rights (local laws, patents, copyrights)
- Computer misuse (Viruses, hacking, information theft)

Software Processes

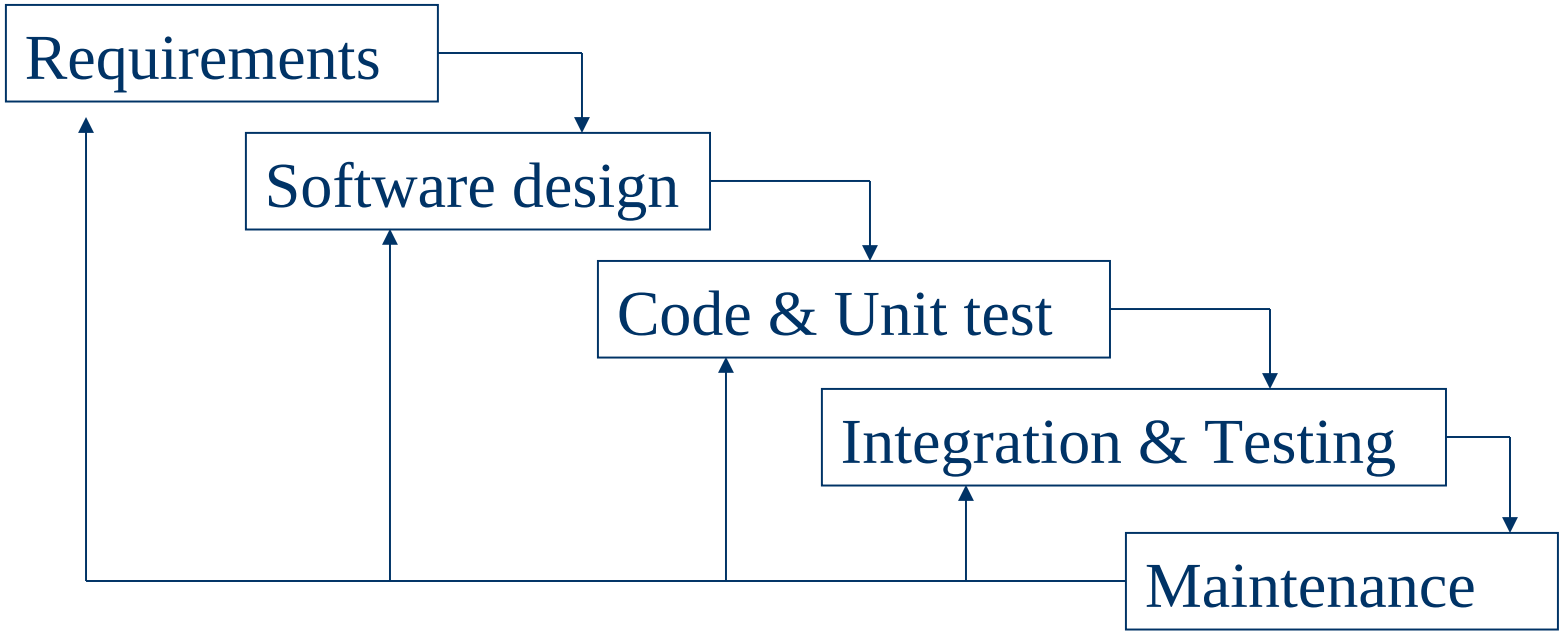
- A Software process is a set of activities and associated results which lead to the production of a software product.
- Activities Common to all software processes:
  - Software specification (initiation, requirements),
  - Software design and implementation,
  - Software validation (testing, QA),
  - Software evolution (enhancement/maintenance)

Software Process Models

- A software process model is an abstract representation of a software (i.e. a roadmap)
- Software process models:
  - Waterfall model
  - Evolutionary development
  - Formal systems development
  - Reuse-based development
- Hybrid software process models:
  - Incremental development
  - Spiral development

Waterfall model

- Also referred to as “Life cycle”, conducted in five stand-alone phases:

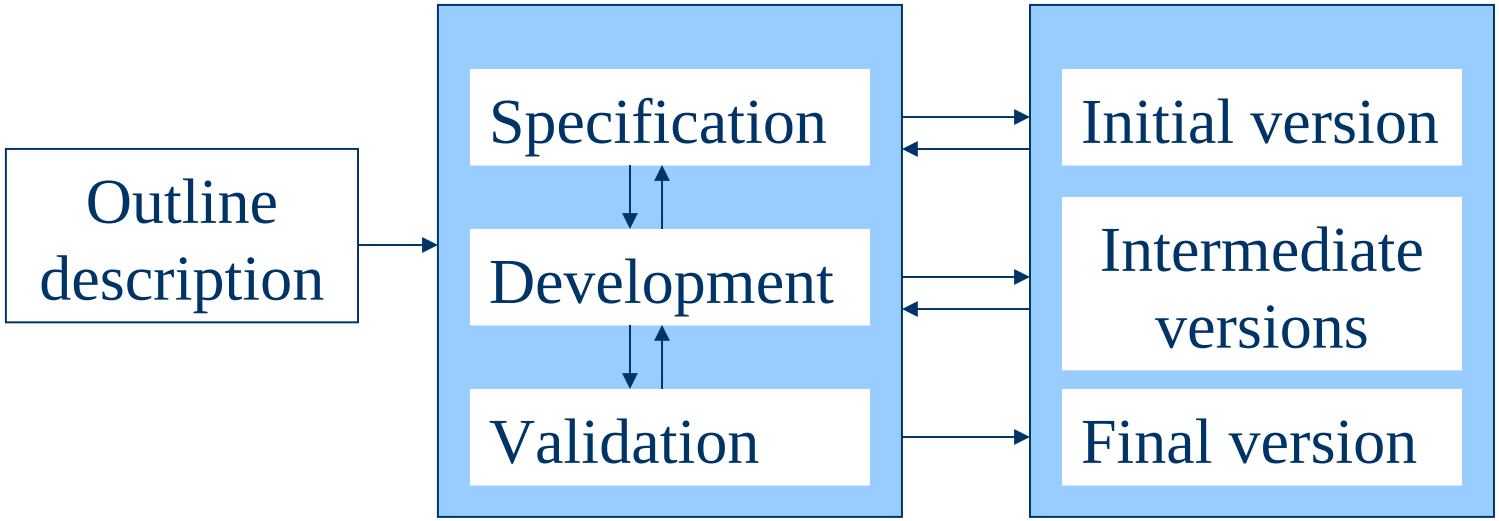


Waterfall model: Plus and Minuses

- Advantages of the model:
  - Simple to follow
  - Relatively simple to track progress
  - Good structural design
- Challenges:
  - In practice, often phases overlap
  - Hard to modify and implement changes
  - Need complete requirements from customers to start (the biggest challenge)

Evolutionary development

- Develop an initial implementation, expose to users comments, refine until satisfied:



Evolutionary development types

There are two types of evolutionary development:

- Exploratory development
  - Start with requirements that are well defined
  - Add new features when customers propose new requirements
- Throw-away prototyping
  - Objective is to understand customer’s requirements (i.e. they often don’t know what they want, hence poor requirements to start
  - Use means such as prototyping to focus on poorly understood requirements, redefine requirements as you progress

Evolutionary development: advantages and challenges

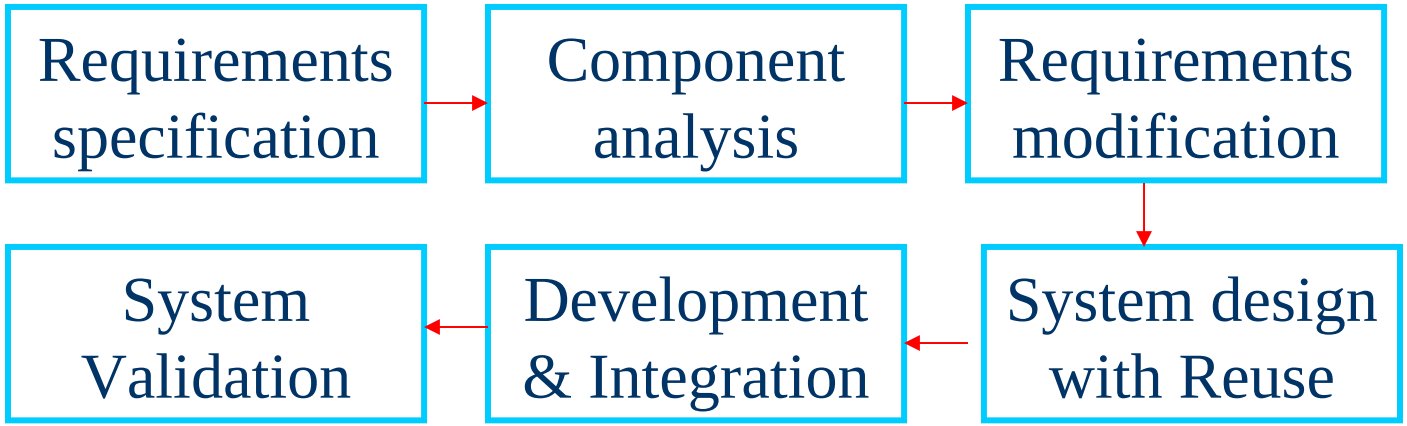
- Advantages:
  - Happier customers since you help them define requirements
  - Flexibility in modifying requirements
  - Prototypes are very visual, hence no ambiguities
- Challenges:
  - Hard to trace the “process” due to the ad-hoc nature
  - Systems are often poorly structured
  - Special tools and techniques may be required (for rapid development) that may be incompatible
  - Not cost-effective to produce documents

Formal systems development

- Similar to Waterfall model however development is based on formal mathematical transformation of system specification to an executable program
- Software requirements specifications is refined into detailed formal specification which expressed in mathematical notion
- Design, implementation and testing are replaced by a series of mathematical transformations
- Require specialized expertise
- usually used where safety is highly required
- Not often used

Reuse–oriented development

- Reuse-oriented approach relies on a large base of reusable software components!
- Design system to capitalize on the existing components

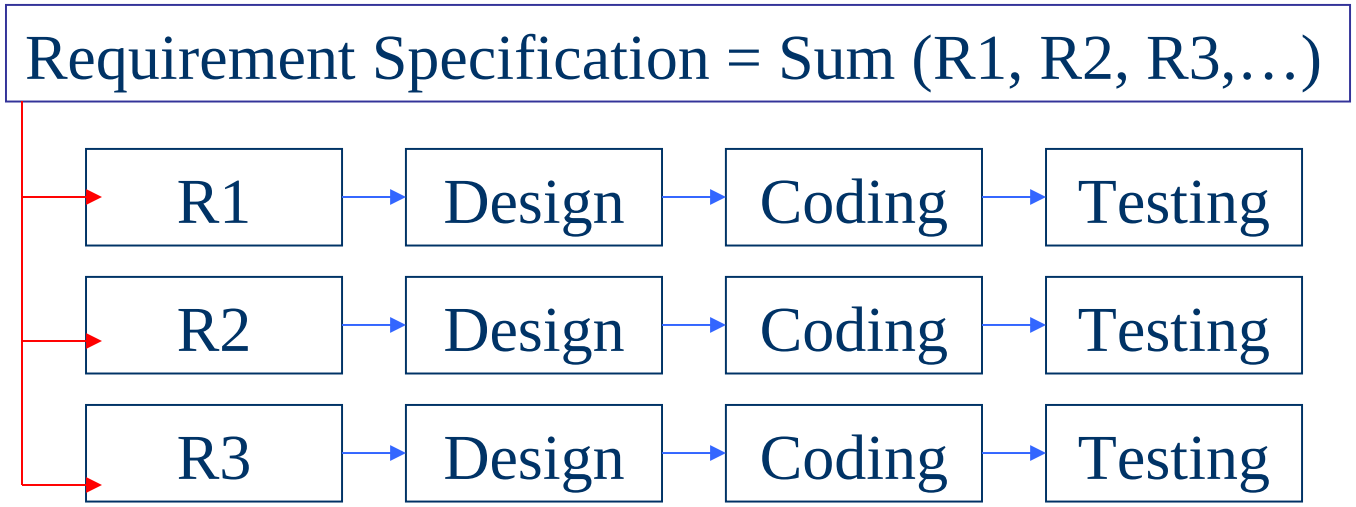


Reuse–oriented: Plus and Minuses

- Advantages:
  - Reduced cost and risk
  - Fast delivery
- Challenges:
  - Requires a large enough component base
  - Some control over the system evolution is lost as new versions of reusable components are not under the control of organization using the component
  - Potential issues in backward/forward compatibility

Incremental development

- A hybrid model where the software specification, design, implementation, and testing is broken down into a series of increments which are developed and delivered



Incremental development: Advantages and Challenges

- Advantages:
  - Products delivered incrementally hence faster
  - Lower risk of overall project failure
  - Requirements are implemented based on priority
- Challenges:
  - Relationship between different increments may be cumbersome or non-cohesive
  - Size of each increment and the number of increments may cause challenges

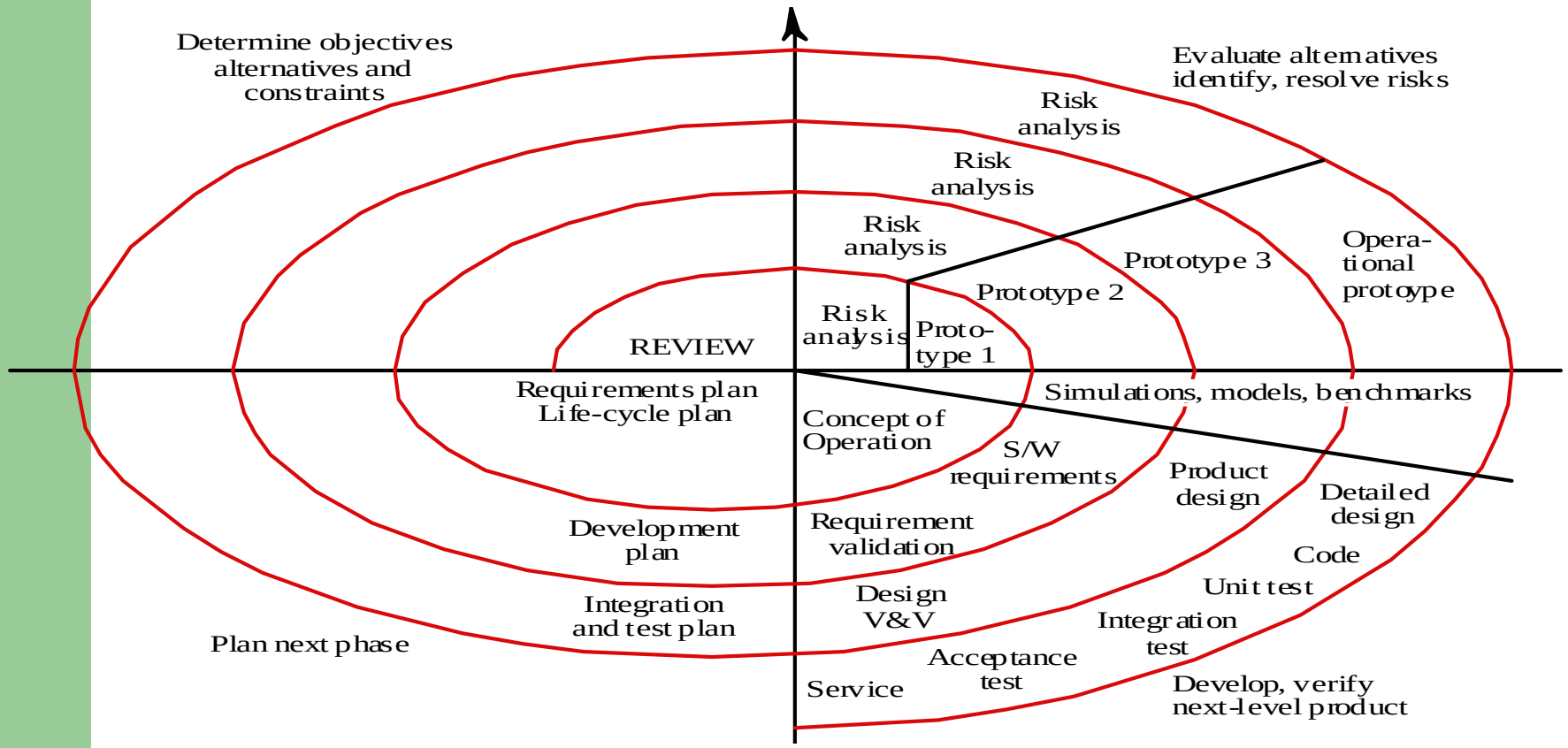
Spiral development

- A hybrid model where the development of the system spirals outward from an initial outline through to the final developed system
- Each loop in the spiral represents a phase of the software process
- Example: the innermost loop might be concerned with system feasibility, next loop with system requirements, next loop with system design and so on.

Spiral development loops

- Each loop in the spiral is split into four sectors:
  1. Object Setting: set specific object for that phase
  2. Risk assessment and reduction
  3. Development and validation: select a development model based on risk levels
  4. Planning: decide if a next loop is required

Spiral Development



Spiral development: Plus & Minuses

- Advantages:
  - Explicit consideration of risks (alternative solutions are evaluated in each cycle)
  - More detailed processes for each development phase
- Disadvantages:
  - Cost
  - Sometime difficult to implement or too time consuming

Software specification activities

1. Feasibility study: doable? Cost-effective? Market timing appropriate?
2. Requirements analysis: derive requirements based on existing system, customer input, marketing information, and all other sources
3. Requirements specification: create formal requirements document
4. Requirement validation: check requirements for realism, consistency and completeness

Software design & implementation activities

1. Architectural design (the big picture)
2. Abstract specification (for each sub-system)
3. Interface design (with other sub-systems)
4. Component design
5. Data structure design
6. Algorithm design
7. Implementation environment selection

Software validation activities

1. Unit testing
2. Module testing
3. Sub-system testing
4. System testing
5. Acceptance testing

Computer-Aided Software Engineering (CASE)

- CASE is the name given to software that is used to support software process activities such as requirements engineering, design, program development and testing.
- CASE tools include:
  - Design editors, compilers, data dictionaries, debuggers, and other system building tools…

Project Management

- Software management is distinct and often more difficult from other engineering managements mainly because:
  - Software product is intangible
  - There are no standard software processes
  - Large software projects are usually different from previous projects

Management Activities

Most managers do some or all of the following regularly at all times:

- Proposal writing
- Project planning and scheduling
- Project costing
- Project monitoring and reviews
- Personnel selection and evaluation
- Report writing and presentation

Project planning

- Effective management depends on planning the progress of the project in detail and allow for deviations
- Project plan evolves as project progresses and better information becomes available
- Project Constraints, milestones and deliverables must be identified clearly and timely

Project planning process

Establish project constraints

Make initial assessment of the project parameters

Define project Milestones and deliverables

While (project has not been completed or cancelled) **Loop**

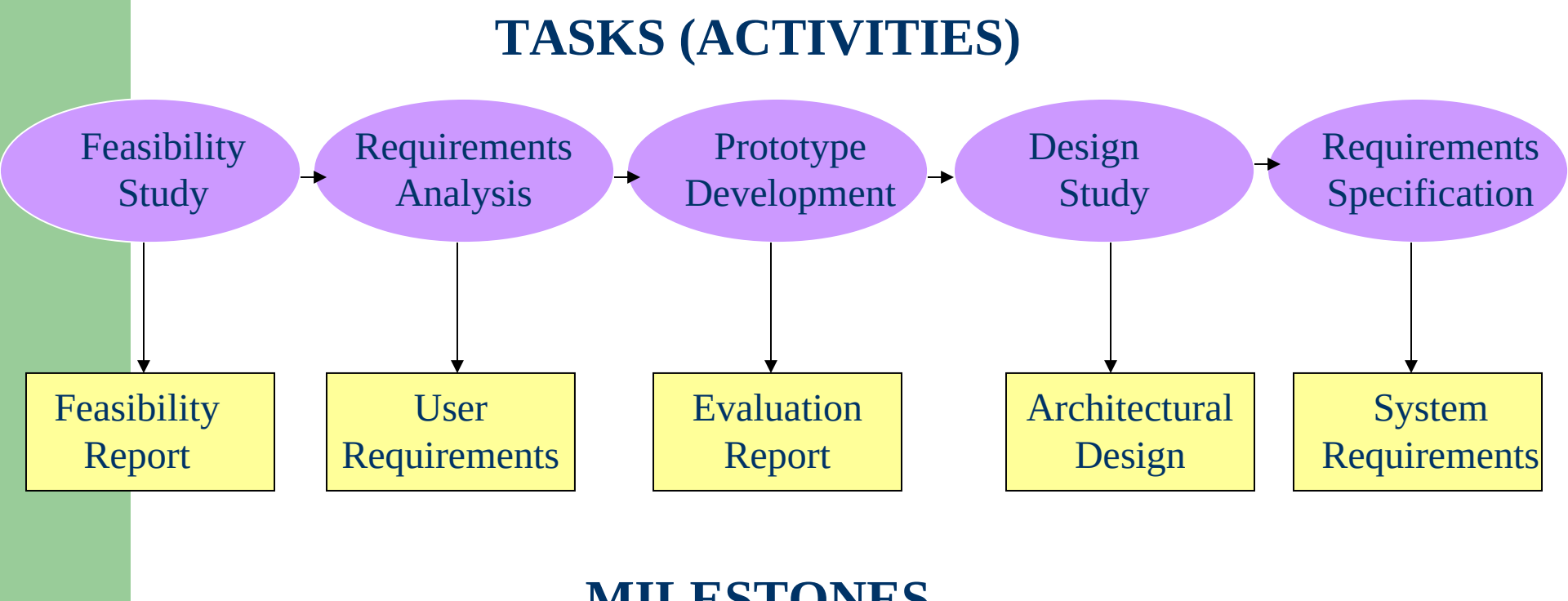
- Draw up project schedule
- Initial activities according to schedule
- Wait (for a while) ➔ review project progress
- Revise estimate of project parameters ➔ update project plan
- Renegotiate project constraints and deliverables
- **If** (problem arise) **Then**
- Initiate technical review and possible revision

End loop

Project Plan Sections:

1. Introduction (objective, constraints→ time, budget)
2. Project Organization (People, teams, roles)
3. Risk Analysis
4. Hardware & Software resource requirements
5. Work breakdown (tasks, milestones, deliverables)
6. Project schedule (time estimates per task, people allocation, task dependencies, milestone dates)
7. Monitoring and reporting mechanism (reports)

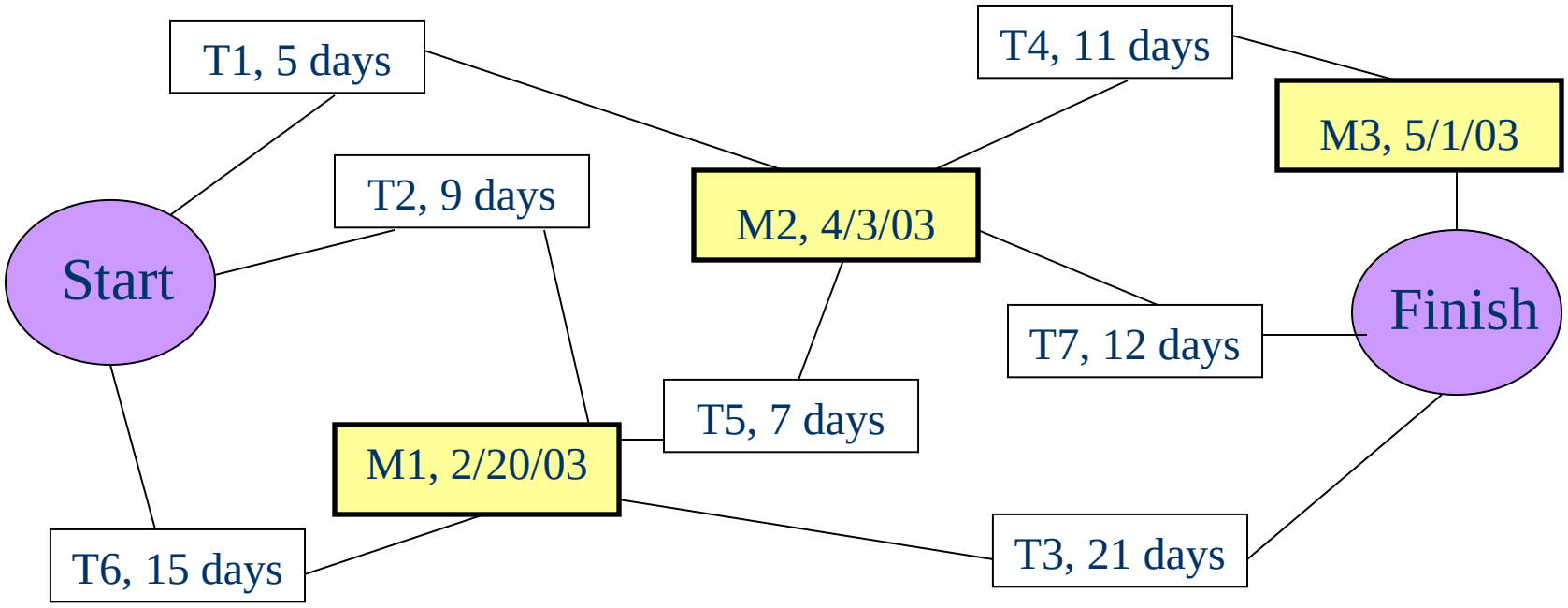
Milestones and Deliverables



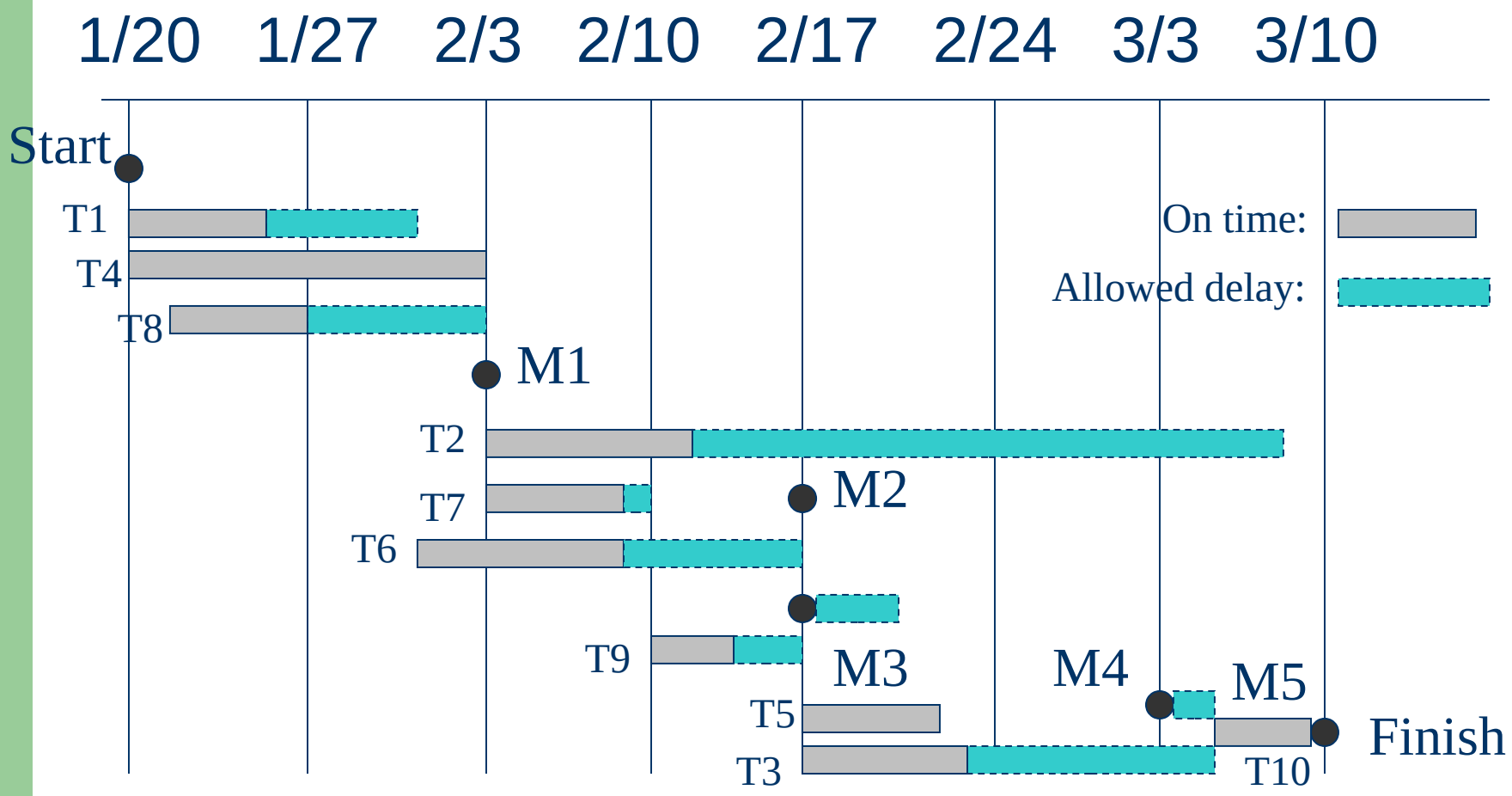
Task Durations and Dependencies

TASK	Duration (days)	Dependencies
T1	8	
T2	6	T1
T3	20	T4, T5, M1
T4	9	T1
T5	17	T3, M2

Activity Network



Activity Bar (Gantt) Chart



Risk Management

Stages for managing risks:

1. Risk Identification (project, product, business, technology)
2. Risk Analysis (likelihood and consequences, low/medium/high)
3. Risk Planning (avoiding, minimizing risk effects, contingency planning)
4. Risk Monitoring (constant assessment, mitigation i.e. reduce risk severity)

Requirements Engineering

- Definition: **Description and Specifications of a system**
- Topics covered:
  - Functional and Non-functional requirement
  - User Requirements
  - System requirements
  - The software requirements document

Software Requirements

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed
- Requirements may be functional or non-functional
  - Functional requirements describe system services or functions
  - Non-functional requirements is a constraint on the system or on the development process

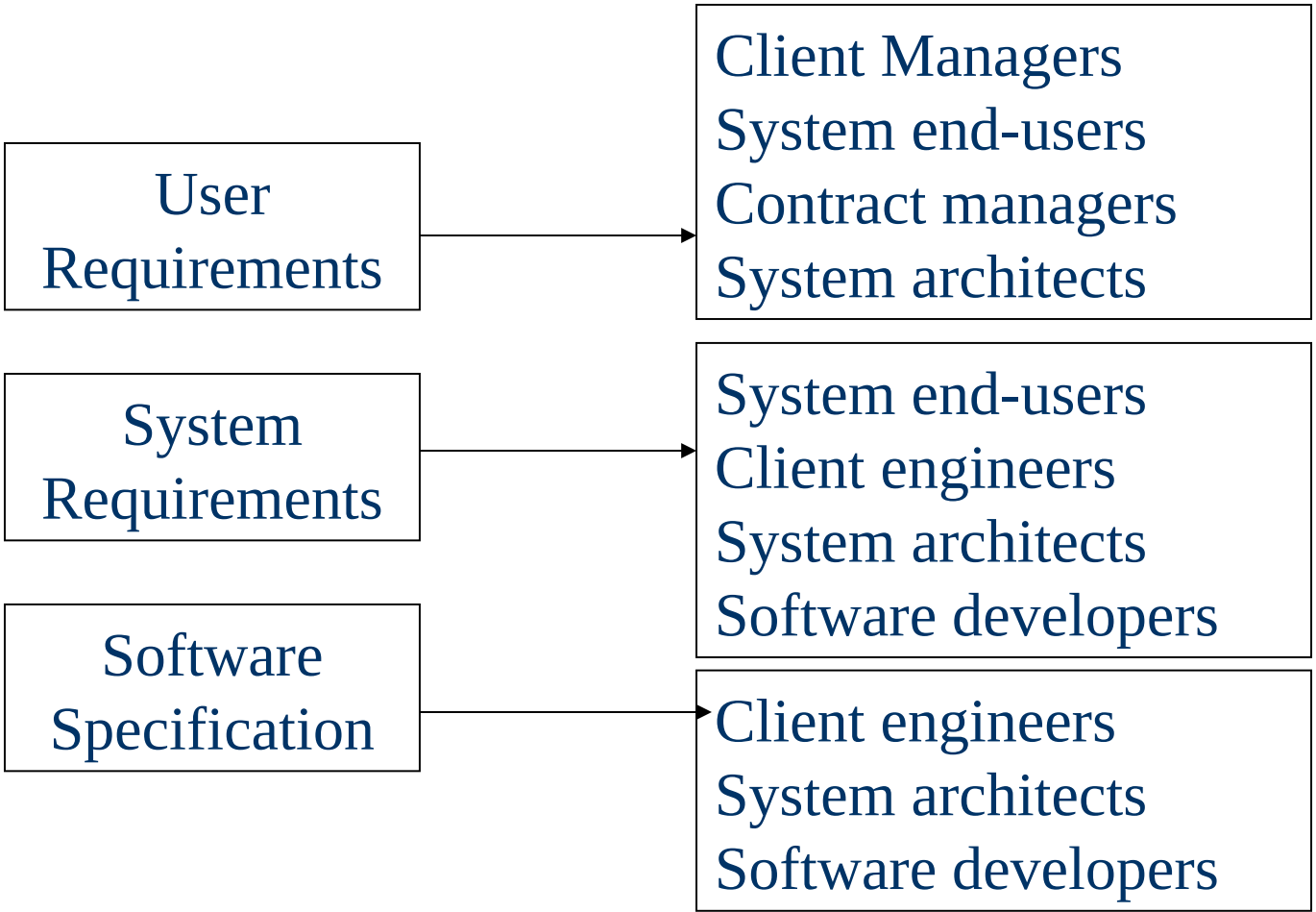
What is a requirement?

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

Types of requirements

- User requirements
  - Statements in natural language (NL) plus diagrams of the services the system provides and its operational constraints. Written for customers
- System requirements
  - A structured document setting out detailed descriptions of the system services. Written as a contract between client and contractor
- Software specification
  - A detailed software description which can serve as a basis for a design or implementation. Written for developers

Requirements Targets



Requirements Types:

1. Functional requirements: services the system should provide
2. Non-functional requirements: constraints on the services of functions offered by the system. e.g. speed, time to market
3. Domain requirements: related to the application domain of the system (may be functional or non-functional requirements)

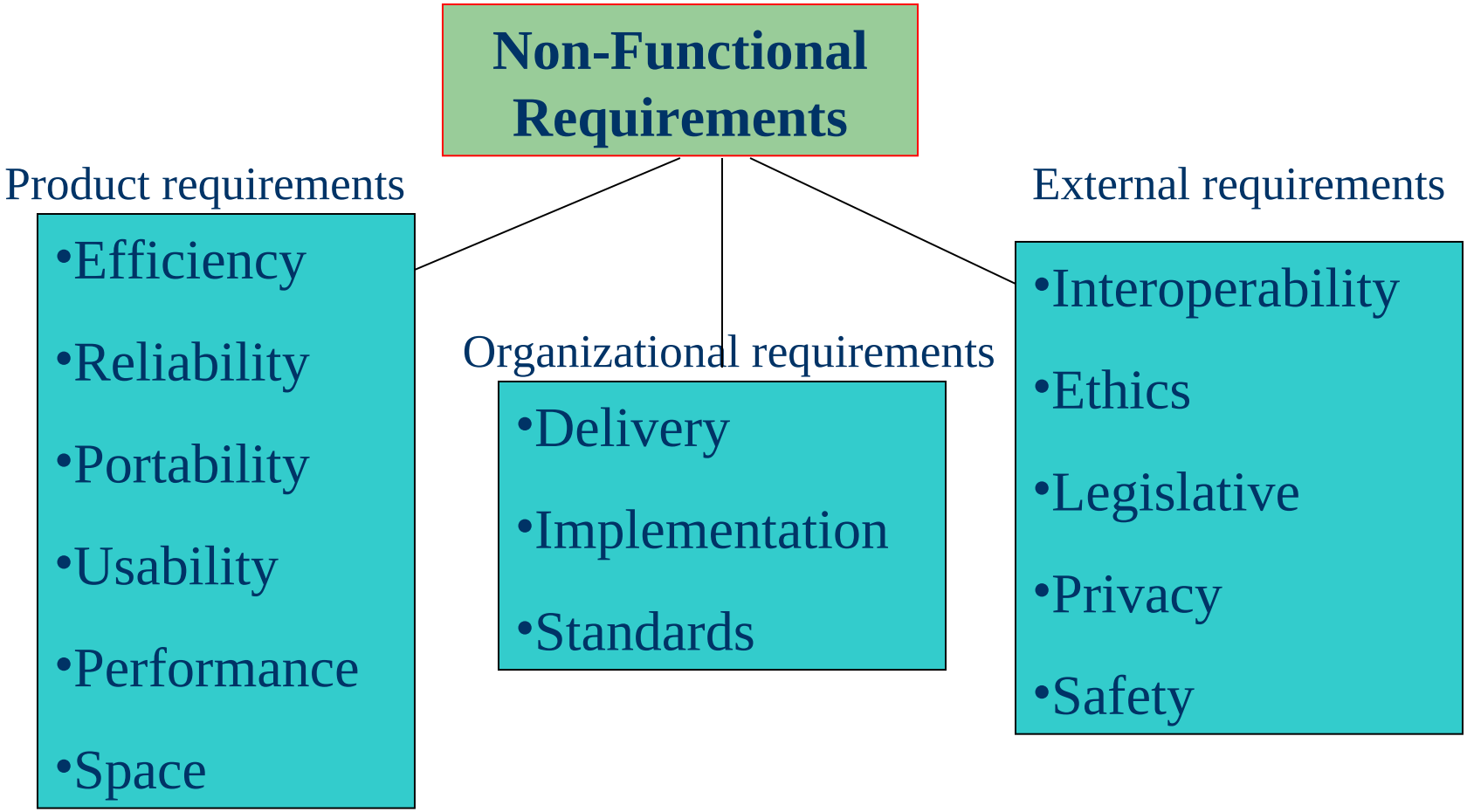
Functional requirements

- Functionality or services that the system is expected to provide.
- Functional requirements may also explicitly state what the system shouldn’t do.
- Functional requirements specification should be:
  - Complete: All services required by the user should be defined
  - Consistent: should not have contradictory definition (also avoid ambiguity→ don’t leave room for different interpretations)

Non-Functional requirements

- Requirements that are not directly concerned with the specific functions delivered by the system
- Typically relate to the system as a whole rather than the individual system features
- Often could be deciding factor on the survival of the system (e.g. reliability, cost, response time)

Non-Functional requirements classifications:



Domain requirements

- Domain requirements are derived from the application domain of the system rather than from the specific needs of the system users.
- May be new functional requirements, constrain existing requirements or set out how particular computation must take place.
- Example: tolerance level of landing gear on an aircraft (different on dirt, asphalt, water), or what happens to fiber optics line in case of sever weather during winter Olympics (Only domain-area experts know)

Problems with natural language

- Lack of clarity
  - Precision is difficult without making the document difficult to read
- Requirements confusion
  - Functional and non-functional requirements tend to be mixed-up
- Requirements amalgamation
  - Several different requirements may be expressed together
- Ambiguity
  - The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult
- Over-flexibility
  - The same thing may be said in a number of different ways in the specification

Alternatives to NL specification

- Structured Natural language (via standard forms & templates)
- Program Description Language (PDL)
- Use-Cases (scenario-based technique)
- Mathematical specification (notations based on mathematical concepts such as finite-state machines or set.)

Structured language specifications

- A limited form of natural language may be used to express requirements
- This removes some of the problems resulting from ambiguity and flexibility and imposes a degree of uniformity on a specification
- Often best supported using a form-based approach

Form-based specification

ECLIPSE/Workstation/Tools/DE/FS/3.5.1

Function: Add node

Description: Adds a node to an existing design.

Inputs: Node type, Node Position

Outputs: Design identifier

Pre/Post conditions:

Other attributes:

Definition: ECLIPSE/Workstation/Tools/DE/RD/3.5.1

PDL-based requirements definition

- Requirements may be defined operationally using a language like a programming language but with more flexibility of expression
- Most appropriate in two situations
  - Where an operation is specified as a sequence of actions and the order is important
  - When hardware and software interfaces have to be specified
  - Example: ATM machine

PDL disadvantages

- PDL may not be sufficiently expressive to express the system functionality in an understandable way
- Notation is only understandable to people with programming language knowledge
- The requirement may be taken as a design specification rather than a model to help understand the system

ATM Specification: a PDL example

```
Class ATM {
    // declaration here
    public static void main (string args[]) InvalidCard {
    try {
    thisCard.read();  //may throw Invalid card exception
    pin = KeyPaD.READpIN(); attempts = 1;
    While (!thisCard.pin.equal(pin) & attempts < 4)
    pin = KeyPad.readPin(); attempts += 1;
    .
    .
    .
```

The requirements document

- The requirements document is the official statement of what is required of the system developers
- Should include both a definition and a specification of requirements
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

Requirements Engineering (RE) processes

- **Processes used to discover, analyse and validate system requirements**
- RE vary widely depending on the application domain, the people involved and the organization developing the requirements
- However, there are a number of generic activities common to all processes
  - Requirements elicitation
  - Requirements analysis
  - Requirements validation
  - Requirements management

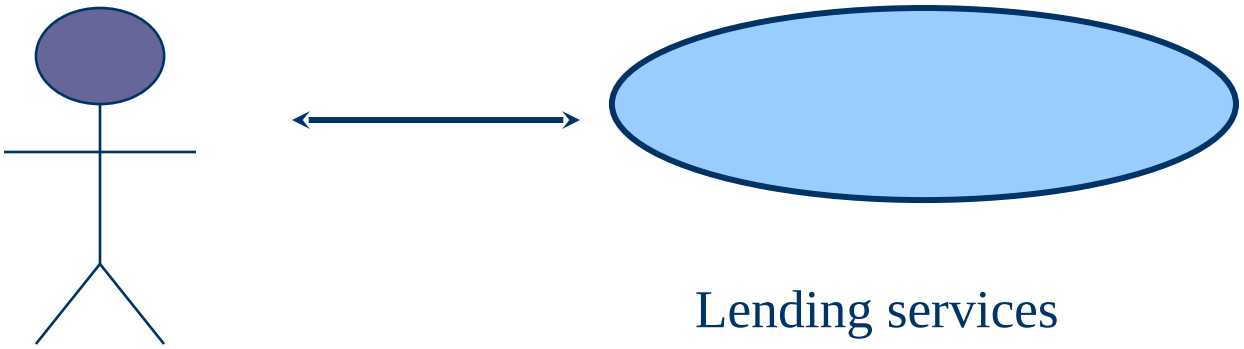
Problems of requirements analysis

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

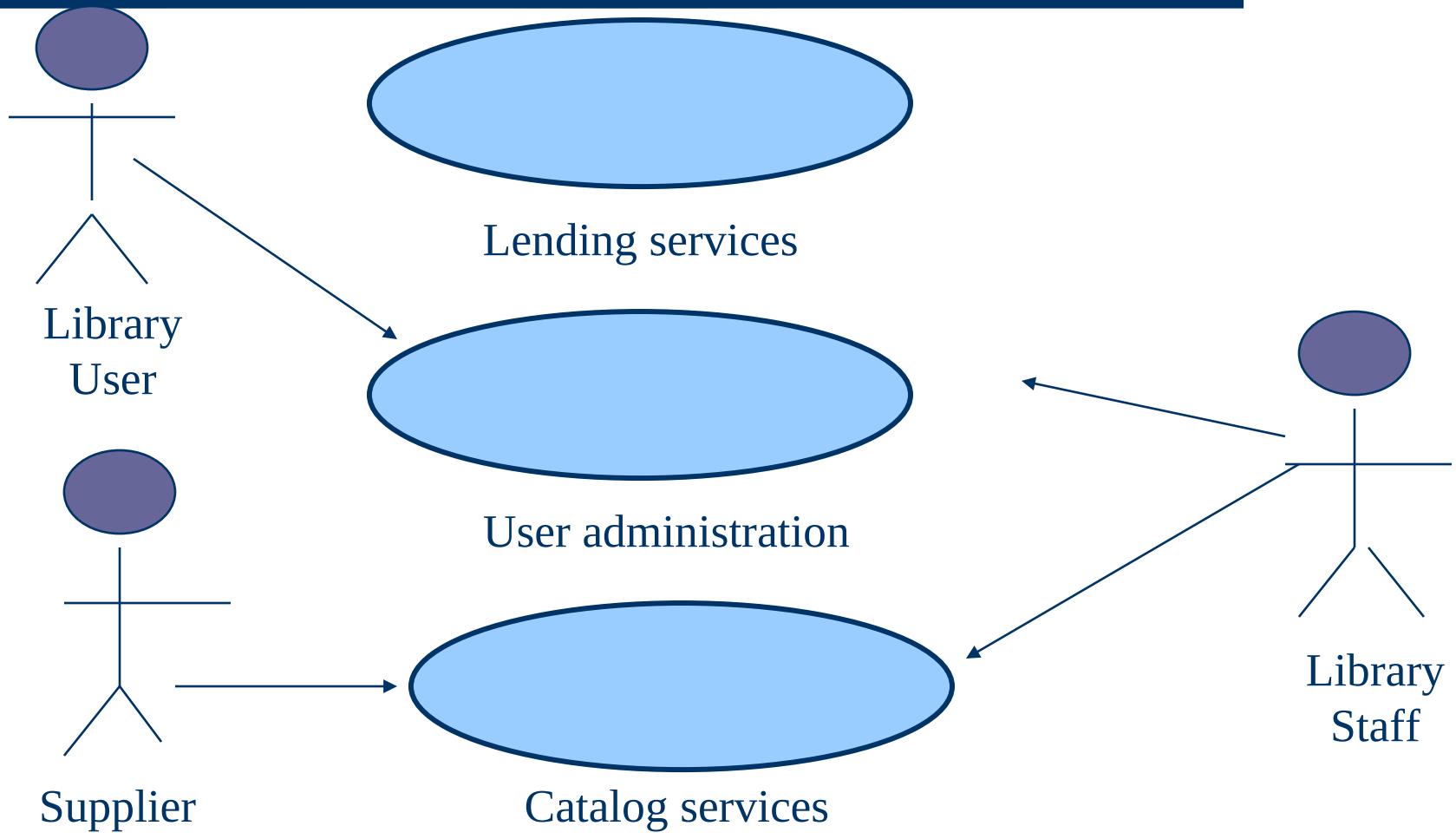
Use cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself
- A set of use cases should describe all possible interactions with the system
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system

Lending use-case



Library use-cases



Ethnography

- Ethnography is an observational technique that can be used to understand social and organizational requirements.
- Developed in a project studying the air traffic control process
- Problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant

Enduring and volatile requirements

- Enduring requirements. Stable requirements derived from the core activity of the customer organisation. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models
- Volatile requirements. Requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy

Classification of requirements

- Mutable requirements
  - Requirements that change due to the system's environment
- Emergent requirements
  - Requirements that emerge as understanding of the system develops
- Consequential requirements
  - Requirements that result from the introduction of the computer system
- Compatibility requirements
  - Requirements that depend on other systems or organizational processes

System Modelling

- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.
- Different models present the system from different perspectives
  - External perspective showing the system’s context or environment;
  - Behavioural perspective showing the behaviour of the system;
  - Structural perspective showing the system or data architecture.

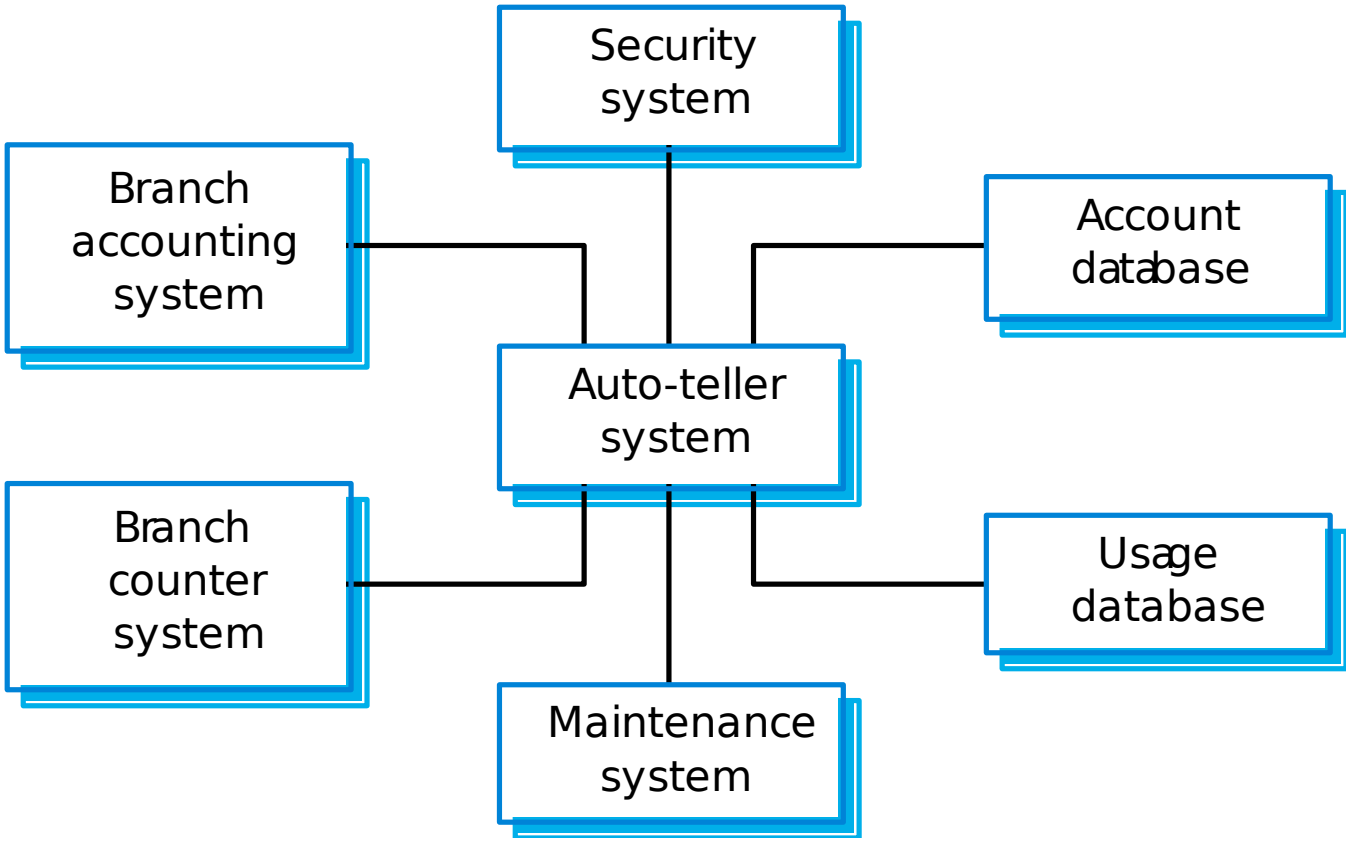
Model types

1. Context Model
   - External View
   - Illustrates Boundaries
   - Component Based
2. Behavioural Models
   - Data Flow Models
   - State Machine Models
3. Semantic Data Models
4. Object Models

Context models

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

The context of an ATM system



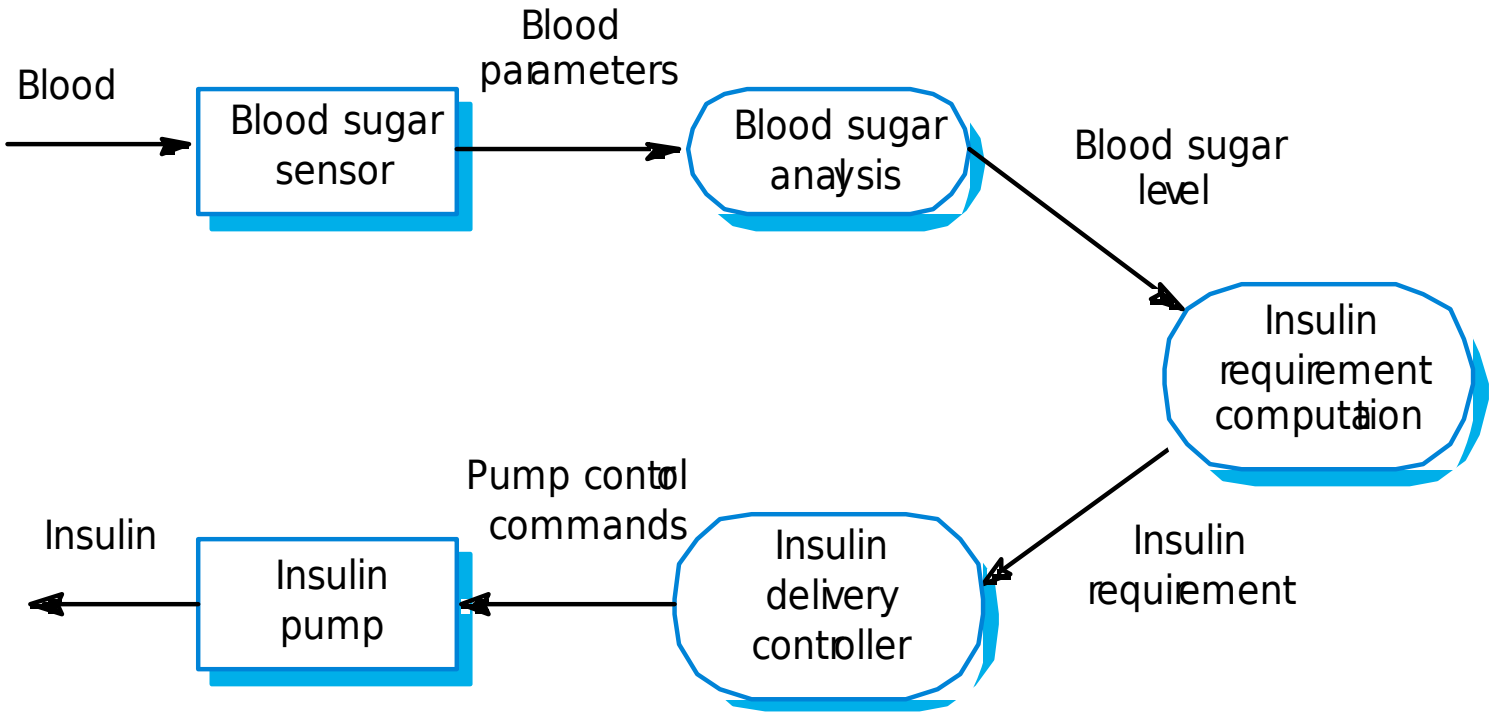
Behavioural models

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
  - Data processing models that show how data is processed as it moves through the system;
  - State machine models that show the systems response to events.
- These models show different perspectives so both of them are required to describe the system’s behaviour.

Data-processing models

- Data flow diagrams (DFDs) may be used to model the system’s data processing.
- These show the processing steps as data flows through a system.
- DFDs are an intrinsic part of many analysis methods.
- Simple and intuitive notation that customers can understand.
- Show end-to-end processing of data.

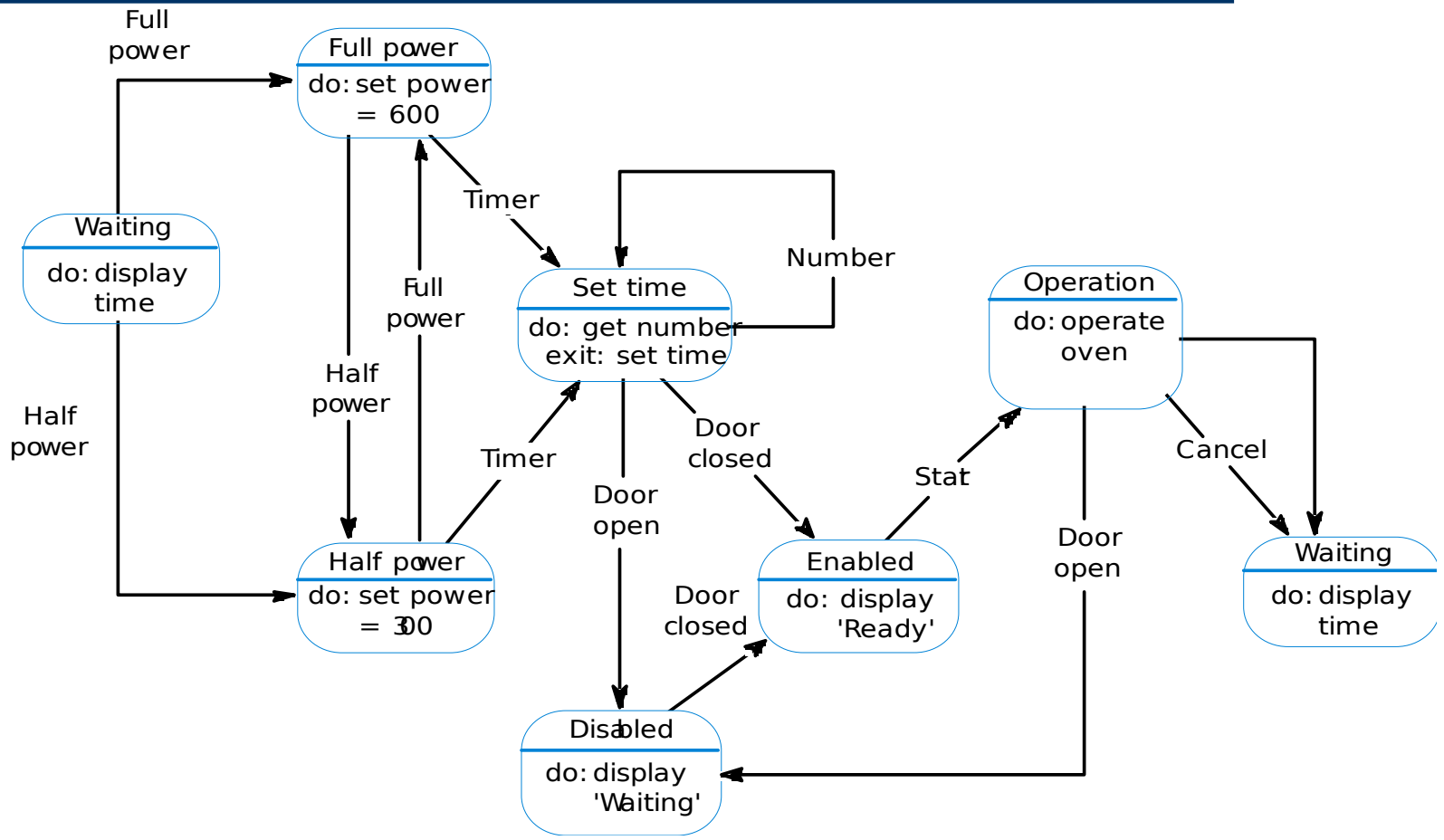
Insulin pump DFD



State machine models

- These model the behaviour of the system in response to external and internal events.
- They show the system’s responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

Microwave oven model



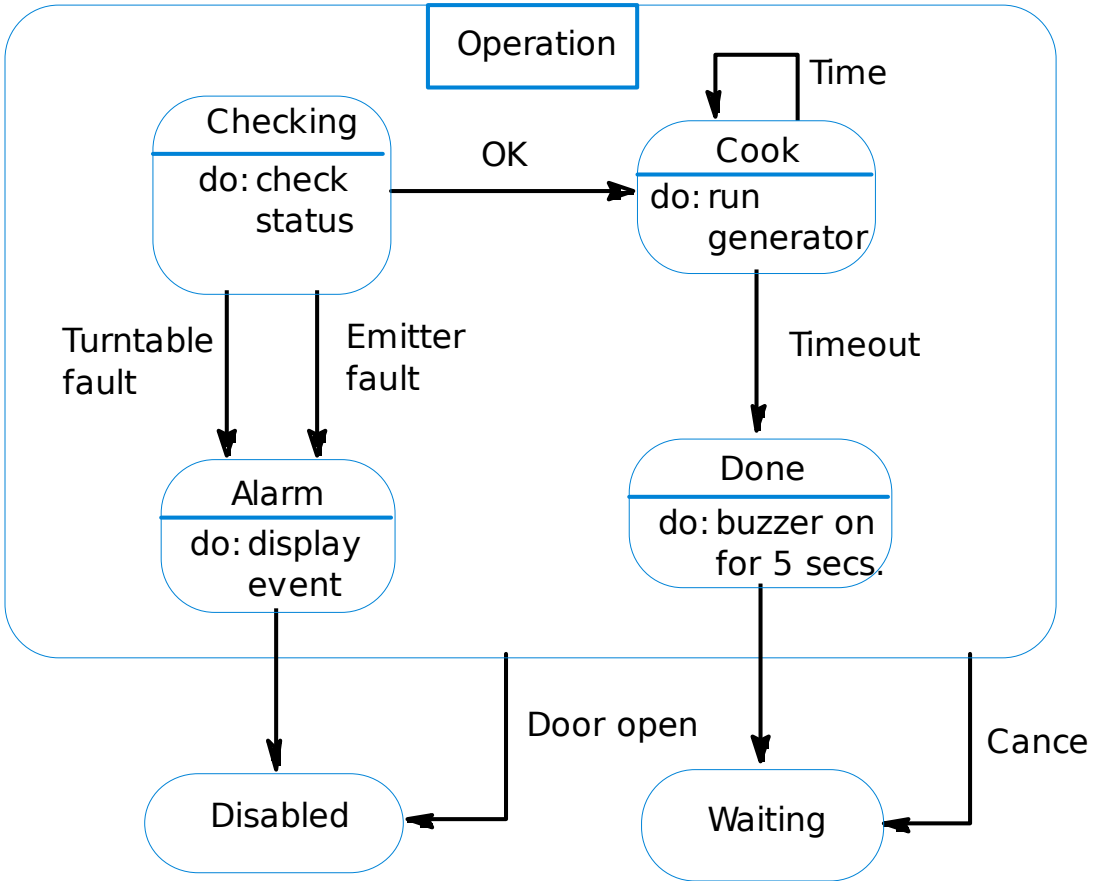
Microwave oven state description

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows ÔHalf powerÕ.
Full power	The oven power is set to 600 watts. The display shows ÔFull powerÕ.
Set time	The cooking time is set to the userÕs input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows ÔNot readyÕ.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows ÔReady to cookÕ.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows ÔCooking completeÕ while buzzer is sounding.

Microwave oven stimuli

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

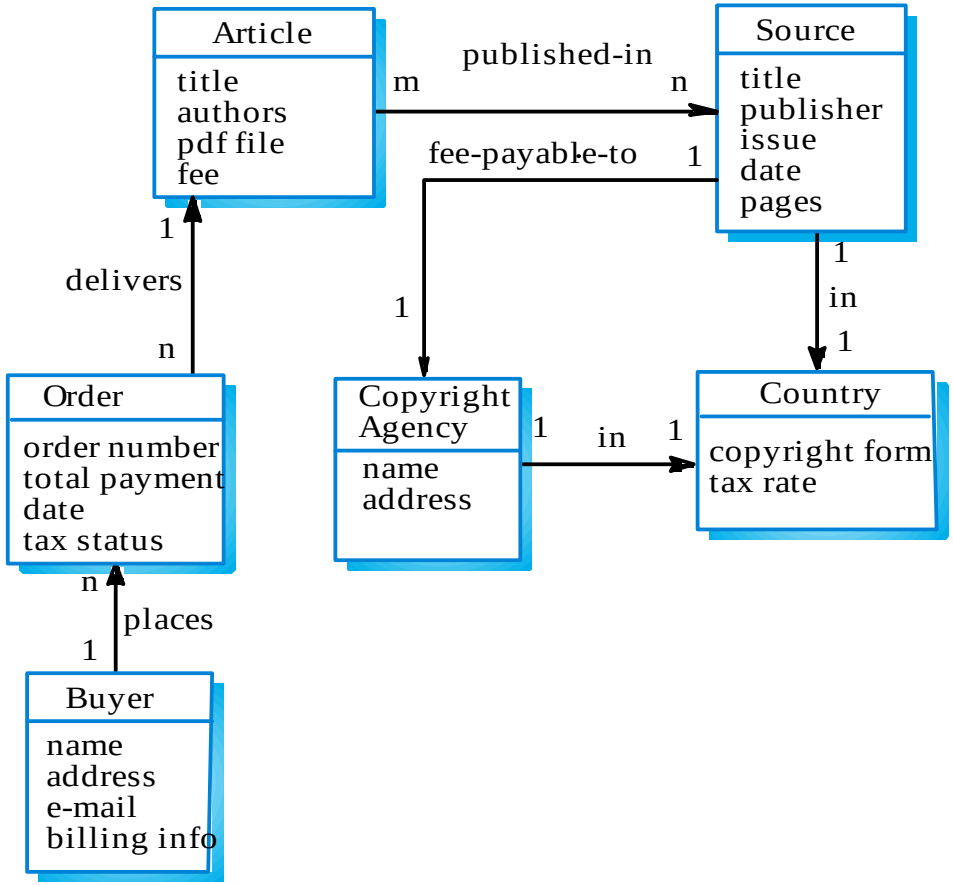
Microwave oven operation



Semantic data models

- Used to describe the logical structure of data processed by the system.
- An entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes
- Widely used in database design. Can readily be implemented using relational databases.
- No specific notation provided in the UML but objects and associations can be used.

Library semantic model



Data dictionaries

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included.
- Advantages
  - Support name management and avoid duplication;
  - Store of organisational knowledge linking analysis, design and implementation;
- Many CASE workbenches support data dictionaries.

Data dictionary entries

Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a co py of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

Object models

- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognized as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

Object models and the UML

- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.

UML Diagram Types

Static diagrams:

- Class
- Component
- Deployment
- Interaction overview
- Object
- Package

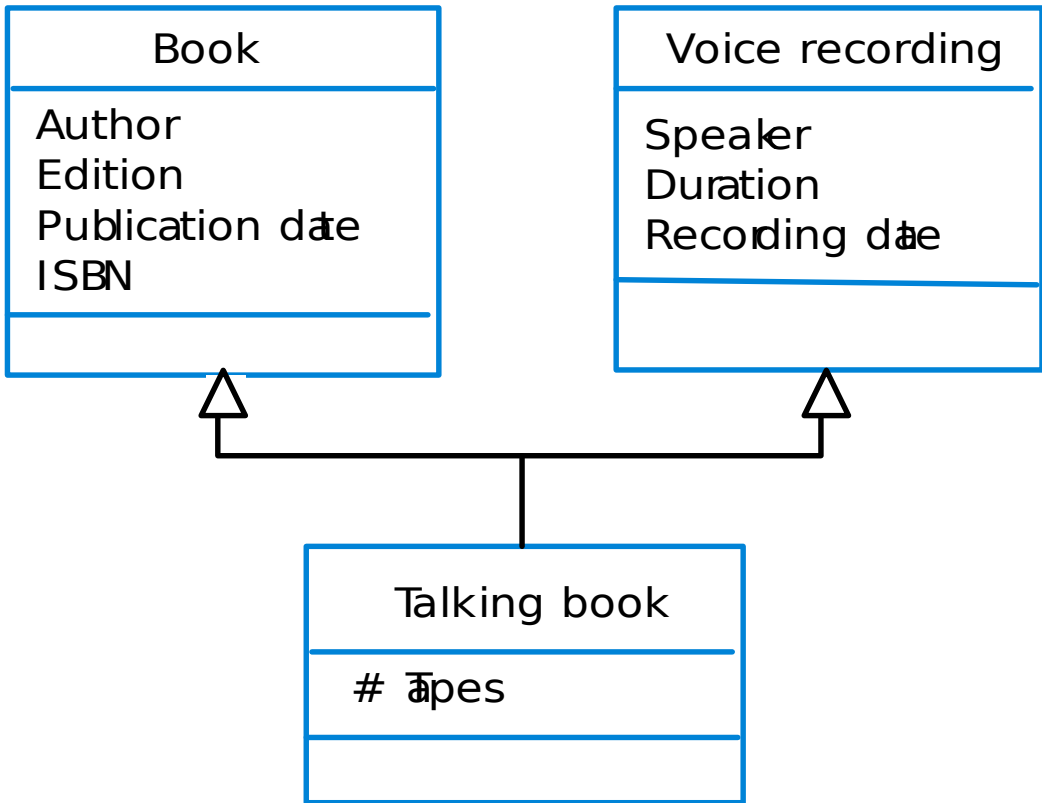
Dynamic diagrams:

- Activity
- Communication
- Composite structure
- Sequence
- State machine
- Timing
- Use case

Multiple inheritance

- Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes.
- This **can lead to semantic conflicts** where attributes/services with the same name in different super-classes have different semantics.
- Multiple inheritance makes class hierarchy reorganisation more complex.

Multiple inheritance



CASE workbenches

- A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.
- Analysis and design workbenches support system modelling during both requirements engineering and system design.
- These workbenches may support a specific design method or may provide support for a creating several different types of system model.

Software Testing

- Verification and validation planning
- Software inspections
- Software Inspection vs. Testing
- Automated static analysis
- Cleanroom software development
- System testing

Verification vs validation

- Verification:
  "Are we building the product right".
- The software should conform to its specification.
- Validation:
  "Are we building the right product".
- The software should do what the user really requires.

Static and dynamic verification

- Software inspections. Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis
- Software testing. Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed

Software inspections

- Software Inspection involves examining the source representation with the aim of discovering anomalies and defects without execution of a system.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.
- Management should not use inspections for staff appraisal i.e. finding out who makes mistakes.

Inspection procedure

- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair errors.
- Re-inspection may or may not be required.
- Checklist of common errors should be used to drive the inspection. Examples: Initialization, Constant naming, loop termination, array bounds…

Inspection roles

- Typically 6 people involved in any inspection:
  - 2 people: Code Author or Owner
  - 2 people: Code Inspectors
  - 1 Person: Reader & Scriber
  - 1 Person: Session Moderator

Inspection checks 1

Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a de limiter explicitly assigned? Is there any possibility of buffer overflow?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? If a break is required after each case in case statements, has it been included?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output? Can unexpected inputs cause corruption?

Inspection checks 2

Interface faults	Do all function and method calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Automated static analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

LINT static analysis

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
 int Anarray;
{   printf("%d",Anarray);  }

main ()
{
 int Anarray[5]; int i; char c;
 printarray (Anarray, i, c);
 printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Cleanroom software development

- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- This software development process is based on:
  - Incremental development;
  - Formal specification;
  - Static verification using correctness arguments;
  - Statistical testing to determine program reliability.

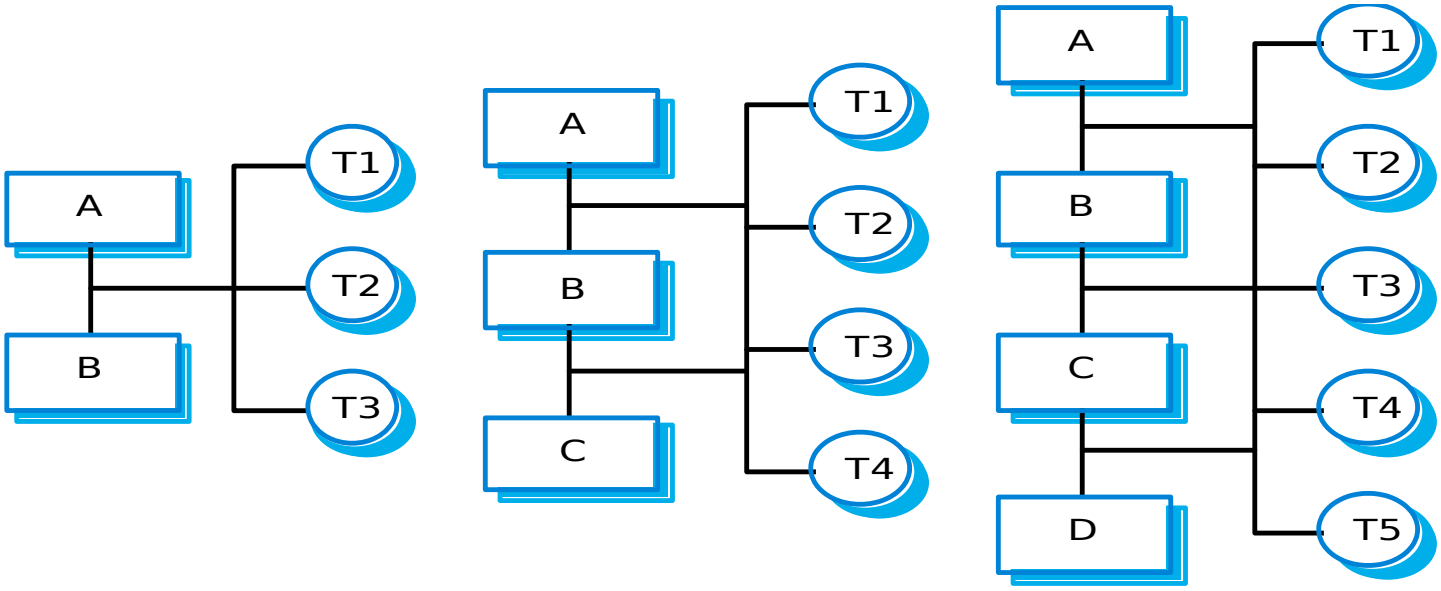
Cleanroom process teams

- Specification team. Responsible for developing and maintaining the system specification.
- Development team. Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
- Certification team. Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

System testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
  - Integration testing - the test team have access to the system source code. The system is tested as components are integrated.
  - Release testing - the test team test the complete system to be delivered as a black-box.

Incremental integration testing

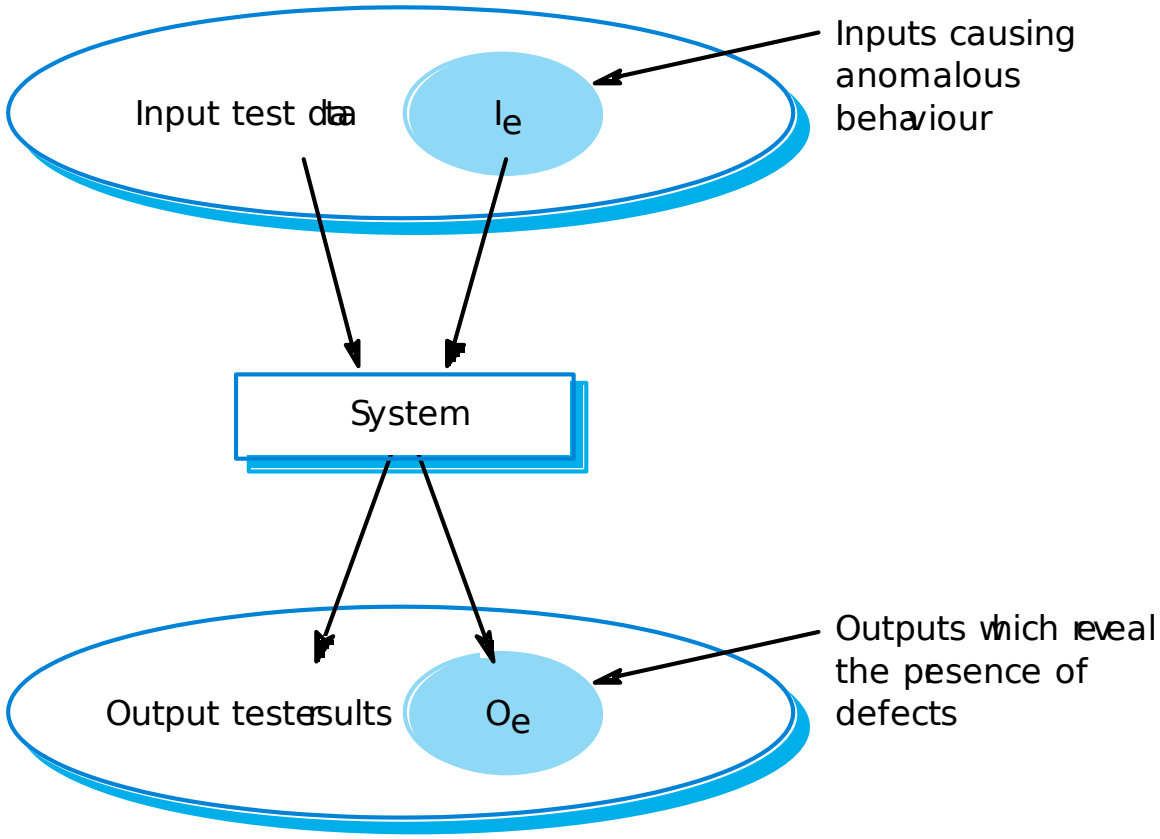


Test sequence 1 Test sequence 2 Test sequence 3

Release testing

- The process of testing a release of a system that will be distributed to customers.
- Primary goal is to increase the supplier's confidence that the system meets its requirements.
- Release testing is usually black-box or functional testing
  - Based on the system specification only;
  - Testers do not have knowledge of the system implementation.

Black-box testing



Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour.. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Component testing

- Component or unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Components may be:
  - Individual functions or methods within an object;
  - Object classes with several attributes and methods;
  - Composite components with defined interfaces used to access their functionality.

Test case design

- Involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Design approaches:
  - Requirements-based testing (i.e. trace test cases to the requirements)
  - Partition testing;
  - Structural testing.

Partition testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

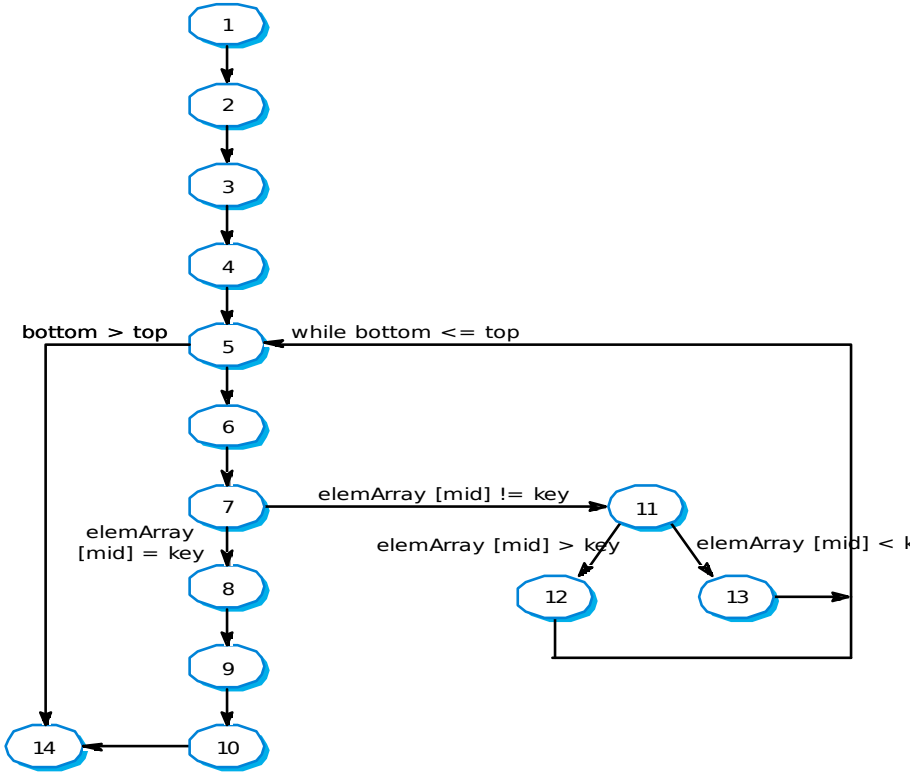
Structural testing

- Sometime called white-box testing.
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases.
- Objective is to exercise all program statements (not all path combinations).

Path testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.

Binary search flow graph



Independent paths

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Test automation

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as Junit support the automatic execution of tests.
- Most testing workbenches are open systems because testing needs are organisation-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.

Topics covered

- Architectural Design
- System Organization
- Subsystems and Modules
- Modular Decomposition Styles
- Control Models

Software architecture

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is **architectural design**.
- The output of this design process is a description of the **software architecture**.

Architectural design

- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

Architecture and system characteristics

- Performance
  - Localize critical operations and minimize communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localize safety-critical features in a small number of sub-systems.
- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

Architectural conflicts

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localising safety-related features usually means more communication so degraded performance.

System organization

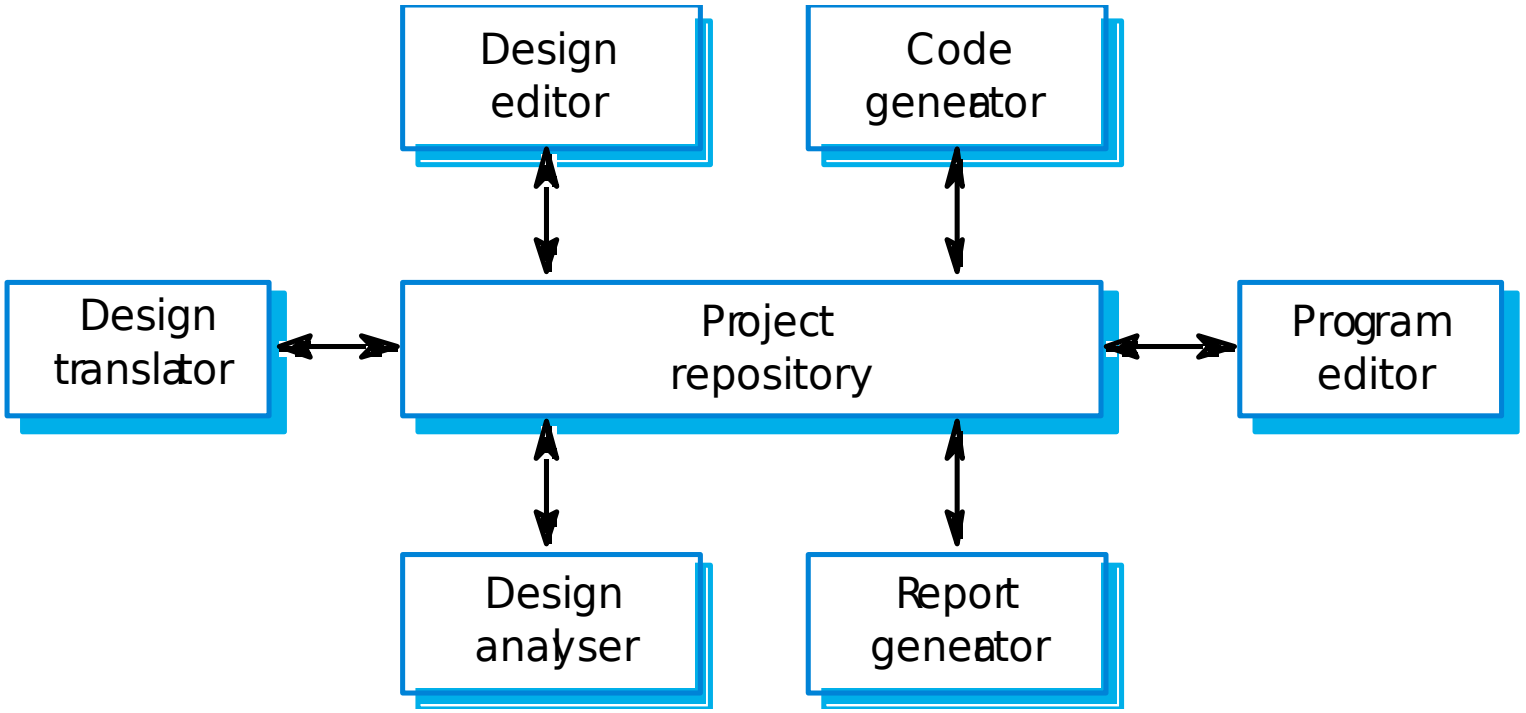
- Reflects the basic strategy that is used to structure a system.
- Two organizational models are widely used:
  - Data repository Model
  - Client-Server Model

The repository model

- Sub-systems must exchange data. Shared data is held in a central database or repository and may be accessed by all sub-systems;

- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

CASE toolset architecture



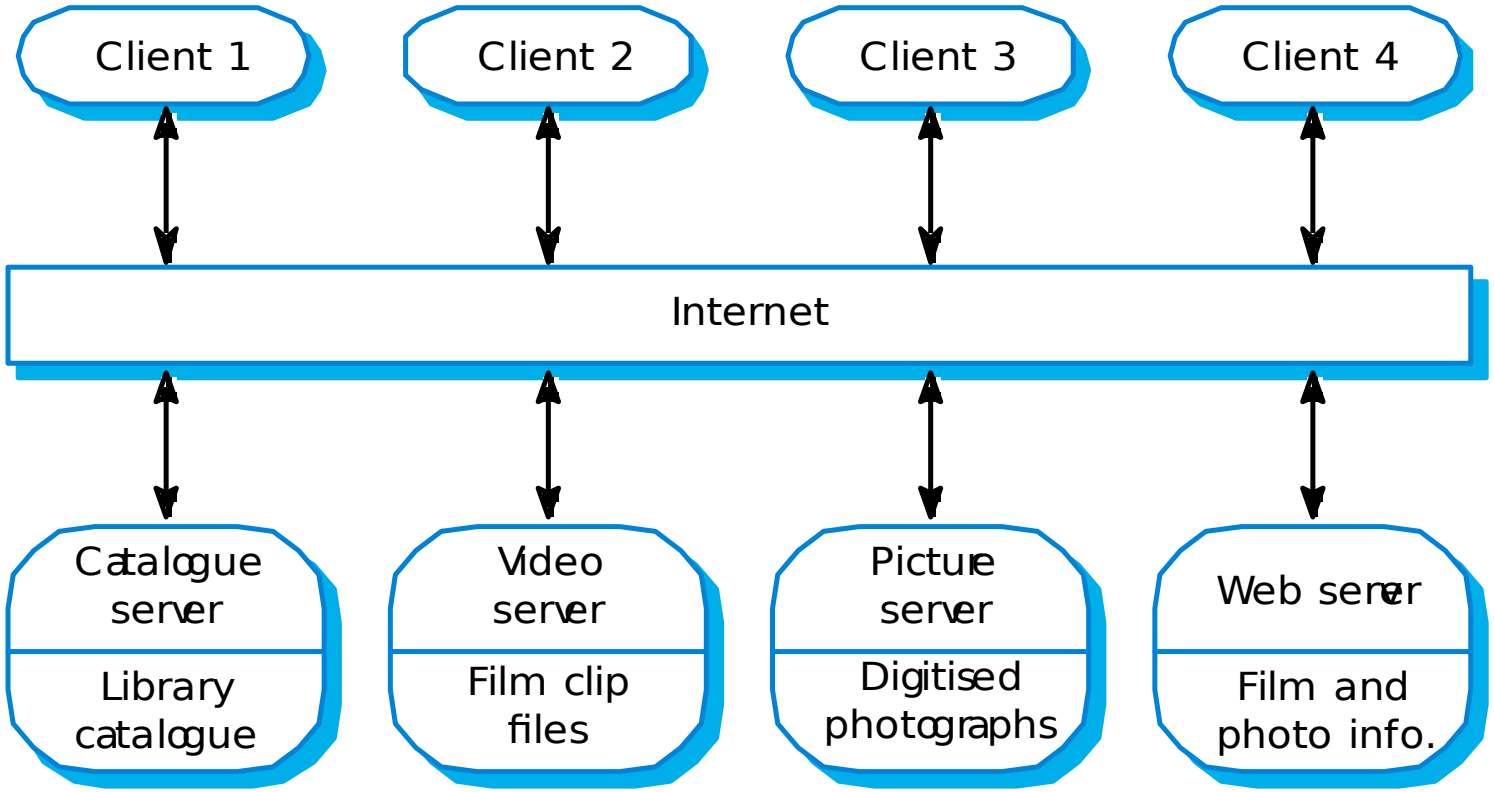
Repository model characteristics

- Advantages
  - Efficient way to share large amounts of data;
  - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
  - Sharing model is published as the repository schema.
- Disadvantages
  - Sub-systems must agree on a repository data model. Inevitably a compromise;
  - Data evolution is difficult and expensive;
  - No scope for specific management policies;
  - Difficult to distribute efficiently.

Client-server model

- Distributed system model which shows how data and processing is distributed across a range of components.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

Film and picture library



Client-server characteristics

- Advantages
  - Distribution of data is straightforward;
  - Makes effective use of networked systems. May require cheaper hardware;
  - Easy to add new servers or upgrade existing servers.
- Disadvantages
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
  - Redundant management in each server;
  - No central register of names and services - it may be hard to find out what servers and services are available.

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.

Modular decomposition

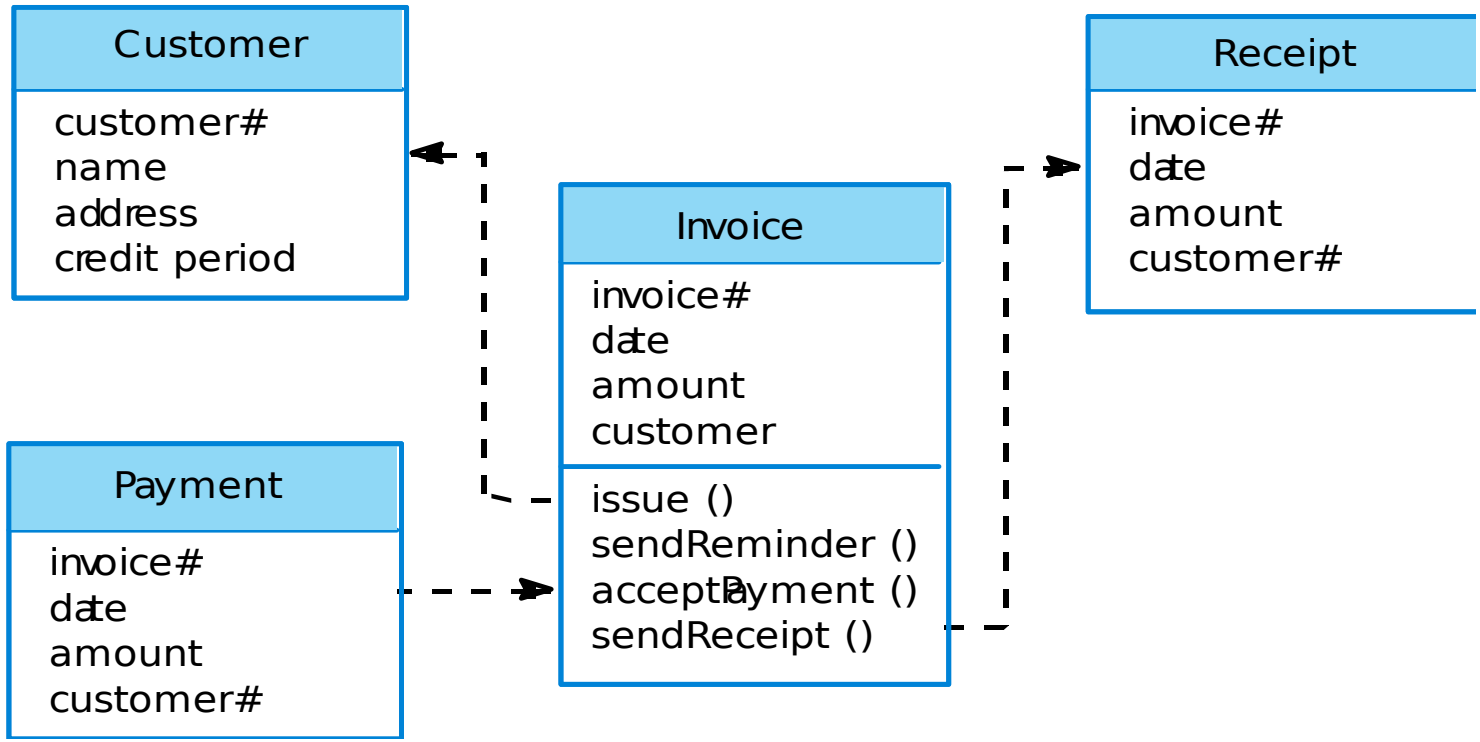
The structural level where sub-systems are decomposed into modules. Two modular decomposition models are:

- An object model where the system is decomposed into interacting object;
- A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.

Object models

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.

Invoice processing system: Object Model Example



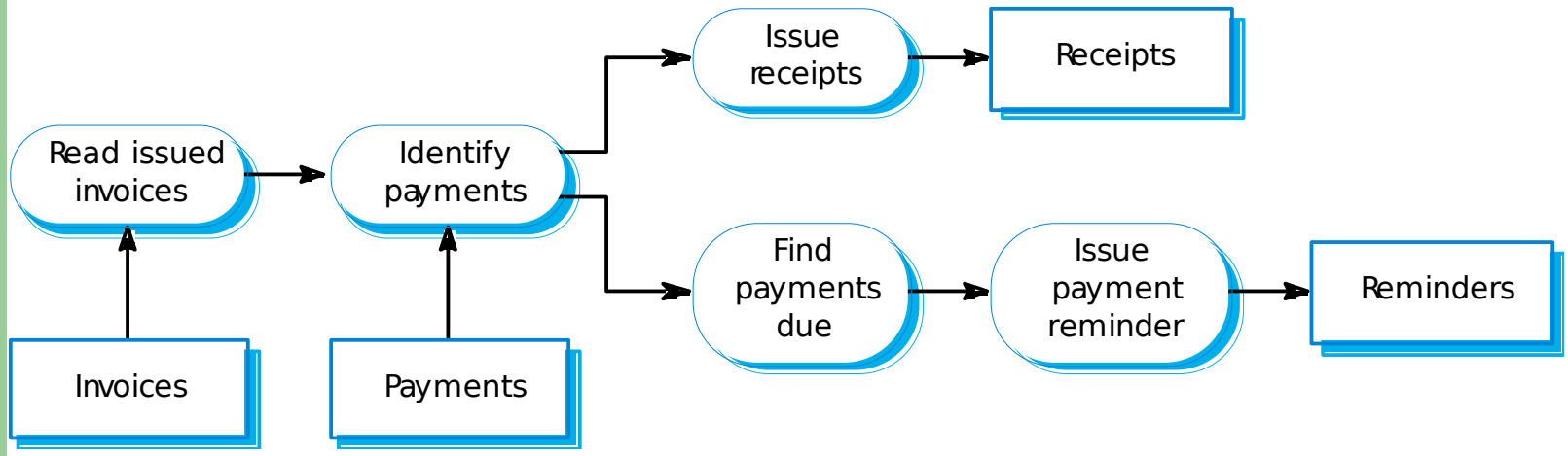
Object model advantages

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-oriented pipelining

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

Invoice processing system: Pipeline Example



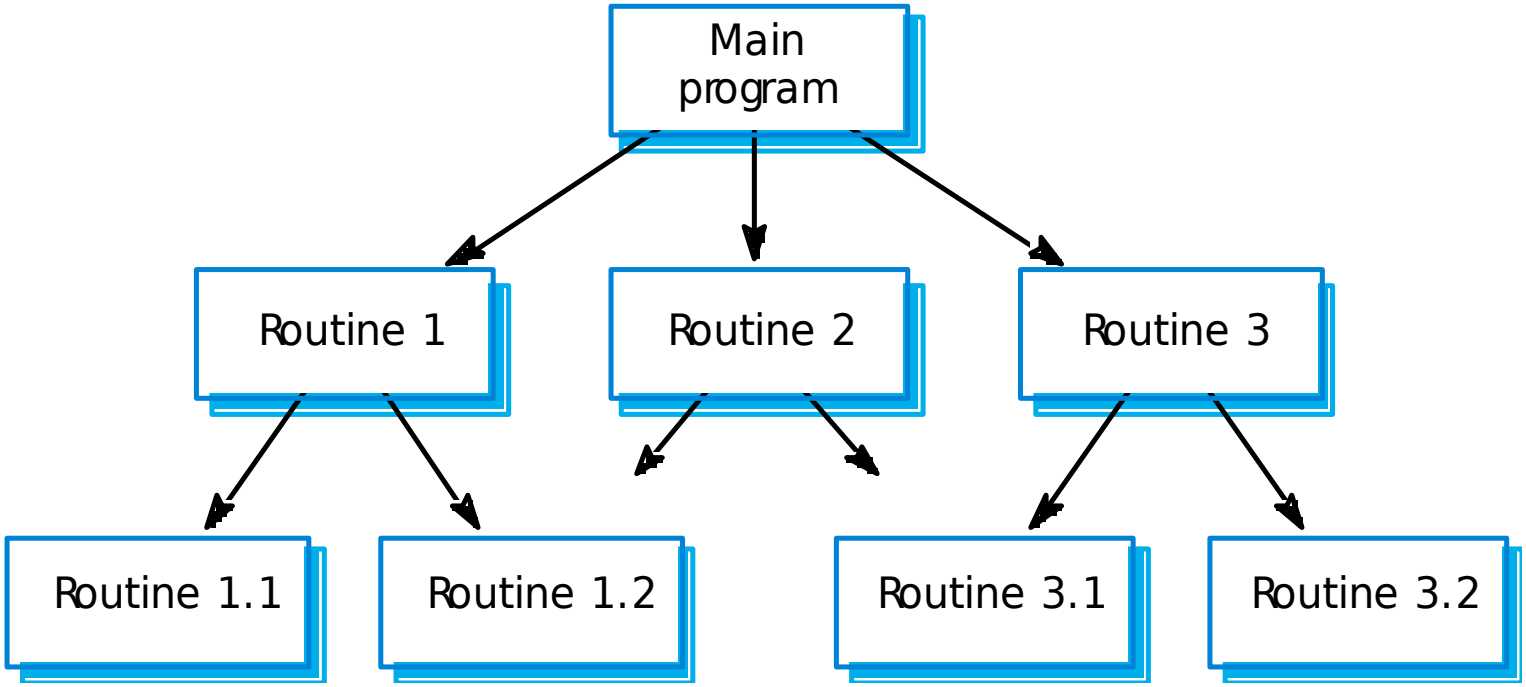
Control styles

Are concerned with the control flow between sub-systems. There are many different system decomposition model Including:

- Call-return model
- Manager model
- Event-Driven Systems
  - Broadcast model
  - Interrupt-Driven Model

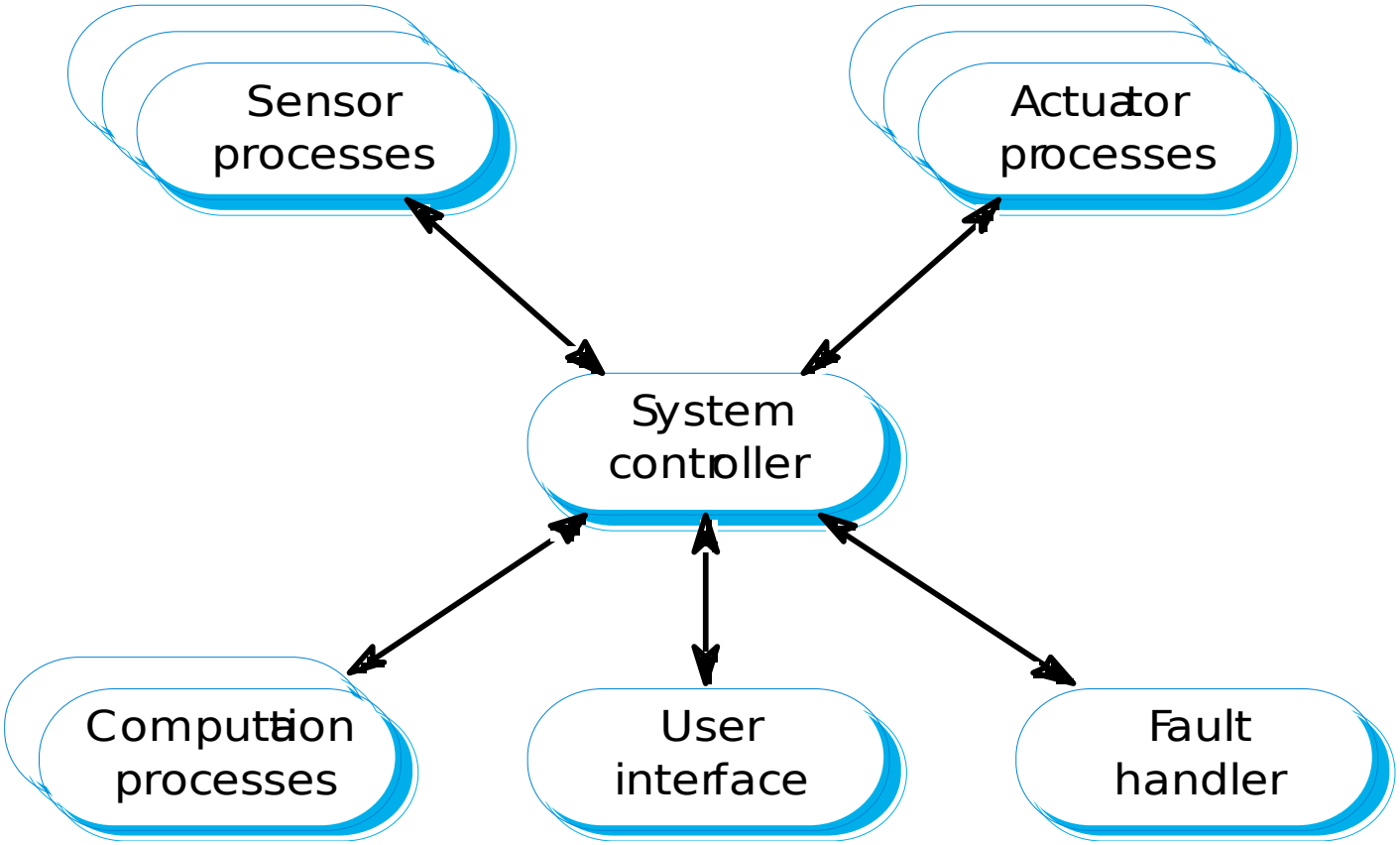
Call-return model

Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards.



Manager Model

One system component controls the stopping, starting and coordination of other system processes.



Event-driven systems

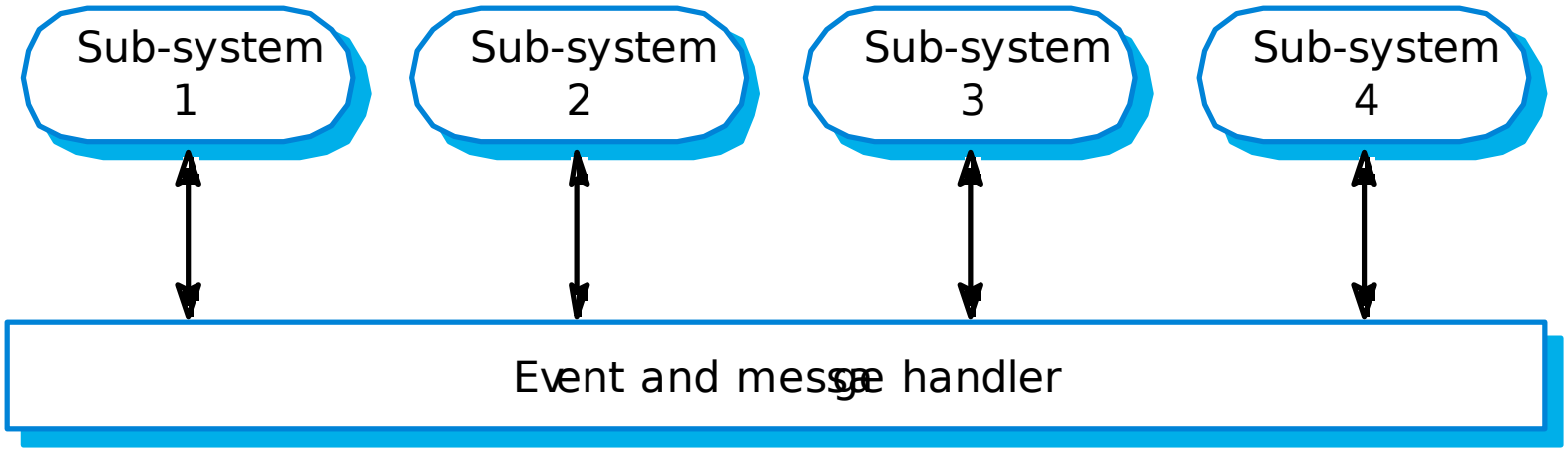
Driven by externally generated events where the timing of event outwit the control of sub-systems which process the event.

- Two principal event-driven models
  - Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so;
  - Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.

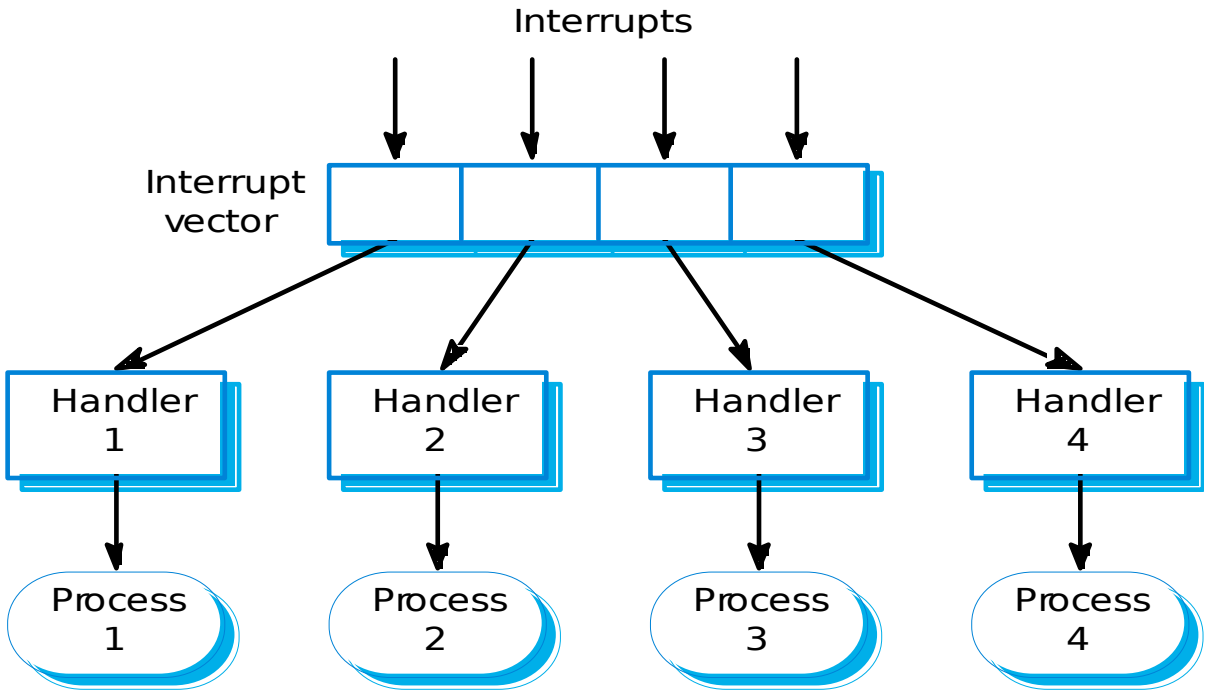
Broadcast model

- Effective in integrating sub-systems on different computers in a network.
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event.
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them.
- However, sub-systems don’t know if or when an event will be handled.

Broadcast model – Cont'd



Interrupt-driven control



Rational®
the software development company

Introduction to the Unified Modeling Language

Terry Quatrani, UML Evangelist

If you're a complete UML beginner, then consider this as UML 101, a basic introduction to the notational elements of the UML.

This article was first published on the Rational Developer Network after presentation at the RUC 2001.

Before joining Rational, in 1993, I worked for another well-known technology company, only there I was using OMT (the methodology developed by Jim Rumbaugh and company). Then I became a Technical Representative at Rational, and went from using OMT and rectangles, to using the Booch methodology and clouds. Back then, Rational only had 300 employees, and the UML was just a glimmer in the eyes of those who would go on to be known as "the Three Amigos." So I guess I date myself a bit by saying that I was doing OO development before UML came along, but that admission does give me a certain amount of credibility to talk to you about UML, don't you think?

The Importance of Modeling

When my son was 13, my husband and I thought he should have some space of his own. So we called in Leon the handyman and sat around the kitchen table. "What do you want?" he asked. Simple…a few closets, some electrical outlets, a cable hookup. Leon went off and remodeled the basement, and life was good. That is until winter came along and poor Michael comes upstairs and says, "Mom, I'm cold." You see, we hadn't thought about putting heat in the basement. So we decided to put in a gas fireplace. However, we don't have natural gas where I live, so we had to get a propane tank. Unfortunately, the only place it would fit was over in one corner, with all the storage in the other corner, and everything else in a third corner. The result is that you can't sit in my basement and see the TV and the fireplace at the same time. If we'd done some planning, we would have been able to design a room that was much more functional and comfortable.

Now in this instance we made some mistakes, but it wasn't a serious disaster. Think about building a skyscraper, though. No one would dream of a major construction project without thorough blueprints. That's *blueprints* plural, because it's important to not only have one plan of the skyscraper – you need to have multiple plans. The electrician needs a view to show where the wiring goes. The plumber needs another plan so he doesn't put a sink in the elevator. And the carpenters need to know where to put this expensive crown molding in the CEO's office. Different workers need different views of what they're trying to build. And that's what we're doing with software.

That different view is what we mean when we talk about the logical view, the use case view, the component view, and the deployment view – all the different views of the application under construction. So whose view should you model? Well, everybody involved in the project lifecycle is going to do some sort of modeling; they just might be doing different things. The business modeler will model different requirements than the application modeler will, but they are all creating a view that is important to their world of the application. And the language they use to do this is the Unified Modeling Language.

What is UML?

The easiest answer to that question is a quote:

"The UML is the standard language for specifying, visualizing, constructing, and documenting all the artifacts of a software system."

The more complex answer requires a short history lesson, because the UML is really a synthesis of several notations by Grady Booch, Jim Rumbaugh, Ivar Jacobson and many others.

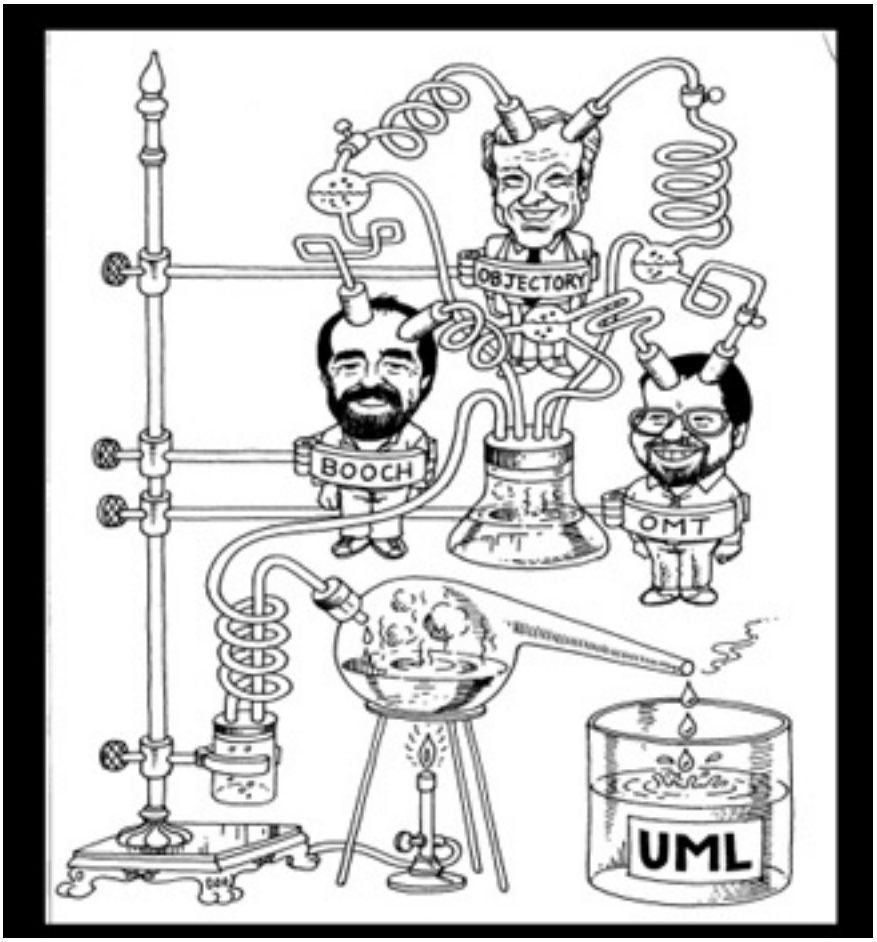


Figure 1: How it all began

Back in the late 80s, when I started modeling, there were many different methodologies. And each methodology had its own notations. The problem was that if different people were using different notations, somewhere along the line somebody had to do a translation. A lot of times, one symbol meant one thing in one notation, and something totally different in another notation. In 1991, everybody started coming out with books. Grady Booch came out with his first edition. Ivar Jacobson came out with his, and Jim Rumbaugh came out with his OMT methodology. Each book had its strengths as well as its weaknesses. OMT was really strong in analysis, but weaker in design. The Booch methodology was stronger in design and weaker in analysis. And Ivar Jacobson's Objectory was really good with user experience, which neither Booch nor OMT really took into consideration back then.

Then, in 1993, a funny thing happened. Grady came out with the second edition of his first book; it still had the good design, but some of the good analysis stuff from OMT had started creeping into his methodology. And actors and use cases from Ivar were in there as well. And Jim was writing a series of articles for the Journal of Object Oriented Programming that people referred to as OMT 2, which still had the good analysis work, plus some of Grady's good designs, and … you guessed it … all of a sudden actors and use cases were added into OMT 2. That was the beginning of the informal unification of methodology. And it came as something of a relief, because it really had been a method war. Imagine a group of engineers sitting around debating is it a rectangle? Is it a cloud? People doing development had a lot of fights over this type of thing.

Jim Rumbaugh joined Rational in 1994, and we locked him and Grady in a room together, told them to work together and come up with a methodology, and didn't let them out until October of 1995. That year, at OOPSLA, they introduced the Unified Method, which would become the UML. At that time, the Unified Method was both the language and the process that went along with it. [1] Once Ivar joined Rational, we threw him into the room, gave them another couple of years to collaborate (during which they decided to separate the language from the process), and in 1997 submitted the Unified Modeling Language to the Object Management Group (OMG) for standardization. Now, contrary to popular belief, Rational does not own the UML, although we continue to work on it. The UML belongs to the OMG. If you go to the OMG Web site, you can download the PDF version, which I keep on my laptop because it's a great reference. And speaking of reference, I think it's time we started exploring the UML, don't you?

Activity Diagrams

The logical place to start walking through some of the UML diagrams is by looking at activity diagrams.

[1] Ever heard the term "The Three Amigos?" This was the night that name was given to Grady, Jim, and Ivar. On the same night the Unified Method debuted, it was announced that Rational had acquired Objectory and that Ivar Jacobson was now part of Rational. Grady walked into that meeting with a t-shirt reading "The Three Amigos," gave one to Jim and Ivar and the rest is history.

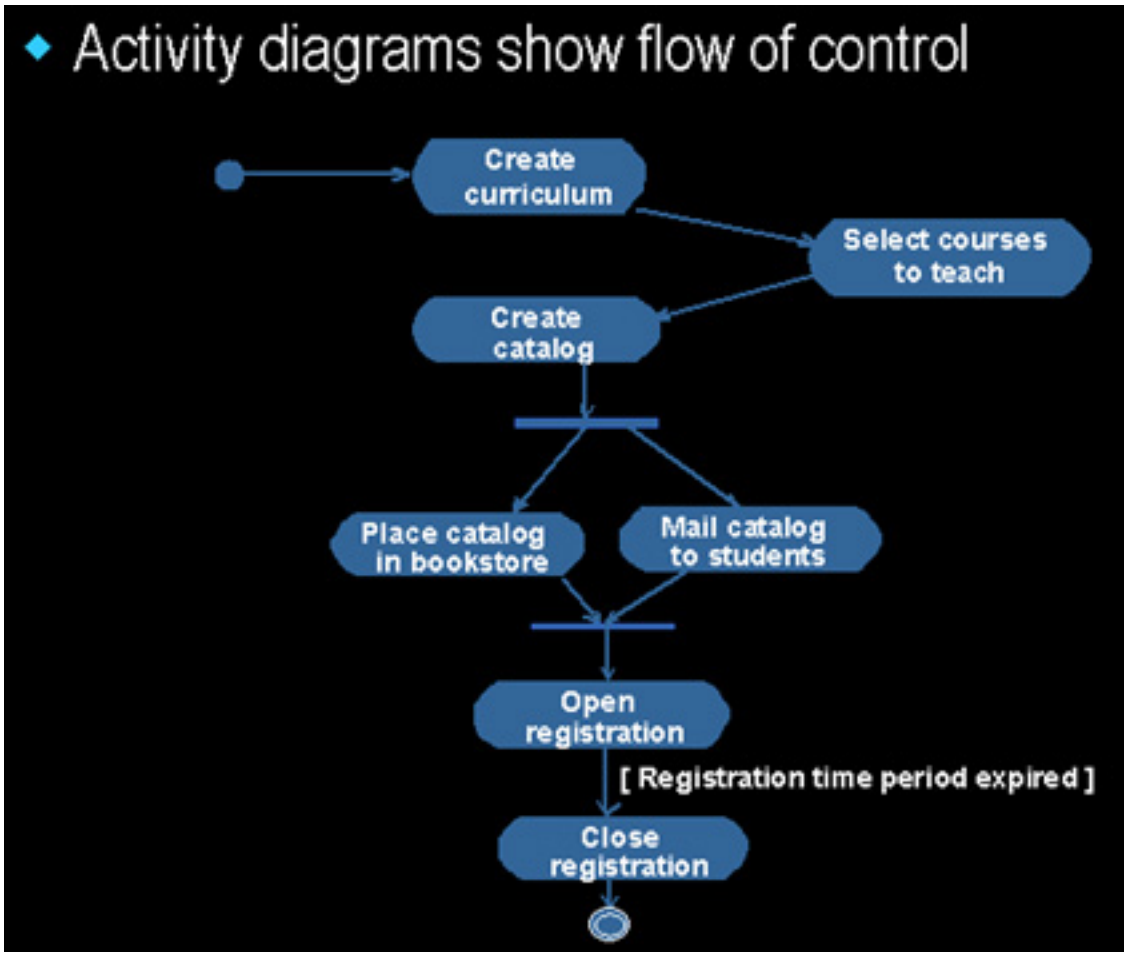


Figure 2: Activity diagram

Activity diagrams show the flow of control. As illustrated in Figure 2, you can see activities represented as rounded rectangles. Activities are typically action states – states that transition automatically to the next state after the action is complete. The filled in circle represents the start of the activity diagram – where the flow of control starts. Transitions shown as arrows show how you move from activity to activity. Synchronization bars show how activities happen in parallel. I can guard a transition that says "I want you to go to this activity only if this condition is true," and I can show you where it stops. Now if you're a certain age, you'll probably look at this activity diagram and think, "hmm…that looks like a flow chart." And that's exactly what it is, except I'm not doing it down at the programming level. Typically, I use an activity diagram fairly early on in my analysis and design process to show business workflow. I'll also use them to show where each of my use cases might be in an activity to illustrate what use case has to happen. I also use activity diagrams to show how things flow between my use cases.

But one of the great things about the UML is its versatility.[2] So while I use activity diagrams at the beginning of the lifecycle, others can use them in a different phase entirely. I've seen people use activity diagrams down at the design level where they had a very complicated set of

[2] Someone once said to me about the UML is that "it's not a revolution, it's an evolution." And it's true, it's taken all the 20 plus years of software development experience and all the good practices we've learned to be combined into this notation. And, like any evolution, along the way you learn what works best in a variety of situations.

algorithms for a particular class. And many people use them to show the flow between the methods of a class.

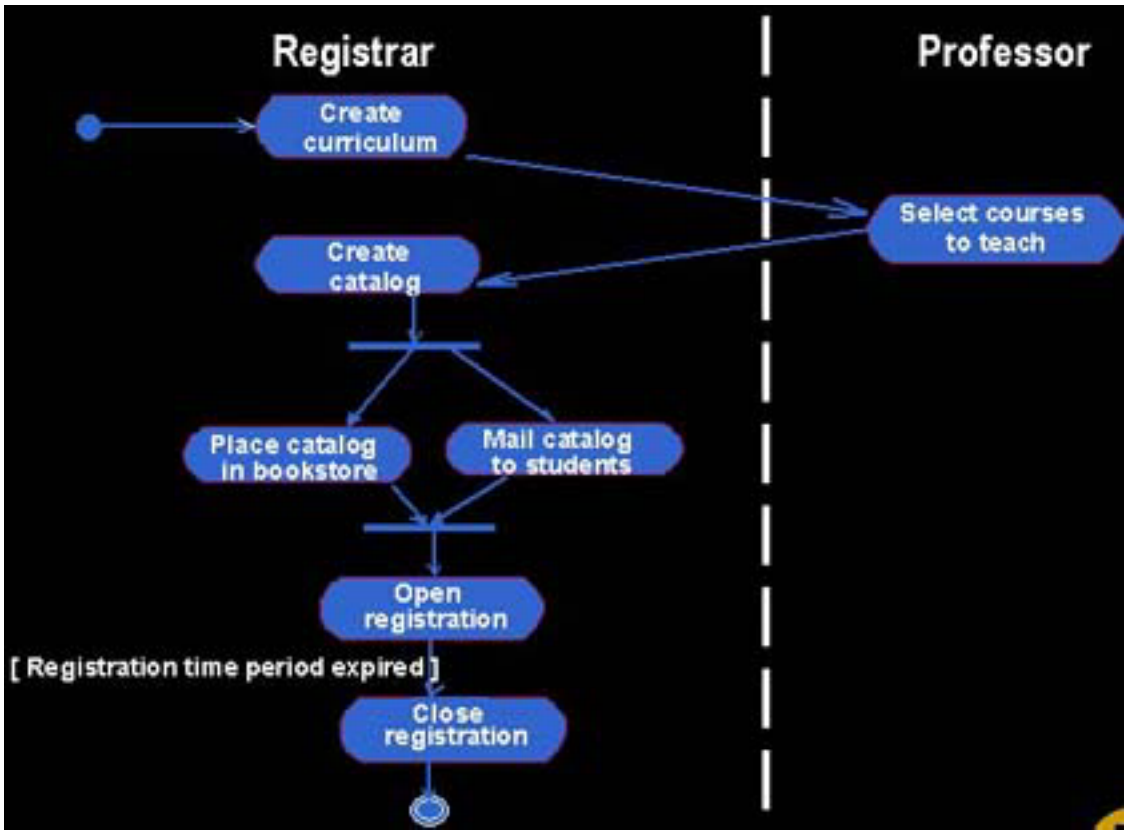


Figure 3: Swimlanes

One of the things I can show with activity diagrams is swimlanes. One of the best ways to use swimlanes and activity diagrams is to show ownership. It clearly illustrates to all parties what individual or group is responsible for what activity.

Use Case Diagrams

The next diagram to look at is the use case diagram. These I create by first looking at my actors. An actor is someone or something that is external to the system, but that is going to interact with the system. Actors are represented as stick figures.

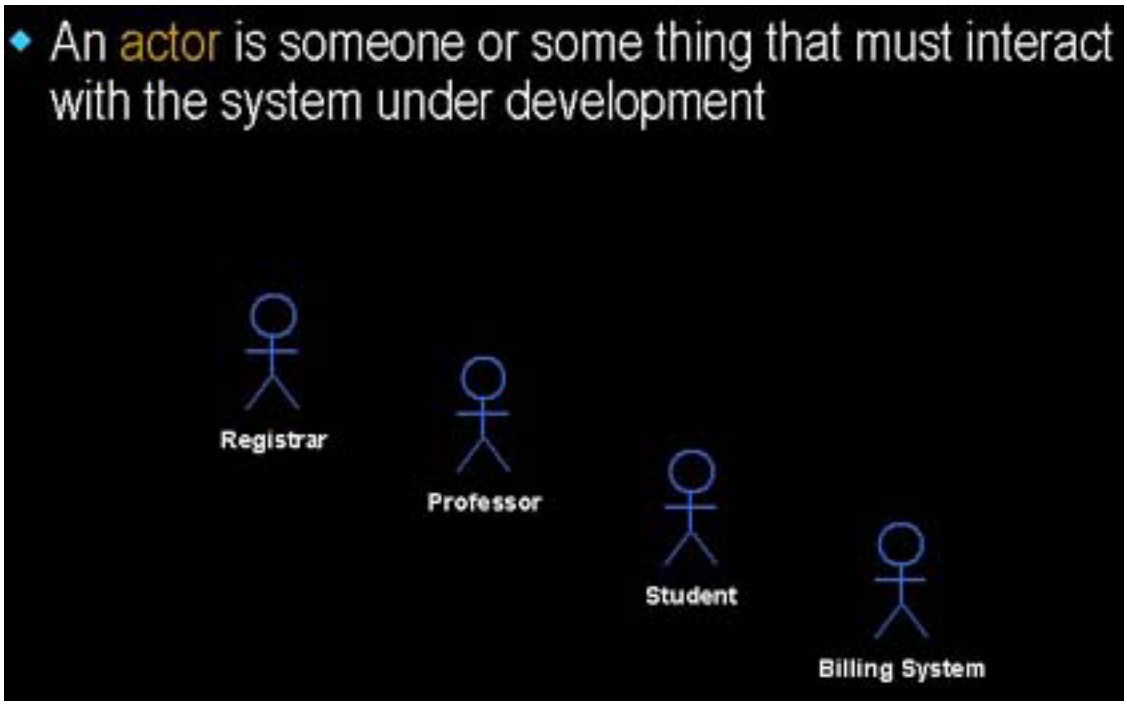


Figure 4: Actors

The example that I worked up for this introduction to UML is a little model of a course registration system. So in this instance, the first thing I would do when starting my analysis process is to ask, "who is going to interact with this system?"

For the course registration model, I have a registrar, a professor, and a student. I also have an external billing system. This billing system also qualifies as an actor. (See, an actor doesn't have to be a person – it's anything that interacts with the system but is outside of the system.)

A use case is a sequence of related transactions performed by an actor in the system in a dialog. Or, to put it in English, a use case is a chunk of functionality. And here's the key: it is *not* a software module – it is something that provides value to the actor.

Use cases are shown as ovals, and the easiest way to find them is to look at each of your actors and ask yourself why do they want to use the system. In my case, my registrar is going to maintain the curriculum, my professor is going to request a roster, my student maintains the schedule, and my billing system receives the billing information. So I create my use cases by looking at it from the customer point of view and asking, "so, mister system actor, why do you want to use the system? What value does this system provide to you?"

The next step, once you've identified how your actors will be interacting with the system, is do document your use cases.
Each use case needs to be documented with the flow of events, and this is done from the actor's point of view. It should detail what the system must provide to the actor when the use case is executed. Typically it will show things like how the use case starts and finishes. What things does that use case have to do? You'll have the normal flow of events (what I call the "happy days" scenario), where everything works. Then you'll get the abnormal flow of events, the "rainy day" scenario. What happens when the system doesn't work? I've found by documenting my flow of events that I always start with the happy days scenario.

Take as an example, walking up to an automated teller machine. You walk up to the ATM and insert your card. It asks for your PIN number. You enter it, and you are asked what you would like to do. You say “I want some money.” It asks where the money should be taken from. You tell it to take it from your checking account. It asks how much. You say $100.00. Then magic happens….it gives you $100.00. Then it asks if you want another transaction. You say no. It gives you your card back, gives you a receipt, and the transaction is over. That’s the happy day scenario.

Second scenario. You go up to the ATM, insert your card, and enter your PIN. The ATM tells you it’s the wrong PIN. You enter your number again. Again you are told that the PIN is incorrect. You repeat once more, with the same results. What happens? The ATM eats your card.

That’s what I mean by an alternate scenario. One of the most important things to capture is these high level alternate scenarios and get your customer to agree with them. You’re not going to capture every “what if.” If you try to document every possibility you’ll be documenting forever. But there are certain things you want to capture such as, in this instance, if you enter the wrong PIN three times in a row, the bank keeps your card. And by documenting this early in the system, you can get agreement with the customer (in this case the bank who runs the ATM) that you are handling risks appropriately

In the curriculum flow of events, the use case is as follows:

It begins when the Registrar logs onto the Registration System and enters his/her password. The system verifies that the password is valid (E-1) and prompts the Registrar to select the current semester or a future semester (E-2). The Registrar enters the desired semester. The system prompts the professor to select the desired activity: add, delete, review, or quit.

- If the activity selected is “add”, the S-1: Add a Course subflow is performed.
- If the activity selected is “delete”, the S-2: Delete a Course subflow is performed.
- If the activity selected is “review”, the S-3: Review Curriculum subflow is performed.
- If the activity selected is “quit”, the use case ends.

In my course registration example, for instance, you can see that there are a lot of “if X then Y” workflows. That’s where you want your customer to help you out. Getting agreement early means your customer understands these scenarios and says “yes, this is what we want.” Use cases are a great way to ensure that what you’re building is really what the customer wants, because they show the actors, the use cases, and the relationships between them.

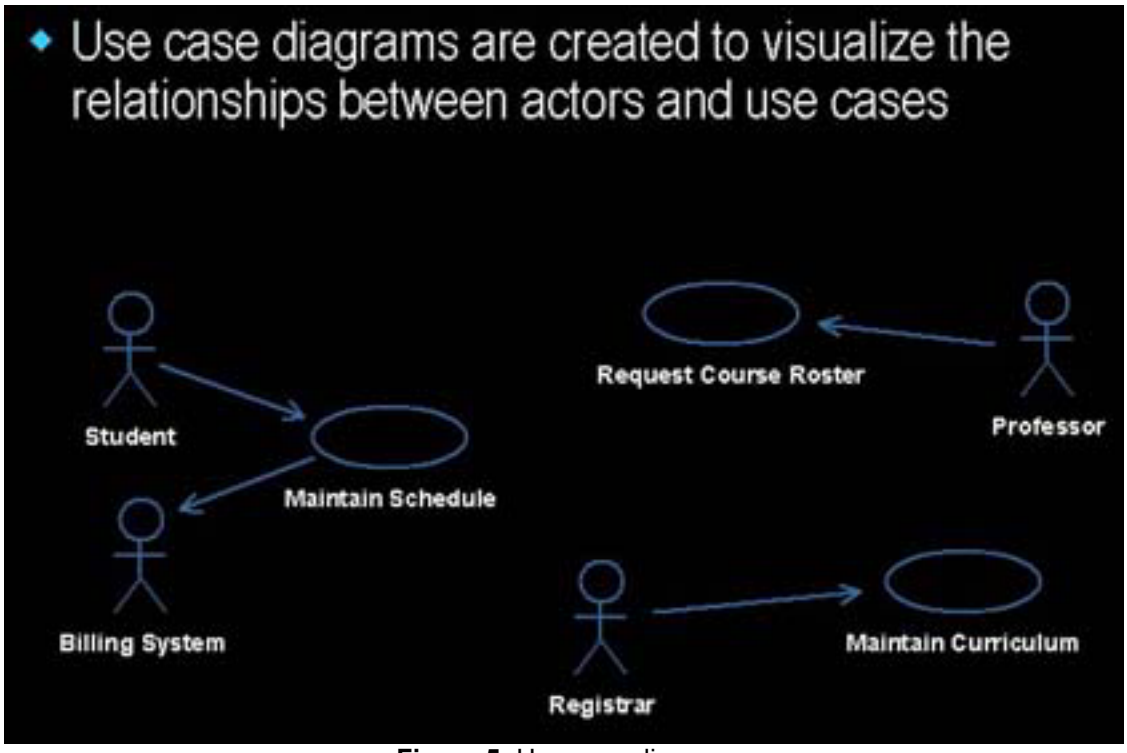


Figure 5: Use case diagram

So we have a great diagram that graphically shows me what? The answer is simple – it is a great diagram that shows a good overview of the system. It shows what is outside the system (actors) and the functionality that the system must provide (use cases). If there is a legacy system you need to take into consideration, here's where you deal with it. Forcing me to work with these types of interfaces very early in the project means that I won't be faced with the prospect of waiting until coding starts to figure out how I'm going to talk to that black box that I can't change.

One more thing you should know about use cases is the use case realization. This is the "how" of the use case. It's usually a bucket that contains three different types of diagrams: sequence diagrams, collaboration diagrams, and a class diagram that we call a view of participating classes. Use case realizations are basically a way of grouping together a number of artifacts relating to the design of a use case.

Sequence Diagrams

Sequence diagrams show object interactions arranged in a time sequence. I can use the flow of events to determine what objects and interactions I will need to accomplish the functionality specified by the flow of events.

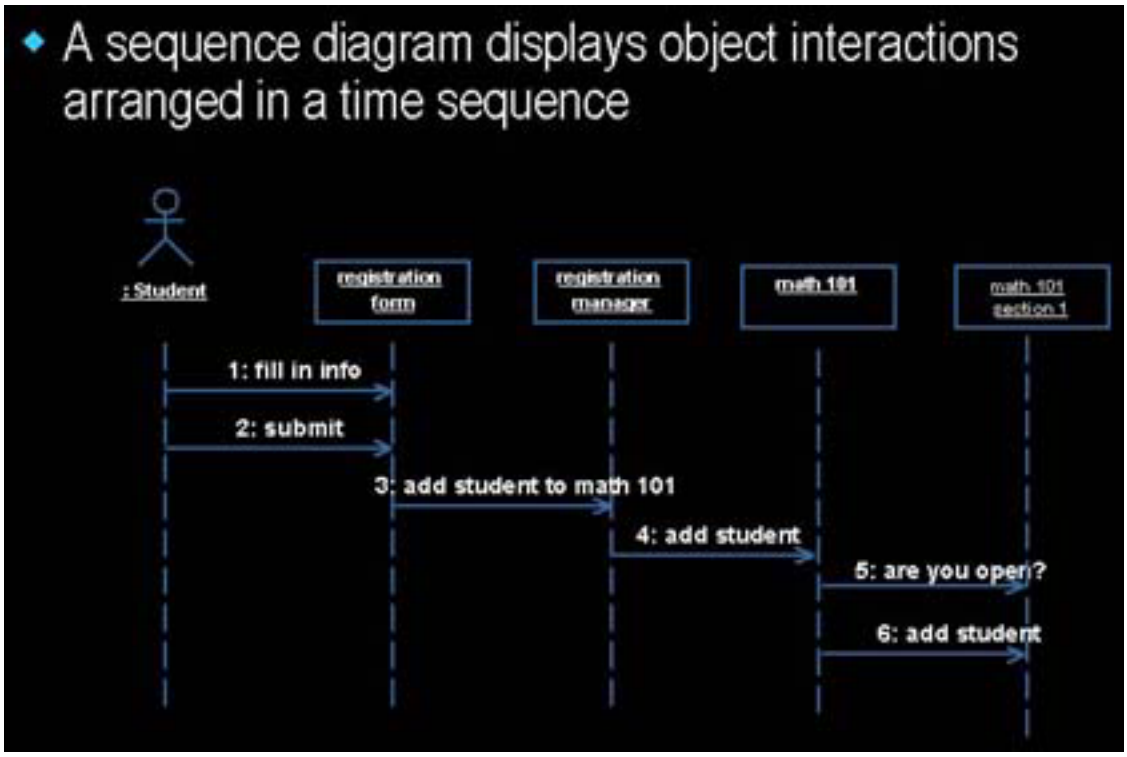


Figure 6: Sequence diagram

Figure 6 shows how a student successfully gets added to a course. The student (let's call him Joe) fills in some information and submits the form. The form then talks to the manager and says "add Joe to Math 101." The manager tells Math 101 that it has to add a student. Math 101 says to Section 1 "are you open?" In this case, Section 1 replies that they are open, so Math 101 tells section 1 to add this student. Again, sequence diagrams are great tools in the beginning because they show you and your customer step-by-step what has to happen.

From an analysis point of view, I've found over the years that sequence diagrams are very powerful in helping me drive requirements; especially requirements that are hard to find. User interface requirements, for instance, are notorious because you always seem to get requirements that are just not testable. A common UI requirement like this is "this system shall be user-friendly." How many of you have met a friendly computer? One of the benefits of these types of diagrams is that every line coming from an actor that represents a person, tells you that something in your UI has to provide a capability needed by that person. In other words, you can use sequence diagrams to drive out testable user interface requirements.

Sequence diagrams are, therefore, good for showing what's going on, for driving out requirements, and for working with customers. That usually leads to the question, though, of how many do you need to create? My answer is, "until you do enough." You're going to find out when you do sequence diagrams that you reach a point where you're not finding any new objects, not finding any new messages, and that you're typing the same thing over and over. In the example of Joe joining Math 101, we learn that the process would be the same if Joe wanted to join History 101. So, rule of thumb, do a sequence diagram for every basic flow of every use case. Do a sequence diagram for high-level, risky scenarios, and that should be enough. That's how many sequence diagrams I do.

Collaboration Diagrams

The next diagram I want to talk about is a collaboration diagram. And, I must confess, I don't use this diagram very much. It's a different view of a scenario – one where I have objects, but they're not ordered according to time. They're shown based on the links between the objects.

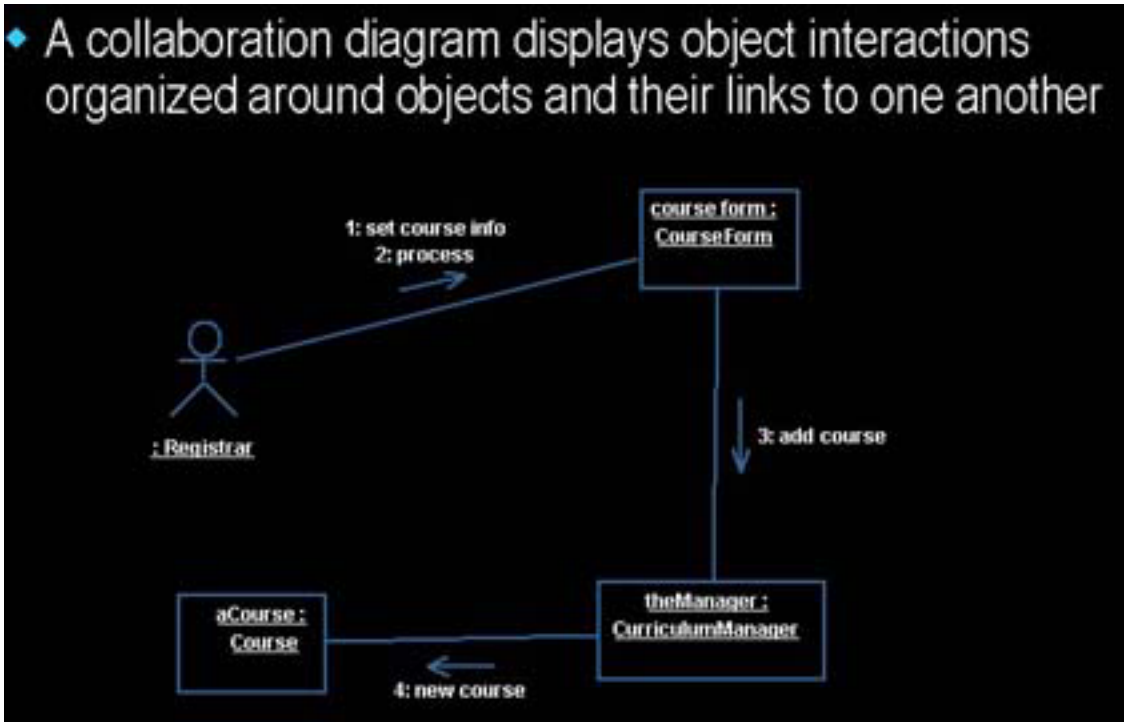


Figure 7: Collaboration diagram

The benefit of collaboration diagrams is if I want to see all of the messages that go between two objects for a particular use case or scenario. Especially in the case of a big, long scenario where the real estate is smaller, it's easier to see these messages on a collaboration diagram. The thing to remember here, is that a collaboration diagram is just a different view of a scenario and you can go back and forth between sequence diagrams and collaboration diagrams to get the view that best illustrates your point.

Occasionally, you might hear the phrase "interaction diagrams." Sometimes people will collectively refer to a collaboration diagram and a sequence diagram as an interaction diagram.

Class Diagrams

A class is a collection of objects with common structure, common behavior, common relationships, and common semantics. You find them by examining the objects in sequence and collaboration diagrams, and they are represented in the UML as a rectangle with three compartments.

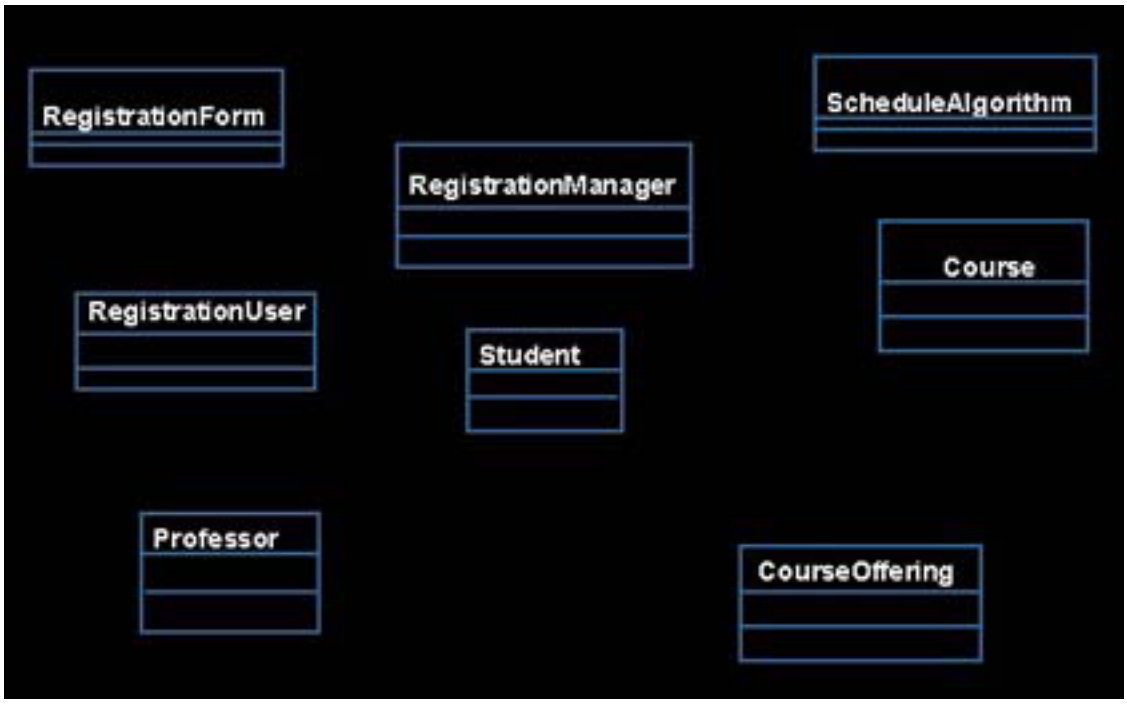


Figure 8: Classes

The first compartment shows the class name, the second shows its structure (attributes), and the third shows its behavior (operations). These compartments can be suppressed, however, so that you can see just the name, just the name and the attributes, or all three. One thing you should also know is that it's important, when naming classes, to use the vocabulary of the domain and pick a standard. For this instance, my classes are all singular nouns that begin with a capital letter. You may choose to do it differently, and that doesn't matter. What does matter is that before your project you pick a standard and stick with it so that everything is consistent across the project.

Class Diagrams show you the static nature of your system. These diagrams show the existence of classes and their relationships in the logical view of a system. You will have many class diagrams in a model.

The UML modeling elements found in class diagrams include:

- Classes and their structure and behavior.
- Association, aggregation, dependency, and inheritance relationships.
- Multiplicity and navigation indicators
- Role names.

Take a look at **Figure 9**. This diagram shows operations (behavior): what an object in that class can do. I find my operations by looking at my interactions diagrams.

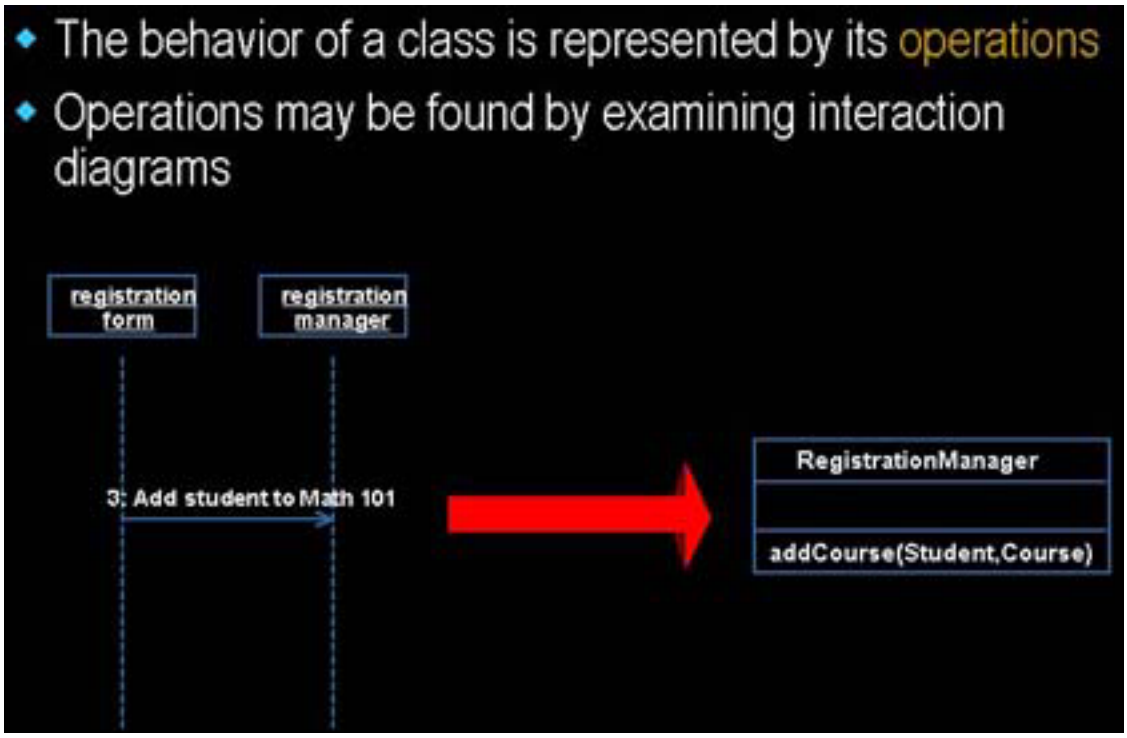


Figure 9: Operations

Here I'm saying that I need to be able to ask the registration manager to add a student to Math 101. That's going to translate into an operation called "addCourse."

The structure of a class is represented by its attributes. So how do I find my attributes? By talking to domain experts. By looking at my requirements. In my example, I learn that each course offering has a number, a location, and a time. This translates out to three attributes.

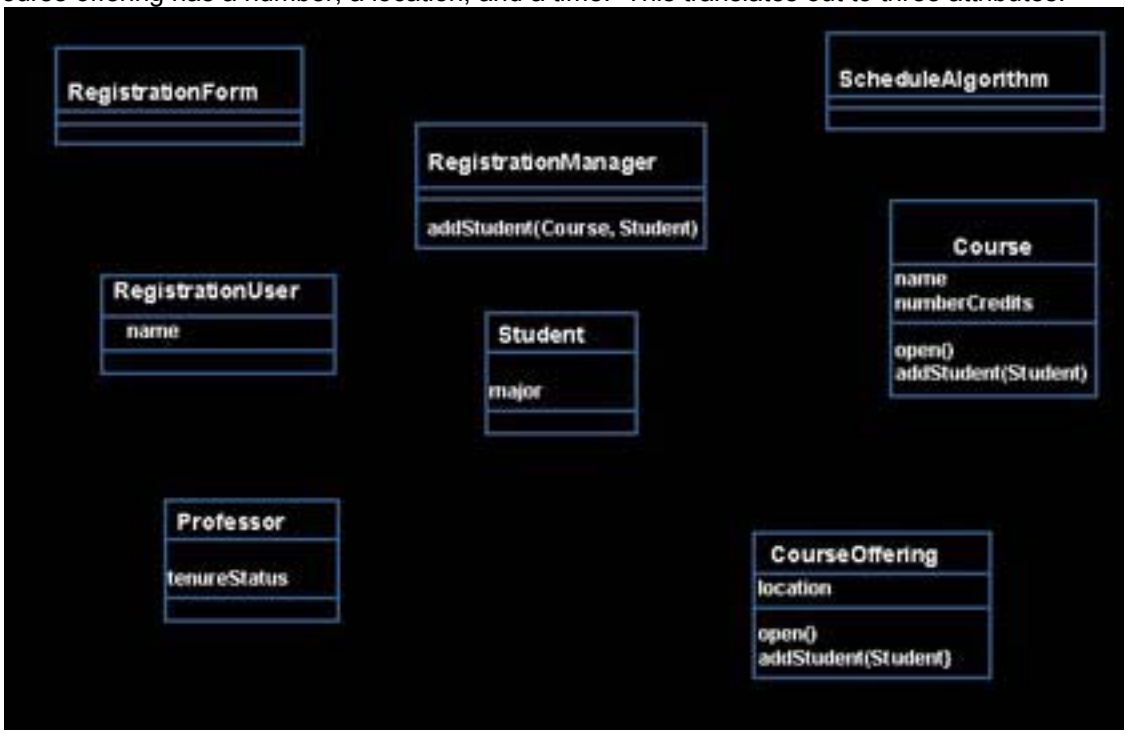


Figure 10: Classes and their attributes

Relationships

Relationships represent a communication path between objects
There are three types of UML relationships: Association, Aggregation, and Dependency.

Association – a bi-directional connection between classes. An association says "I can send you a message because if I'm associated with you, I know that you're there." (Represented in the UML as a line connecting the related classes.)

Aggregation – a stronger form where the relationship is between a whole and its parts. An aggregation tells my developer that there's a strong coupling between those object classes. (Represented in the UML as a line connecting the related classes with a diamond next to the class representing the whole.)

Dependency – a weaker form showing the relationship between a client and a supplier where the client does not have semantic knowledge of the supplier. A dependency says "I need your services, but I don't know that you exist." (Represented in the UML as a dashed line pointing from the client to the supplier.)

To find relationships, once again, I go back to my sequence diagram. If two objects need to "talk", there must be a means to do so (i.e., a relationship between their classes).

I typically start out and make everything an association. As I'm doing more analysis, I might find I have an aggregation because I'm going to have to take care of a parent-child relationship. When I get into the design phase, I find out that I might not need an association because somebody else is going to pass that object into one of my methods – so I make it a dependency.

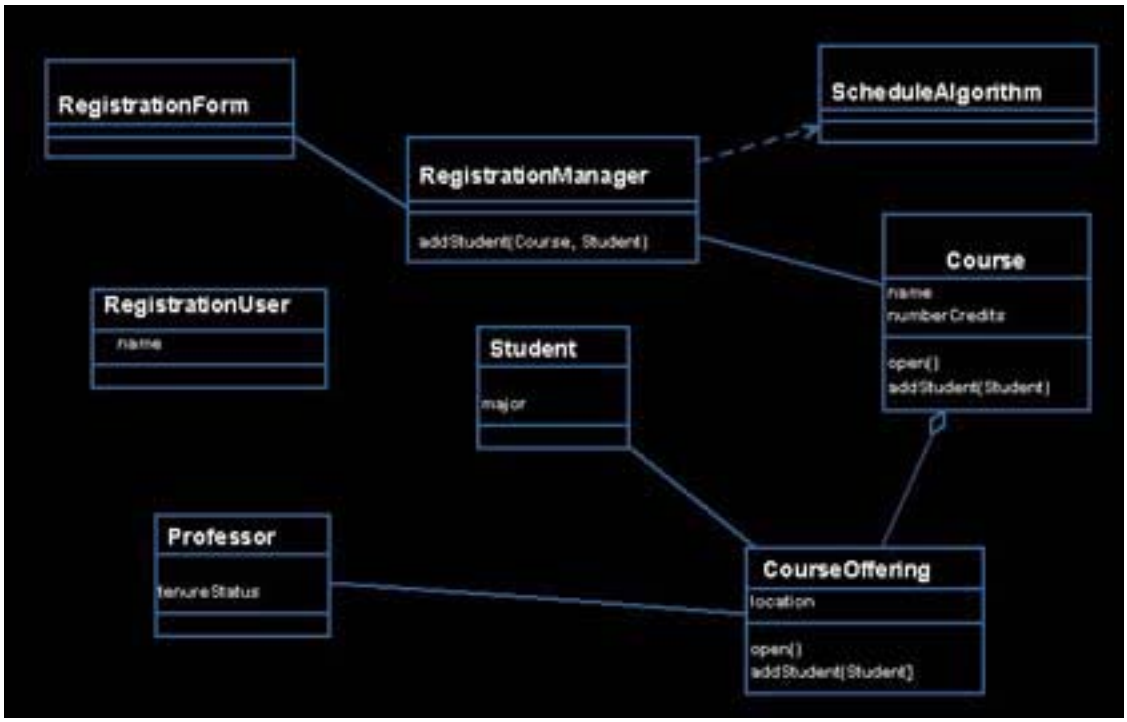


Figure 11: Relationships

In **Figure 11** you see these relationships. As association says the Professor can talk to the Course Offering, and the Course Offering can talk to the Professor. Messages can be initiated and data can flow from any direction. Aggregation is shown by having the diamond toward the whole – in this case a Course is made up of Course Offerings. The reason for this aggregation would be to tell my developers that if they get rid of this Course, they'll probably have to do something special with the Course Offerings. Dependencies are shown as a dashed line. It's saying that the registration manager depends upon the Schedule Algorithm to do something. The Schedule Algorithm is either a parameter to one of the methods or is declared locally by one of the Methods of the Registration Manager.

Multiplicity and Navigation

Multiplicity defines how many objects participate in a relationship. It is the number of instances of one class related to *one* instance of the other class. For each association and aggregation, there are two multiplicity decisions to make: one for each end of the relationship. Multiplicity is represented as a number and a * is used to represent a multiplicity of many.

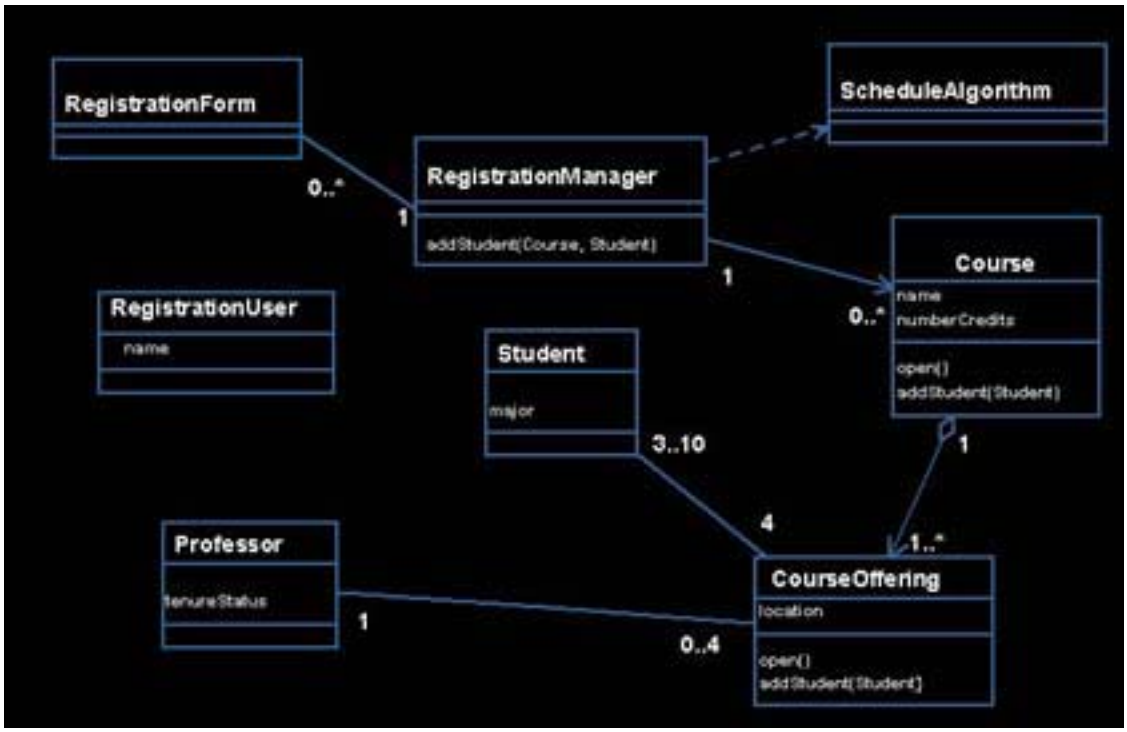


Figure 12: Multiplicity and navigation

One Professor Object is related to zero-to-four Course Offering Objects. One Course Offering Object is related to exactly one Professor Object. I use this to look at and ensure that this handles my requirements. Can I be a Course Offering and be team-taught by a bunch of professors? No, because this says I can only have one professor. Can I be a professor and be on sabbatical? Yes, because this says I have a zero as possible course load. I use multiplicity quite often to help me start capturing and implementing my business rules. If you have, for example, a business rule that says you must have at least 3 students and no more than 10 for a course to be offered in a semester, these multiplicity numbers tell me I've incorporated that business rule into this plan.

Navigation is shown by an arrow, and although associations and aggregations are bi-directional by default, it is often desirable to restrict navigation to one direction. When navigation is restricted, an arrowhead is added to indicate the navigational direction. One of the things I do during the analysis and design phases is look at what I want to be uni-directional. By putting the arrow into this diagram, I say that the Registration Manager can send a message to the Course, because it knows the Course exists. But the Course has no idea that the Registration Manager exists, so the Course cannot initiate a message. Now data can flow between them; for instance the Registration Manager can ask the Course if it's open and the Course can say that it is. But only the Registration Manager can start that conversation.

Obviously the goal here is to get as many arrows as you can by the time you've finished designing, because it's a much easier system to maintain.

Inheritance

Inheritance is the relationship between a superclass and a subclass. It shows uniqueness and commonality, and allows me to add new behavior as I'm moving along but not changing the susperclass.

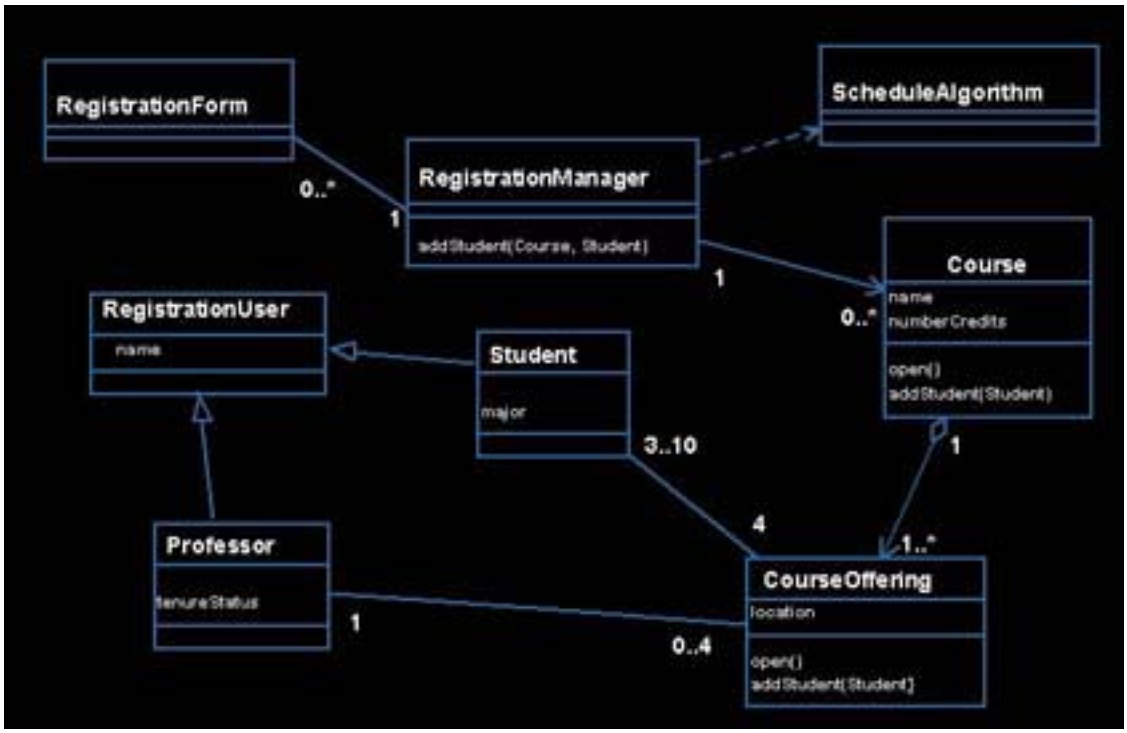


Figure 13: Inheritance

Inheritance is shown with a triangle. This shows that the Professor is a Registration User, as is the Student. Now, a word of warning. Inheritance is useful, however, the goal is not to use as much inheritance as your system will allow. I've seen some really brutal systems where they had inheritance 17-levels deep. If they changed one thing, it became a disaster. So the rule of thumb is to use inheritance *only* when you truly do have an inheritance situation.

State Transition Diagrams

A state transition diagram shows the life history of a given class. It shows the events that cause a transition from one state to another, and the actions that result from a state change.

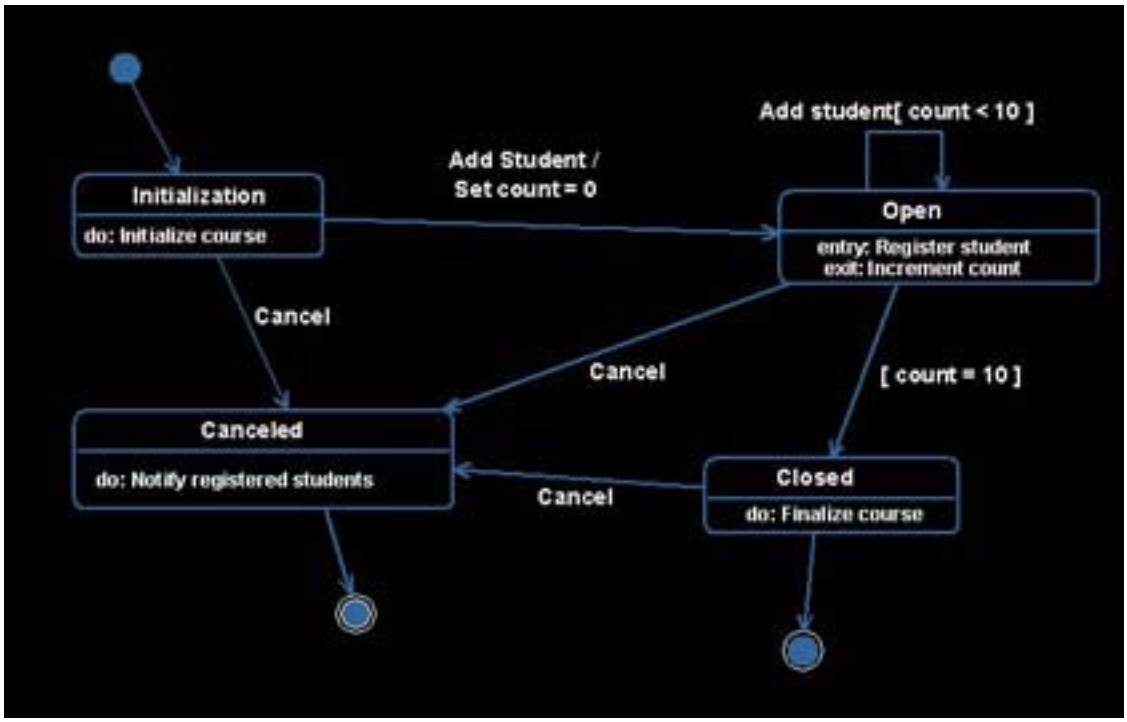


Figure 14: State transition diagram

I use state transition diagrams for object classes that typically have a lot of dynamic behavior. The button is on … the button is off; I'm not going to do a state chart for it. But object classes that have a lot of dynamic behavior, I'm probably going to have to look into the states of the objects.

I start by showing a state, which is a rounded triangle. I can have start states, and I can have stop states, which are shown as bulls eyes. I can also have transitions between states, or guard transitions (things that happen when only when a condition is true), or things that happen when I'm inside the state. I look at this diagram and see the state transition diagram for a course offering. It starts in the initialization state, and I stay in that state until I get an "add student" message. When I get that message, I set my count of student to zero and I transition to the Open state. You'll see in **Figure 14** that I have an entry, and the reason why it's there is that I have two ways of getting into that state. It says that no matter how you come into the state, I want you to register the student. When I exit that state, the count changes to keep track of the number of students in the course. I can keep adding students until I get to 10, and then I go to the Close state. Once the course is finalized, I transition to the stop state. No matter where I am then, if I get the Cancel Event transition, I notify my students and then transition to the stop state.

For object classes that have a lot of dynamic behavior, it's well worth it to do a state diagram to get a handle on everything that has to happen. Ask yourself what happens when I get a message? What do I do when I get the message? What messages to I have to send? A lot of those messages become operations of the object class, as in this example where add a student is an operation. A lot of these actions, like setting the count, incrementing the count, checking the count, these all become private operations of that particular object class and a state diagram is where I see that.

How do you know if you have a dynamic object class? Once again, go back to the sequence diagrams. If you have an object class that's on a lot of sequence diagrams and it's getting and sending a lot of messages, that's a good indication it's a fairly dynamic object class and it should probably have a state chart for it. Also for aggregations, where you have the whole of its parts, I

do a state chart for every aggregate whole. I do this mostly because that aggregate whole is often responsible for managing the messaging, which makes it dynamic.

Component diagrams

Of course no system can be built without taking into account the physical world. That's where component diagrams come in. They are used to illustrate the organizations and dependencies among software components, including source code components, run time components, or an executable component. Components are shows as a large rectangle with two smaller rectangles on the side, as seen in **Figure 15**.

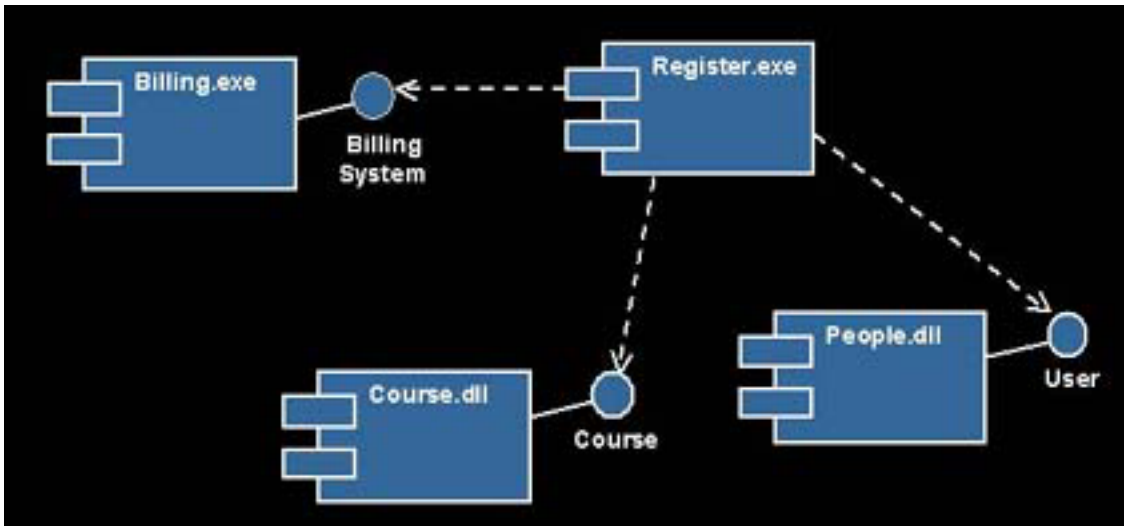


Figure 15: Components

Those round things represent interfaces (often called lollipop notation). In this case, they show that the Register.exe is dependent upon interfaces to both the Course.dll and the People.dll. That means if these interfaces change, it will impact the Register.exe. I know that there's this rule when you're building interfaces that says "thou shall not change the interface." But does anybody actually work where that rule is enforced? This diagram tells us what interfaces are used by what executables, so if the interface changes you know where the impacts may occur.

Deployment Diagrams

When it comes time to think about deploying the system, deployment diagrams are crucial because they show the processors on your system and the connections between them. They also visualize the distribution of components across the enterprise. It's a visual way of knowing what executables are running on my processors

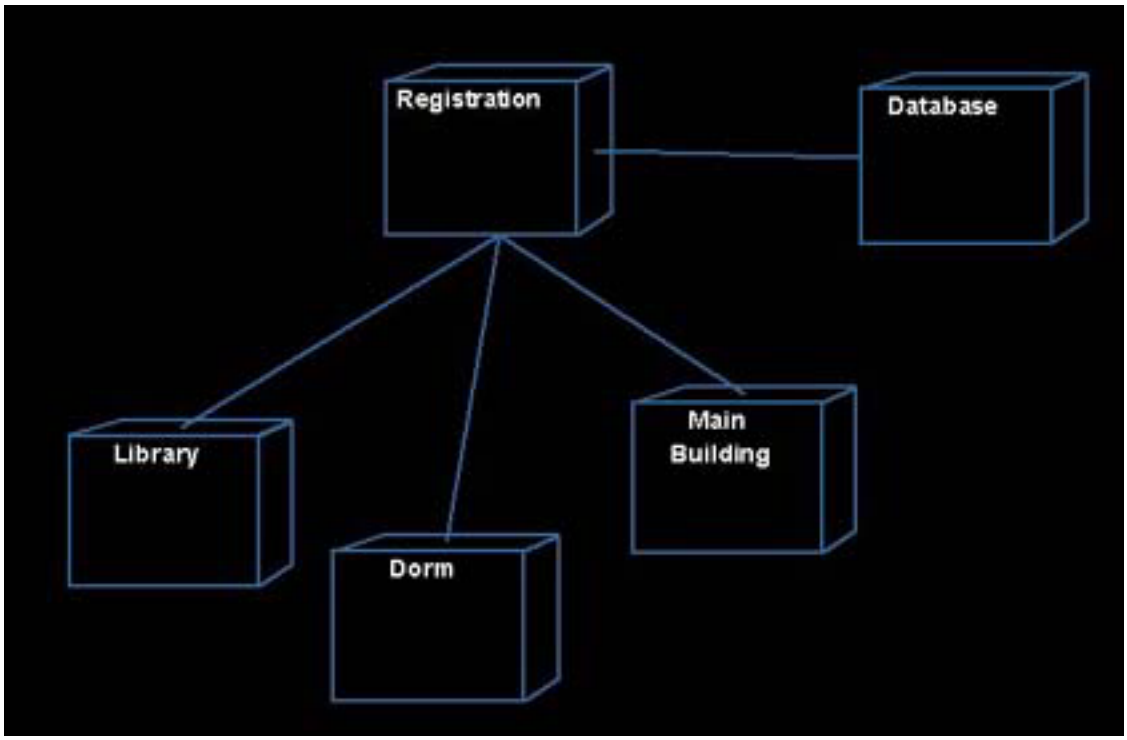


Figure 16: Deployment diagram

Extending UML

The last thing I want to stress about the UML is that it can be extended. When they built the UML, they very wisely realized that there was no way they could create a notation that could please all of the people all of the time. So they gave us the concept of a stereotype. A stereotype says I can take a basic modeling element and give it more meaning. Stereotypes may be used to classify and extend associations, inheritance relationships, classes, and components.

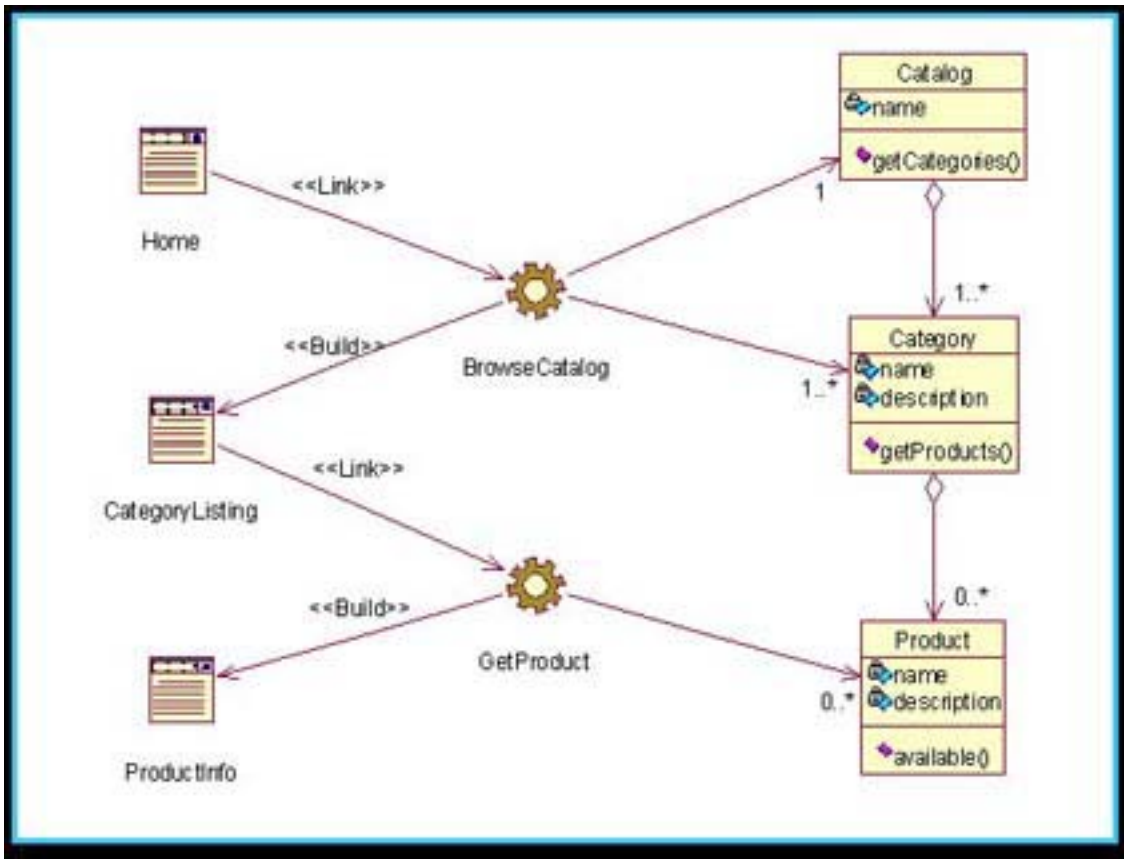


Figure 17: Web stereotype example

Figure 17 shows the diagram of our Web stereotypes. A Web page typically has stuff that runs on the server and stuff that runs on the client. If you're building Web-based applications, what's running on the client and the server is of vital importance. So we have a whole set of stereotypes that show that. The little wheels represent things that run on the server. So just by looking at this one diagram I can see what runs on the server, what runs on the client, and what business objects they have to deal with.

So that's what I call UML 101. Obviously it's not all of UML, in fact there's a thousand pages to the UML spec. But it's a starting point. And for those of you who want to know more, I can make a few recommendations about where to go.

Visual Modeling with Rational Rose 2002 and UML, by Terry Quatrani. Yes, it's my own book, but I think it's a good source of information, even if I do say so myself. Basically I walk you through a process of using the Rational Unified Process, Rational Rose, and UML.

UML Distilled, by Martin Fowler. This is an excellent book explaining UML diagrams and the notations behind it.

UML Explained, by Kendall Scott. – As the title suggests, Kendall does a good job of introducing key UML concepts and describing their uses.

There are also the UML books by the amigos.

The Unified Modeling Language User Guide and The Unified Modeling Language Reference Manual.