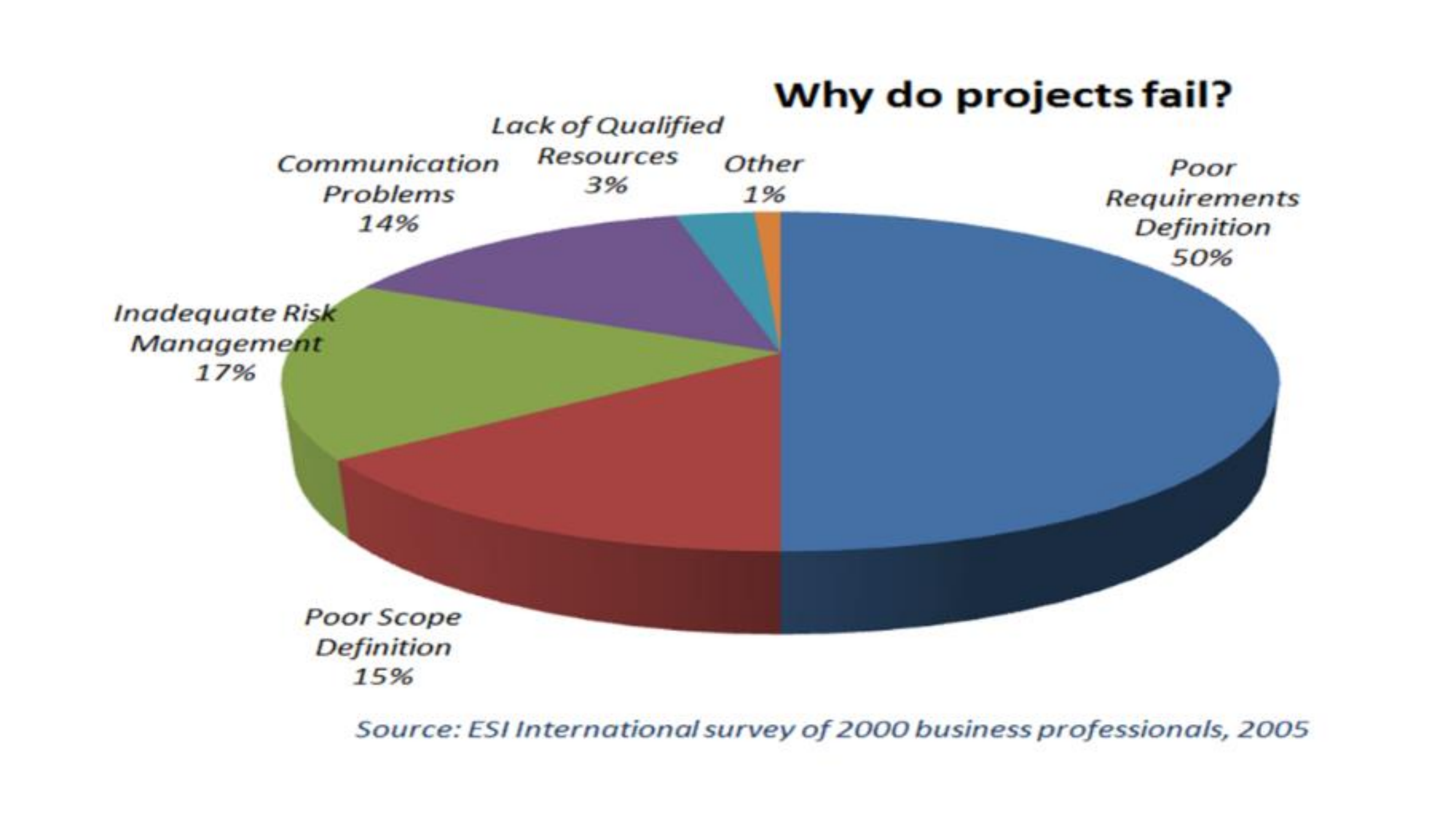
Introduction to Software Engineering

Dr. Ganesh Bhutkar

Vishwakarma Institute of Technology, Pune, India

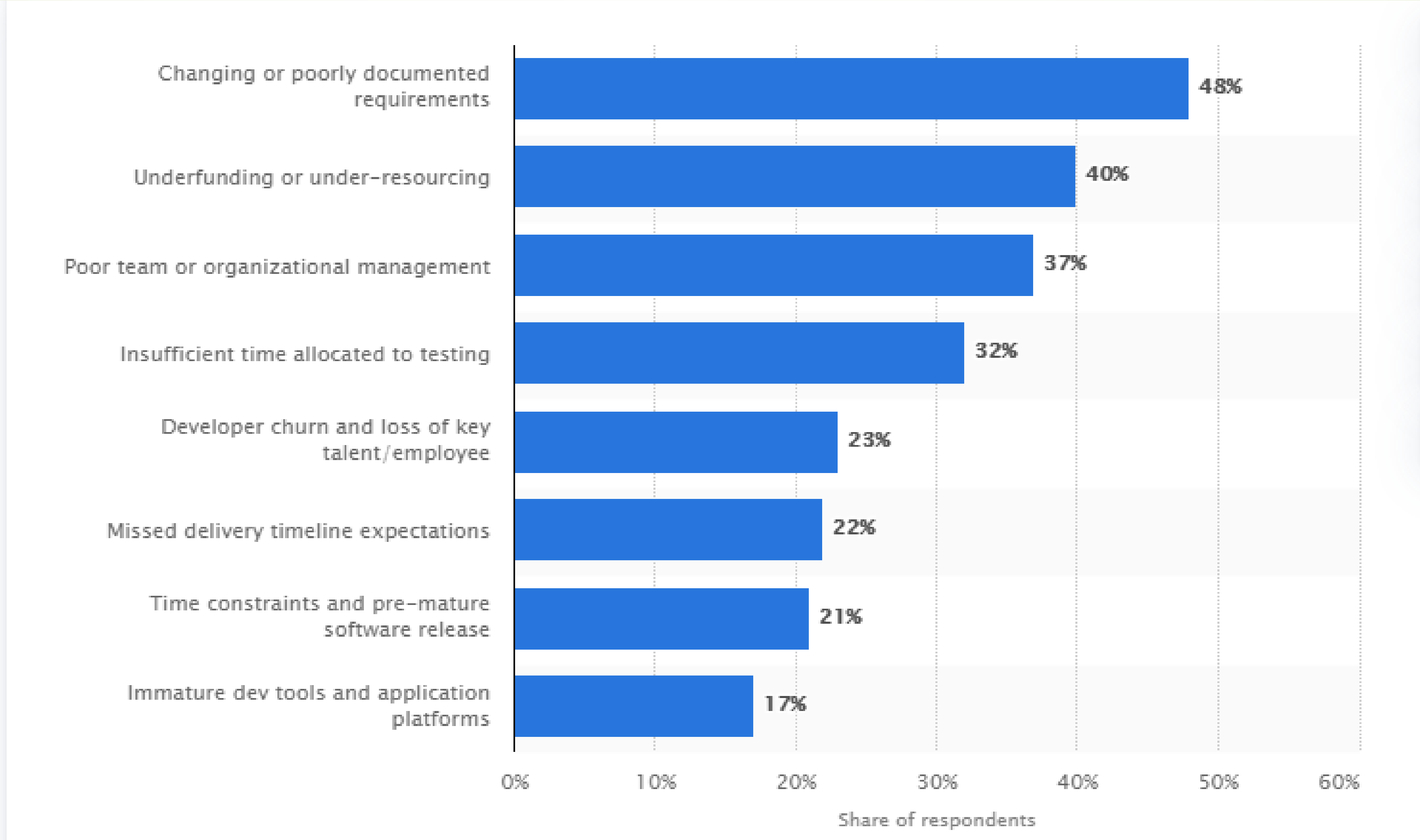
Course: Software Modelling And Design (SMAD) – CS3218

Why do Projects fail?



Reference: http://apppm.man.dtu.dk/index.php/File:Why_projects_fail.png

Project Failure Factors



Reference: https://www.statista.com/statistics/627648/worldwide-software-developer-survey-project-failure/

Software Engineering:
A Problem Solving Activity

- **Analysis**: Understand the nature of the problem and break the problem into pieces
- **Synthesis**: Put the pieces together into a large structure

For problem solving, we use -

- **Techniques / Methods**:
  - Formal procedures for producing results using some well-defined notation
- **Methodologies:**
  - Collection of techniques applied across software development and unified by a philosophical approach
- **Tools:**
  - Instrument or automated systems to accomplish a technique

Definition: Software Engineering

Software Engineering is a collection of techniques, methodologies and tools that help with the production of -

- **a high quality software system,**
- **with a given budget,**
- **before a given deadline**

with **evolving requirements and changes** in given environment.

Software Engineer

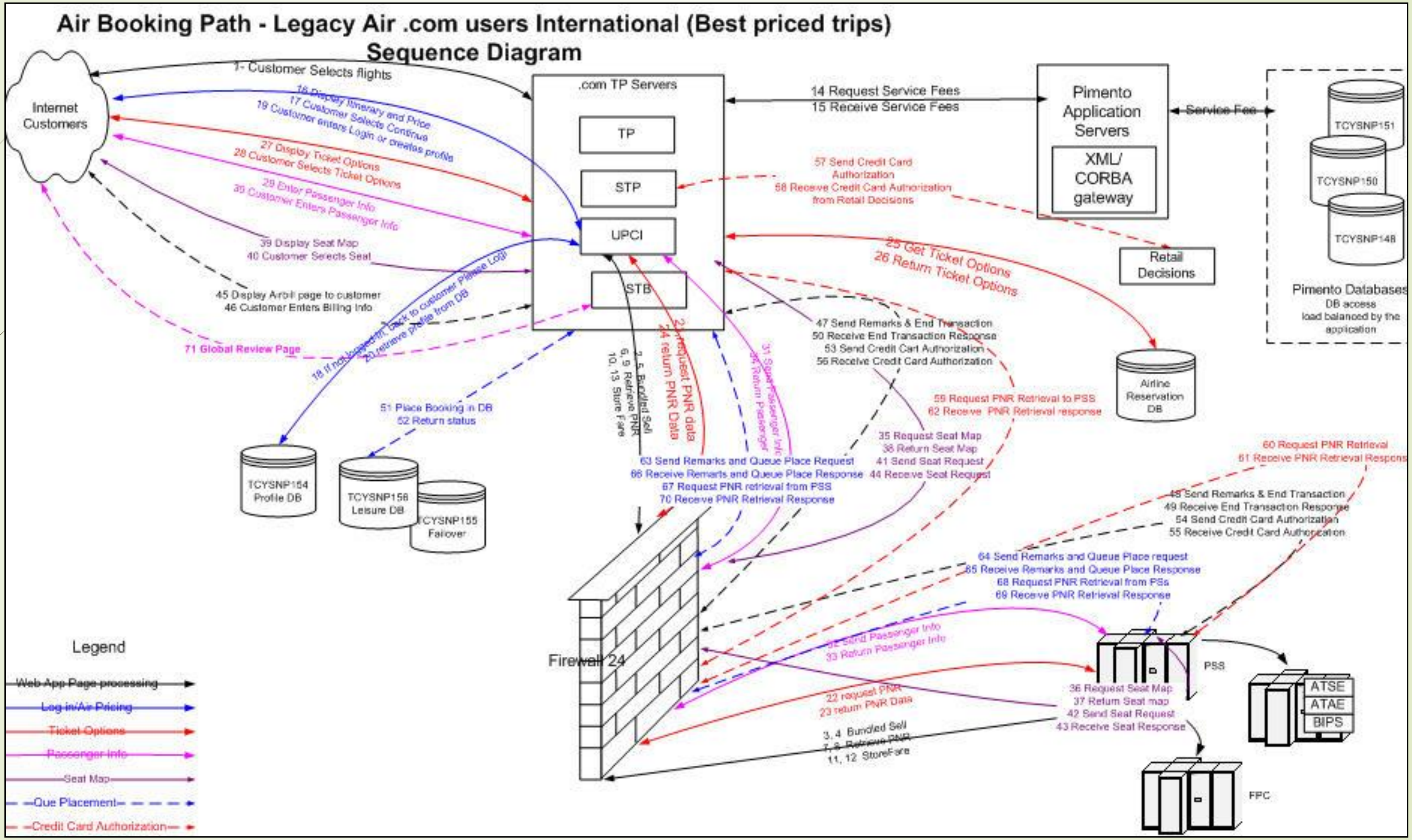
- **Computer Scientist / Researcher**
  - Proves computational theorems / algorithms / concepts,
  - Designs languages, prototypes & defines knowledge / databases,
  - Has longer time (Months & Years)
- **Engineer**
  - Develops a solution for an application-specific problem for a client,
  - Uses computers, languages, tools, techniques and methods,
  - Has long time (Months & Years)
- **Software Engineer – domains, deadlines, learning, internationalization, health**
  - Works in **multiple application domains**
  - Has **deadlines with short duration (Weeks & Months)**
  - Deals with **changes occurring in requirements and available technology**

Challenges in Quality Assurance of Software Systems

- **Complexity:**
  - The system is so complex that **no single programmer / engineer can understand** it anymore.
  - The introduction of one bug fix causes many other **bugs (Cascading Effect)**

- **Changes:**
  - Each implemented **change erodes / spoils the structure** of the system, which makes the next change even more expensive.
  - **As time goes on, the cost to implement a change will be too high**, and the system may not be able to support its intended task completely.

Complex Message Flow



System Models

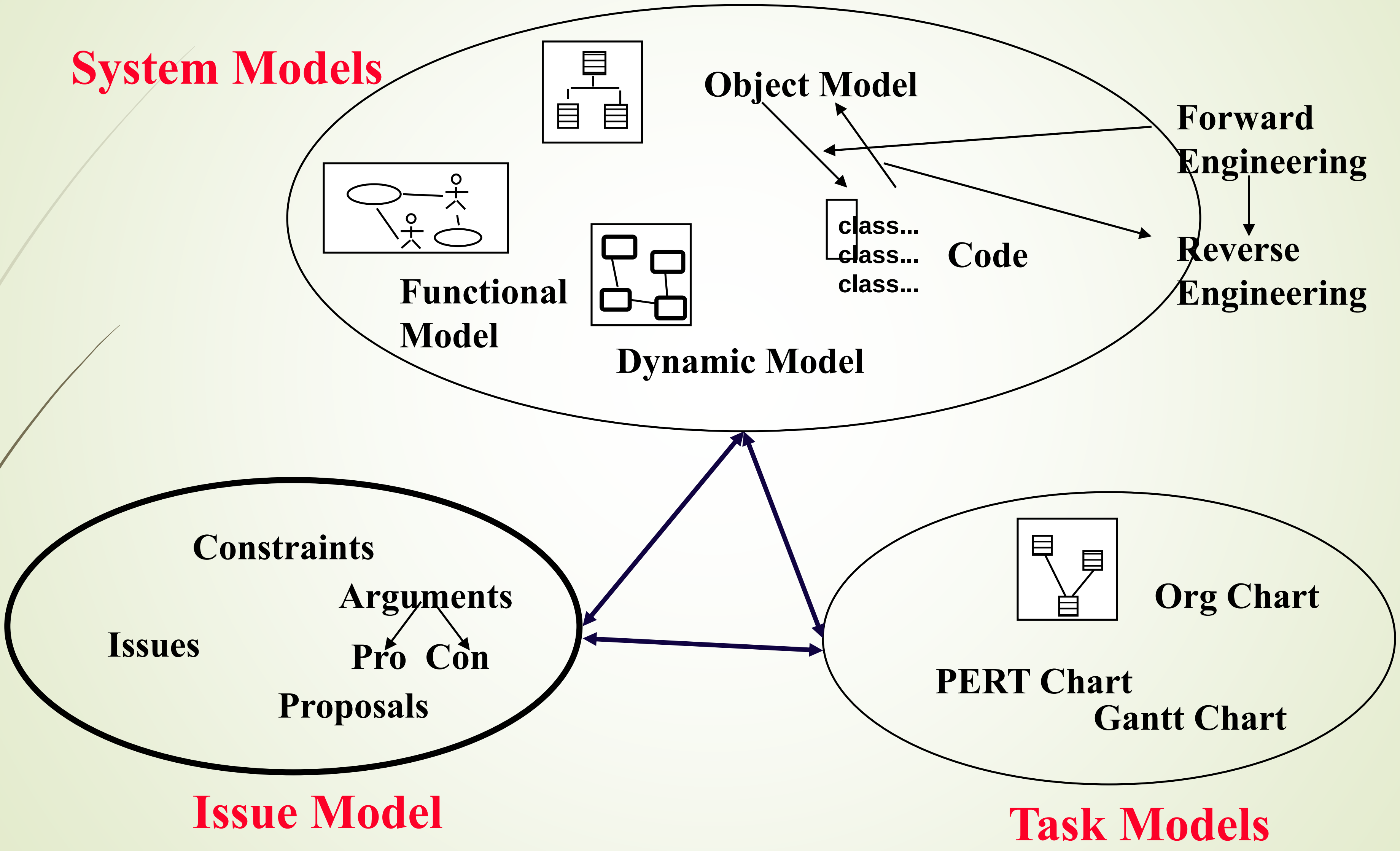
- A system model is a **process-oriented representation that emphasizes the influences or flow of information between modules.**

- A system model describes **how processes interact and what operations these processes perform, approximately.**

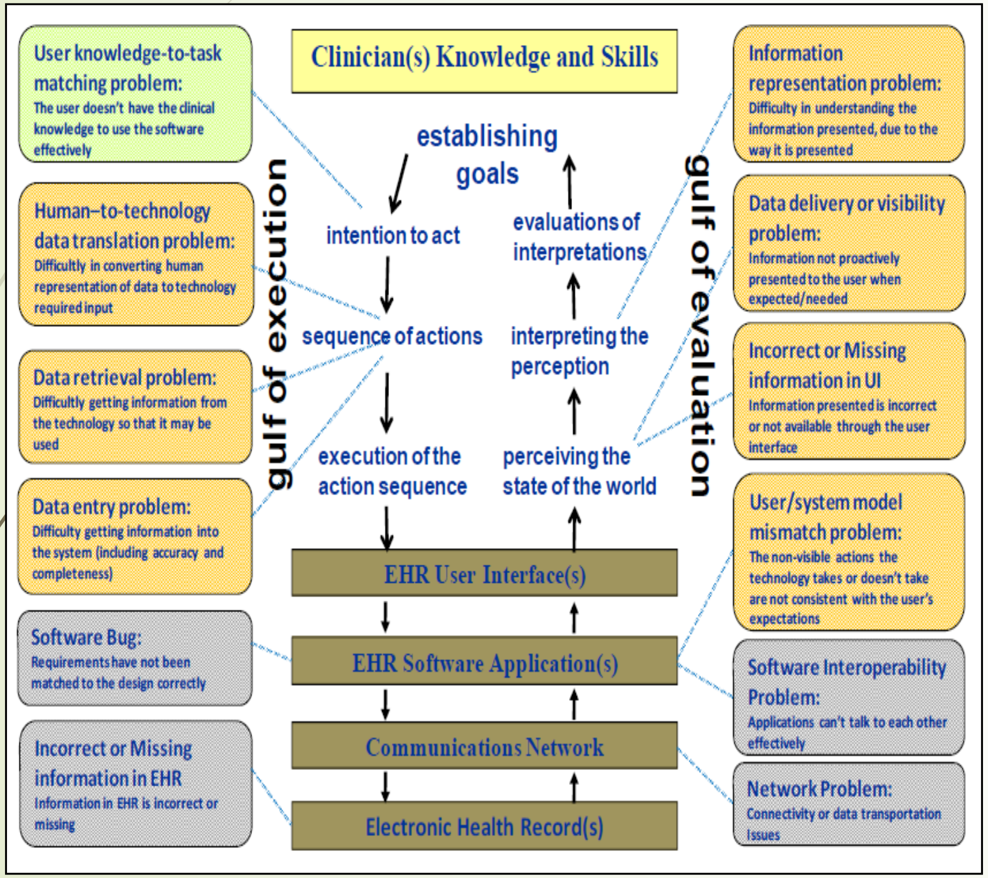
- **Types of System Models –**

  - **Conceptual Models -** **Concept / Phenomenon**
  - **Mathematical Models -** **Mathematics / Statistics**
  - **Computational Models -** **Algorithms / Computational Entities**

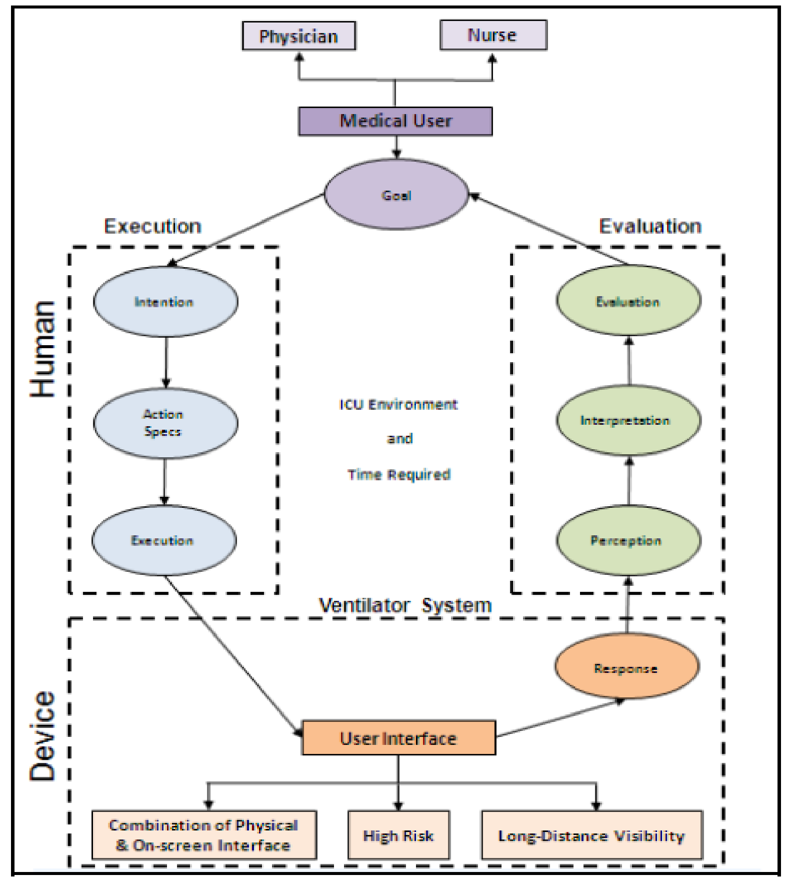
Bermuda Triangle of Modelling



Examples of Popular Models



Patient Safety Informatics Cognitive Model (PSI-CAM) by Chapman et al. for Cataloguing Errors (2012)



Usability Model for Medical UI of Ventilator System by Bhutkar et al. (2013)

Thank You !

Software Development Paradigms

Dr. Ganesh Bhutkar

Vishwakarma Institute of Technology, Pune, India

Course: Software Modelling And Design (SMAD) – CS3218

Symptoms of Software Development Problems

- **Inaccurate understanding** of end-user needs
- Inability to deal with **changing requirements**
- Modules that don't fit together (**integration**)
- Software that's **hard to maintain** or extend (brittle)
- Late discovery of **serious project flaws** (validation)
- **Poor software quality** (architecture, risks unanticipated…)
- **Process not responsive** to Change (Gantt Charts…)
- **Unacceptable** software **performance**
- Team members in each other's way, unable to reconstruct **who changed what, when, where, why** (software architecture, …

…and we could go on and on…

Reference:

Need a Better Hammer !

►We need a **process** that will serve as a framework for large scale as well as small projects.

►Process needs to be

- **Adaptive** – embraces 'change!'
- **Iterative** (small, incremental 'deliverables')
- **Risk-driven** (identify / resolve risks up front)
- **Flexible & customizable** (not a burden; adaptive to projects)
- **Architecture-centric** (breaks components into 'layers' or common areas of responsibility…)
- Supporting **effective user involvement**

►Identify best ways of doing things – a **better process** – acknowledged in global IT World…

Reference:

Software Development Processes

- **Two Major Processes**
  - **Engineering** – development and quality steps needed to engineer the software
  - **Project Management** – planning and controlling the development process
- **Key Roles**
  - **Software developers** execute engineering process.
    - Software architects, lead developers, ...
  - **Project manager(s)** executes the management process.

Software Process…

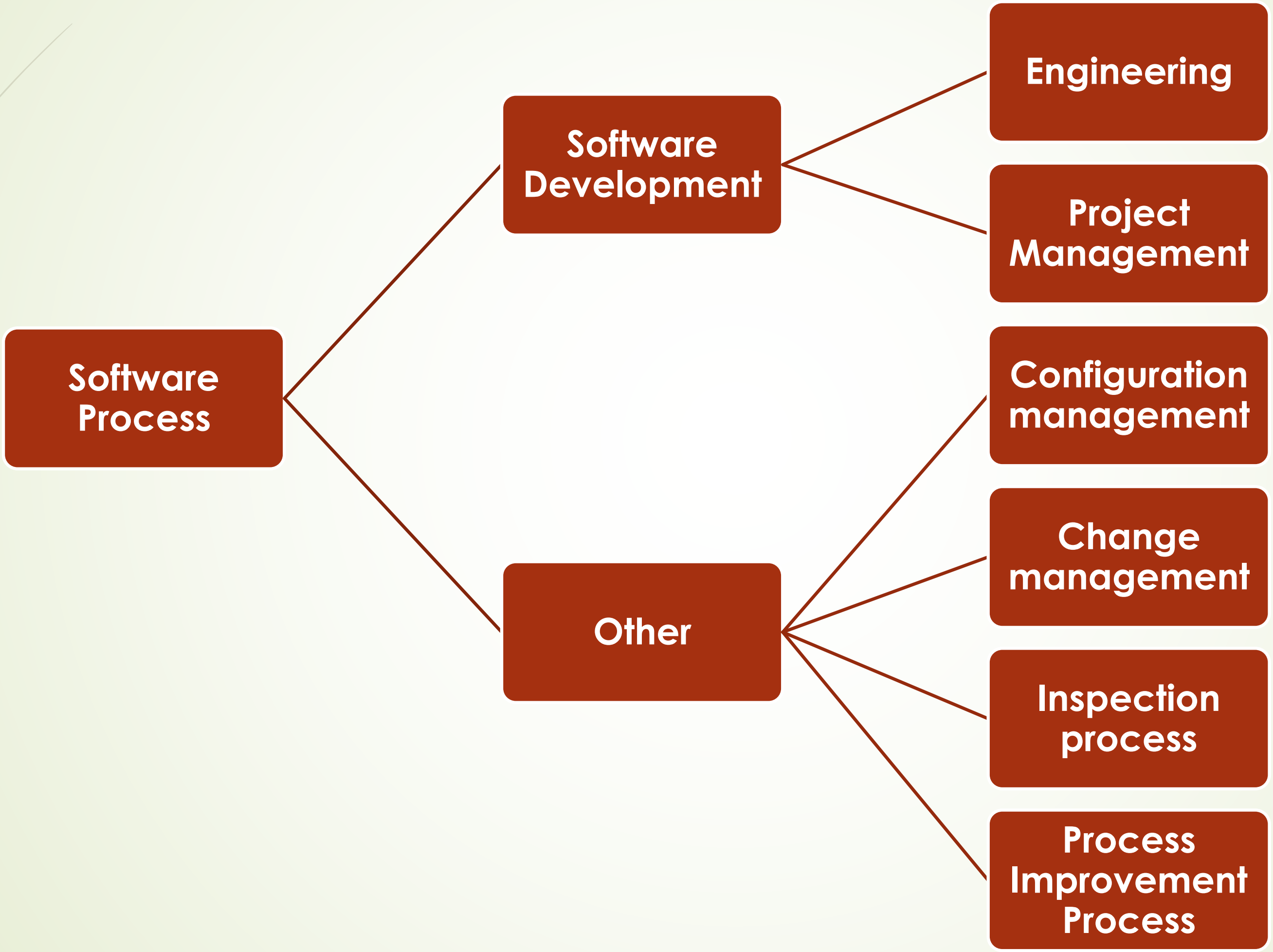
- **Process:** A particular method, generally involving a number of steps.

- **Software Process:** A set of steps, along with ordering constraints on execution, to produce software with desired outcome.

- Many types of activities are performed by **different people**.

- Software process is comprising of many component **sub-processes.**

Key Processes



Software Development Life Cycle (SDLC)

- **SDLC** is a **process used by the software industry** to design, develop and test a quality software.
- It is a **framework that describes the activities** performed at each stage of a software development project.
- There are **5-7 stages / phases** of SDLC.

Software Development Paradigms

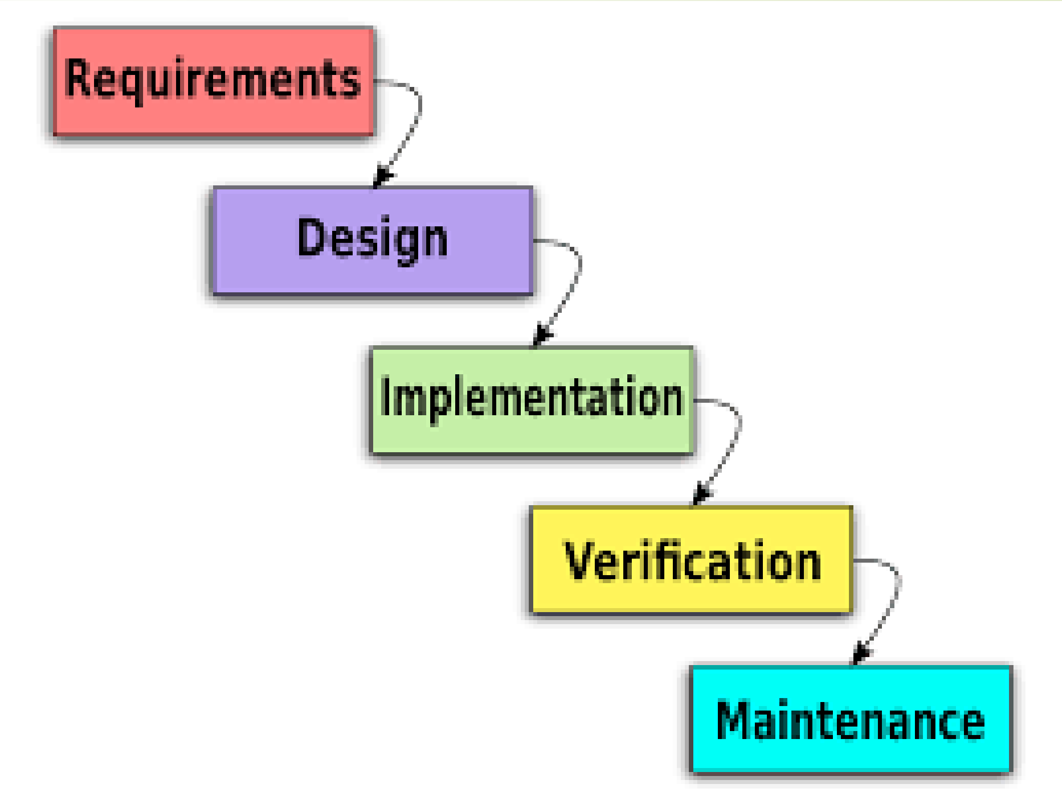
- Software development paradigms are **all the engineering concepts pertaining to developments software applied**. They consist of the process such as **Requirement Gathering, Software Design, Programming**, **Testing** etc. They are the **parts of software development**.

- Traditionally, Software developers have experimented with three major software development paradigms: **Procedural, Data-driven and Object-oriented.**

- **Waterfall model** is the simplest model of software development paradigm. It says the all the phases of SDLC will function one after another in linear manner.

Waterfall Model

- **Requirements –** needed information, functions, behavior, performance & interfaces.
- **Design –** data structures, software architecture, interface representations, & algorithmic details.
- **Implementation –** source code, database, user & documentation.



Waterfall Model: Strengths

- **Easy to understand** & easy to use
- **Provides structure** to inexperienced staff
- **Milestones are well understood**
- Sets **requirements stability**
- Good for **management control** (plan, staff, track)
- Works well when **quality is more important** than cost or schedule

Waterfall Model: Deficiencies

- All **requirements must be known** upfront.
- Deliverables created for each phase are considered frozen – **inhibits flexibility.**
- Can give a **false impression of progress.**
- **Does not reflect problem-solving nature** of software development – iterations of phases.
- Integration is **one big bang at the end.**
- **Little opportunity for customer** to preview the system (until it may be too late).

When to Use the Waterfall Model

- Requirements are very **well-known.**
- Product definition is **stable.**
- Technology is **well-understood.**
- It's a **new version of an existing product.**
- **Porting an existing product** to a new platform.

Code and Fix Model

- The code and fix model probably is the **most frequently used development methodology** in software development.
- It starts with **little or no initial planning.** You immediately start developing, fixing problems as they occur, until the project is complete.

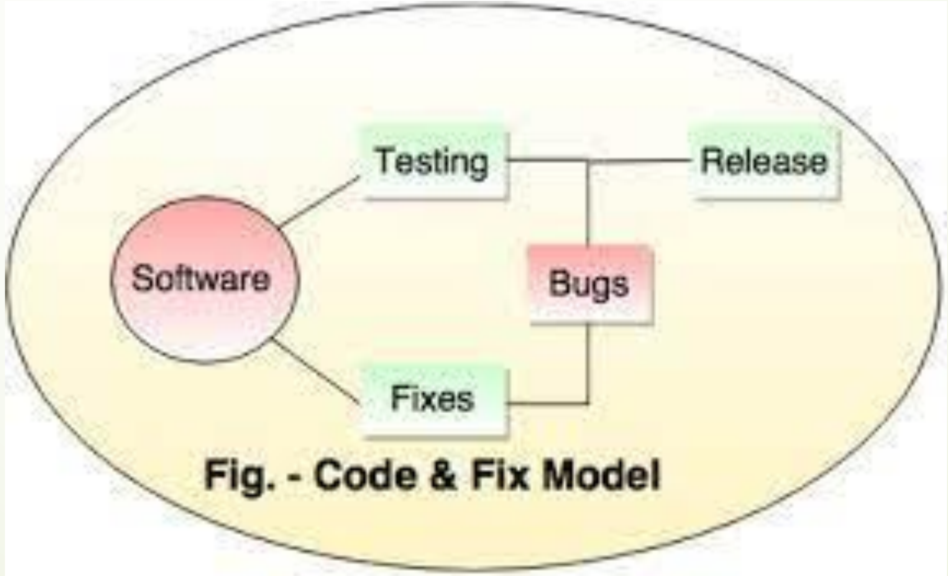
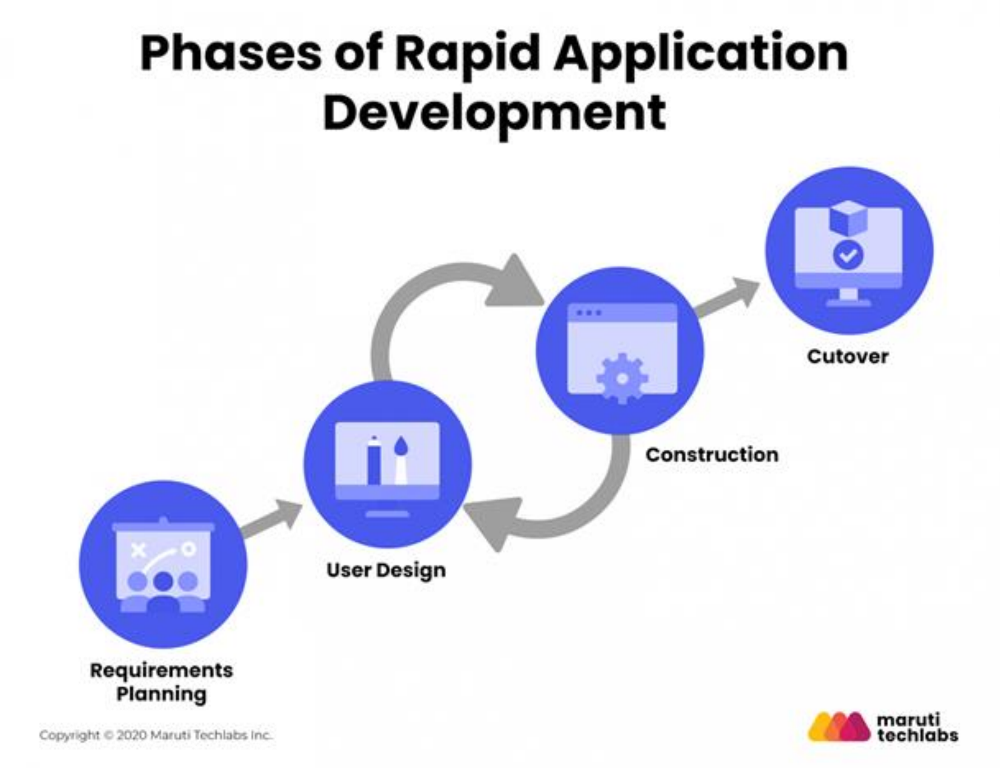


Fig. - Code & Fix Model

Rapid Application Development (RAD) Model



- **Requirements planning phase -** a structured discussion of business problems
- **User design / description phase -** automated tools capture information from users
- **Construction phase -** productivity tools, such as code generators, screen generators, etc. inside a time-box ("Do until done")
- **Cutover phase –** an installation of the system, user acceptance testing and user training

RAD Strengths

- **Reduced cycle time** and improved productivity with fewer people means lower costs
- **Time-box** approach mitigates cost and schedule risk
- **Customer involved throughout** the complete cycle minimizes risk of not achieving customer satisfaction and business needs
- Focus moves from documentation to code (**WYSIWYG**)
- **Uses modeling concepts** to capture information about business, data, and processes

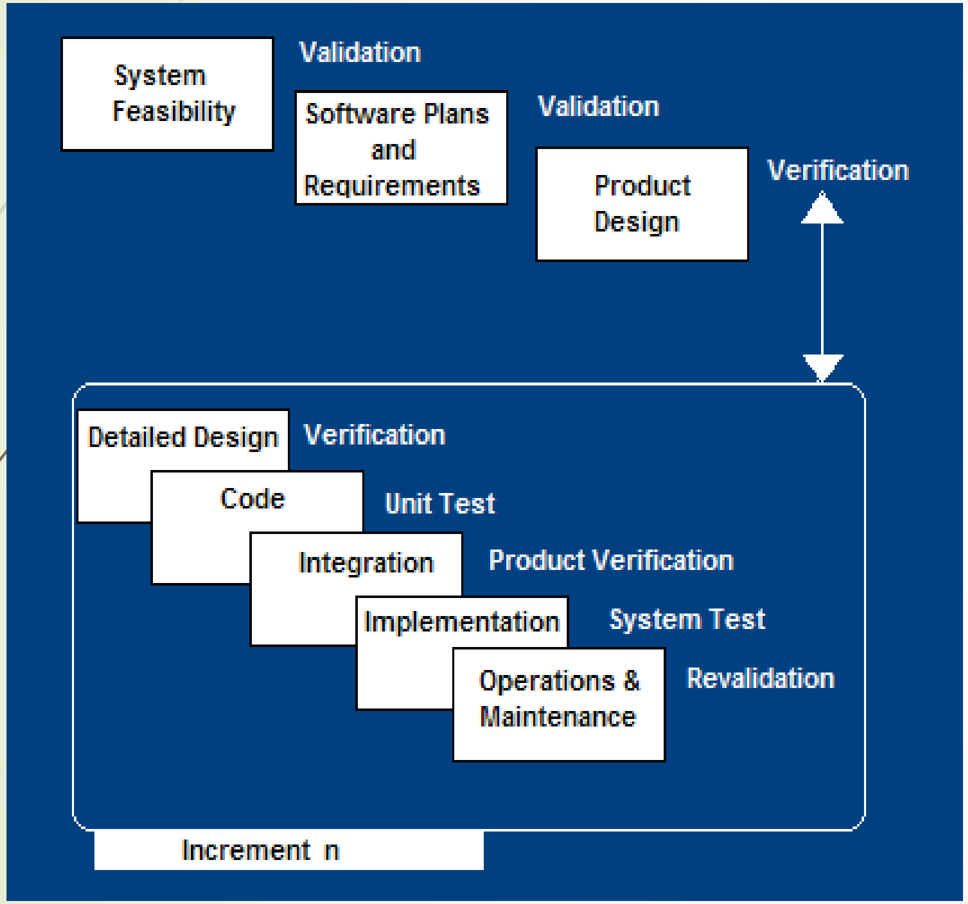
RAD Weaknesses

- Accelerated development process **must give quick responses** to the user
- Risk of **never achieving closure**
- Hard to use with **legacy systems**
- Requires a system that can be **modularized**
- Developers and customers must be **committed to rapid-fire activities** in an abbreviated time frame.

When to Use RAD Model

- Reasonably **well-known requirements**
- Users involved **throughout the life cycle**
- Project can be **time-boxed**
- Functionality delivered in **increments**
- **High performance not required**
- **Low technical risks**
- System **can be modularized**

Incremental Model



- Construct a **partial implementation** of a total system
- Then, slowly update **added / increased functionality**
- The incremental model **prioritizes requirements** of the system and then **implements them in groups.**
- Each subsequent release of the system **adds function to the previous release**, until all designed functionality has been implemented.

Incremental Model Strengths

- Develop high-risk or **major functions first**
- Each release delivers an **operational product.**
- Customer can **respond to each build.**
- Uses "divide and conquer" **breakdown of tasks**
- Lowers **initial delivery cost**
- Initial **product delivery is faster.**
- Customers get **important functionality early.**
- Risk of **changing requirements is reduced.**

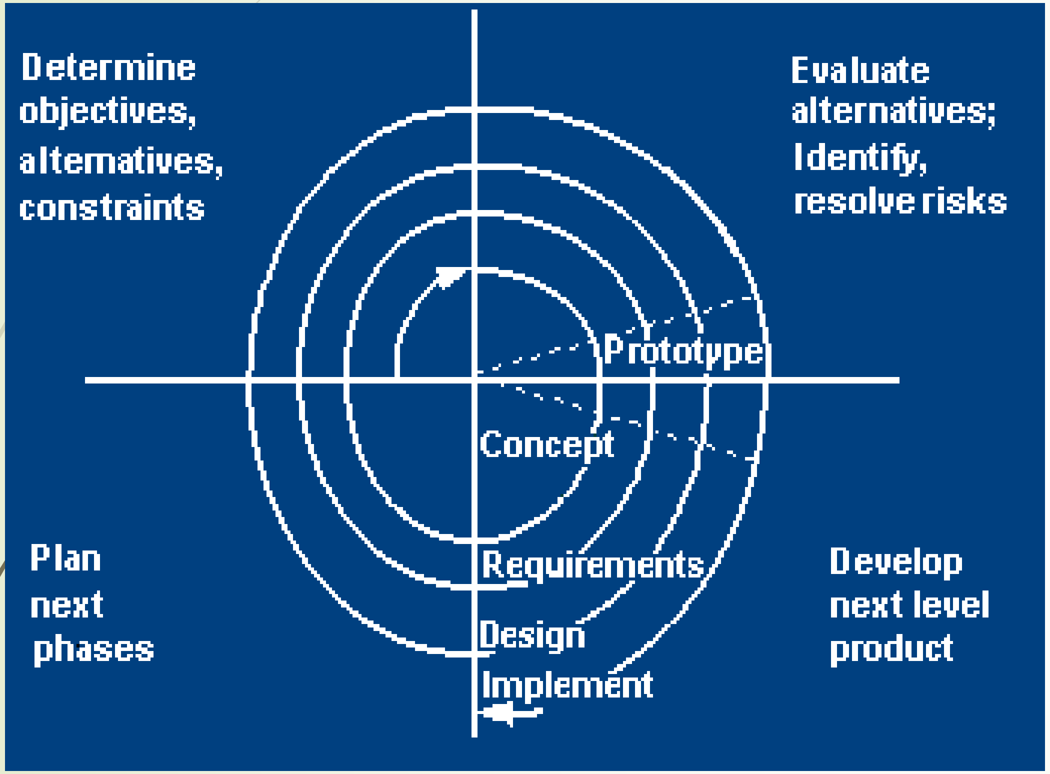
Incremental Model Weaknesses

- Requires **good planning and design**
- Requires **early definition of a complete and fully functional system** to allow for the definition of increments
- **Well-defined module interfaces** are required (some will be developed long before others).
- Total **cost** of the complete system is **not lower.**

When to use the Incremental Model

- Risk, funding, schedule or program complexity - a need for **early realization of benefits**
- Most of the requirements are known up-front but are expected to **evolve over time.**
- A need to **get basic functionality to the market early**
- On projects which have **lengthy development schedules**
- On a project with **new technology**

Spiral Model



- Adds **risk analysis**, and **prototyping** to the waterfall model
- Each **cycle** involves the **same sequence** of steps as the waterfall process model

Spiral Quadrant - I
Determine objectives, alternatives & constraints

- **Objectives:** functionality, performance, hardware / software interface, critical success factors & others
- **Alternatives:** build, reuse, buy, sub-contract & others
- **Constraints:** cost, schedule, interface & others

Spiral Quadrant - II
Evaluate alternatives, identify & resolve risks

- **Study alternatives** relative to objectives and constraints
- **Identify risks** (lack of experience, new technology, tight schedules, poor process & Others)
- **Resolve risks** (evaluate if money could be lost by continuing system development)

Spiral Quadrant - III
Develop next-level product

- Typical activities:
  - Create a design
  - Review design
  - Develop code
  - Inspect code
  - Test product

Spiral Quadrant - IV
Plan next phase

- Typical activities:
  - Develop **project plan**
  - Develop **configuration management plan**
  - Develop a **test plan**
  - Develop an **installation plan**

Spiral Model Strengths

- Provides **early indication of insurmountable risks,** without much cost
- Users see the system early because of **rapid prototyping tools**.
- **Critical high-risk functions** are developed first.
- The **design** does **not** have to be **perfect**.
- **Users** can be closely tied to **all lifecycle steps**.
- Early and **frequent feedback from users**
- Cumulative **costs are assessed frequently**.

Spiral Model Weaknesses

- **Time spent for evaluating risks is too high** for small or low-risk projects.
- **Time spent planning, resetting objectives and prototyping** may also be **excessive**.
- The model is **complex**.
- Risk **assessment expertise is required**.
- Spiral **may continue indefinitely**.
- May be **hard to define objective, verifiable milestones** that indicate readiness to proceed through the next iteration

When to use Spiral Model

- When **creation of a prototype** is appropriate
- When **costs and risk evaluation** is important
- For **medium to high-risk projects**
- **Long-term project commitment**
- **Users** are **unsure** of their **needs**.
- **Requirements** are **complex**.
- **New product line**
- **Significant changes** are **expected** (research and exploration)

Iterative Software Development

- Until recently, developed under assumption - **Most requirements can be identified up front.**

- The research deconstructing this myth includes work by Capers Jones. (**See next slide**) In this very large study of 6,700 projects, **creeping requirements** — those not anticipated near the start — are a **very significant fact of software development life**, ranging from around **25% on average projects up to 50% on larger ones**.

Graph – Requirements vs Project Size

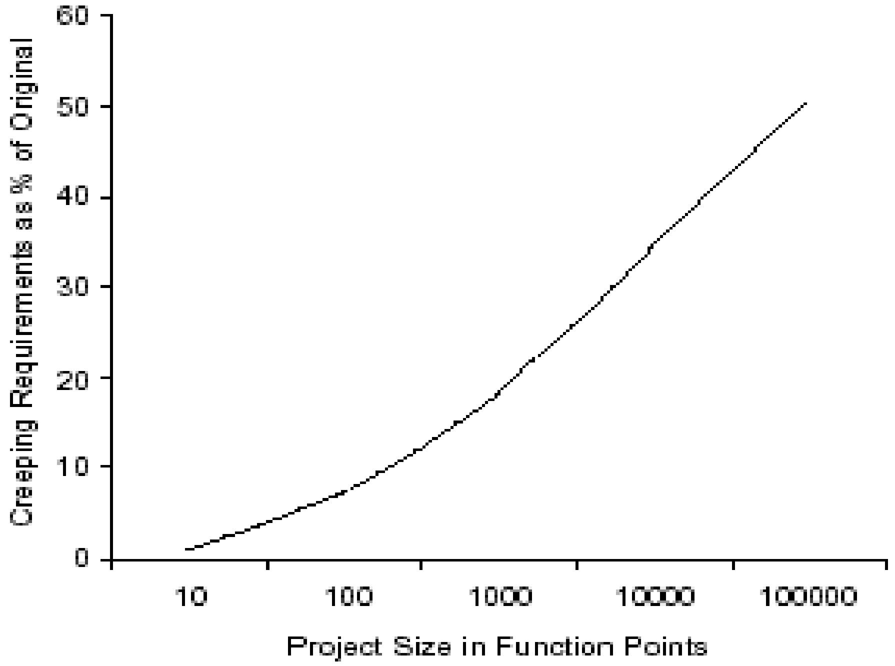


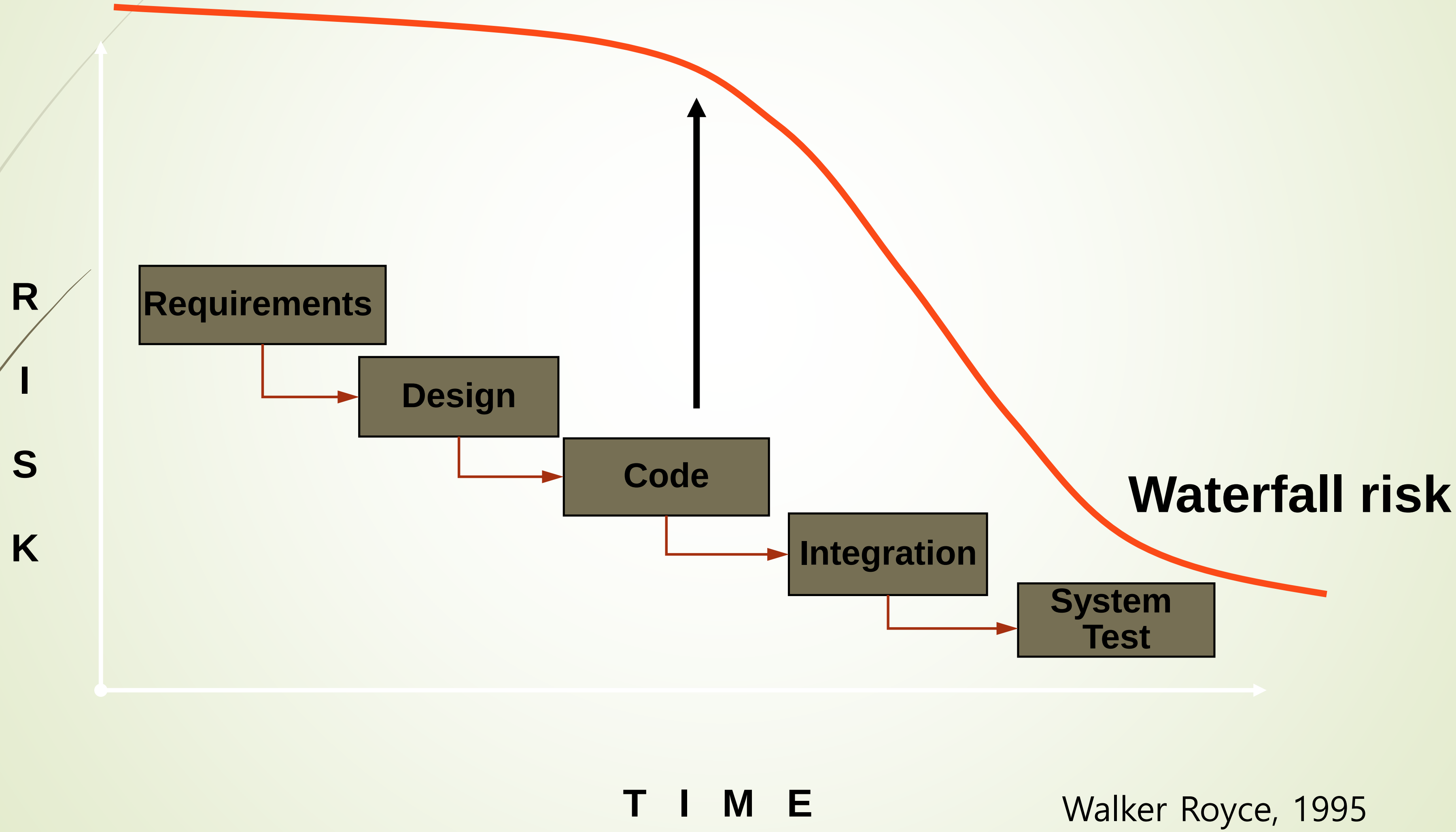
Figure 1 Changing Requirements are the Norm

Interestingly,

- An **initial design will likely be flawed** with respect to its key requirements. **Requirements rarely fully known up front !**
- **Late-phase discovery of design defects** results in costly over-runs and / or project cancellation.
  - Oftentimes **requirements change** – even during implementation!

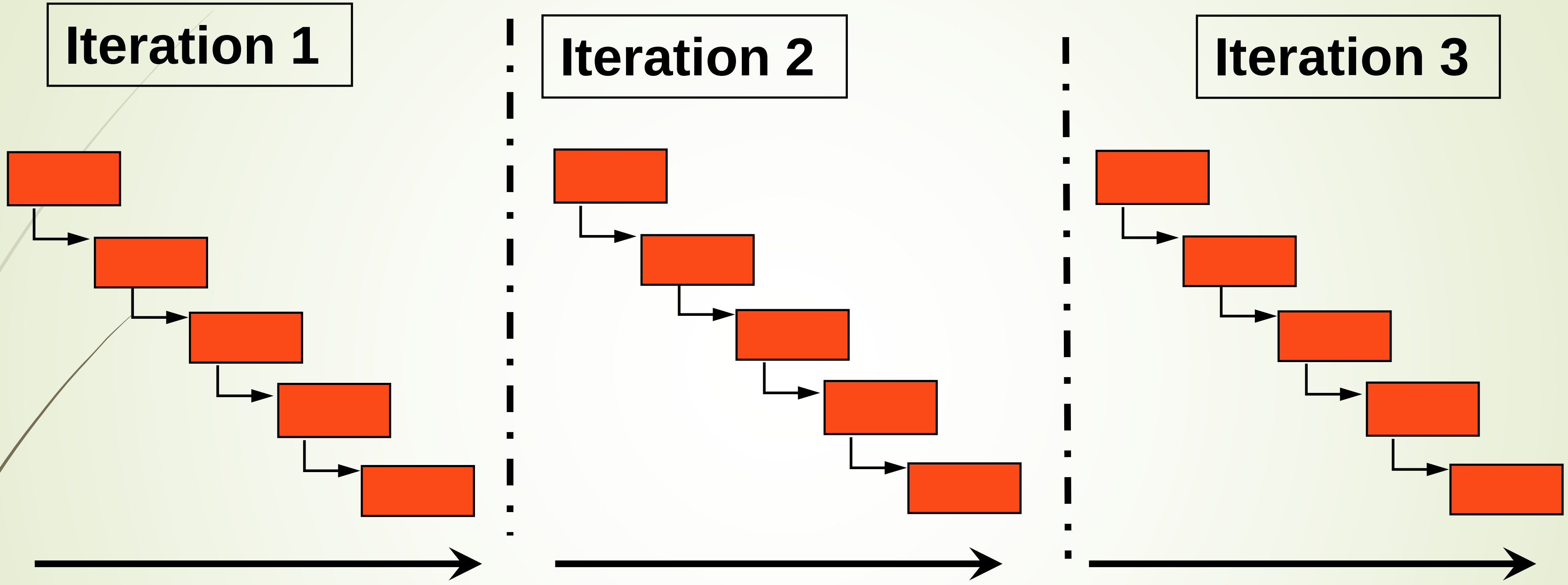
- While large projects are more prone to **cost overruns**, medium-size / small projects are vulnerable to **cancellation**.
- The **key reasons** continue to be -
  - **poor project planning and management,**
  - **shortage of technical and project management expertise,**
  - **lack of technology infrastructure,**
  - **disinterested senior management &**
  - **inappropriate project teams.**

Waterfall Delays Risks



Walker Royce, 1995

Iterative Development

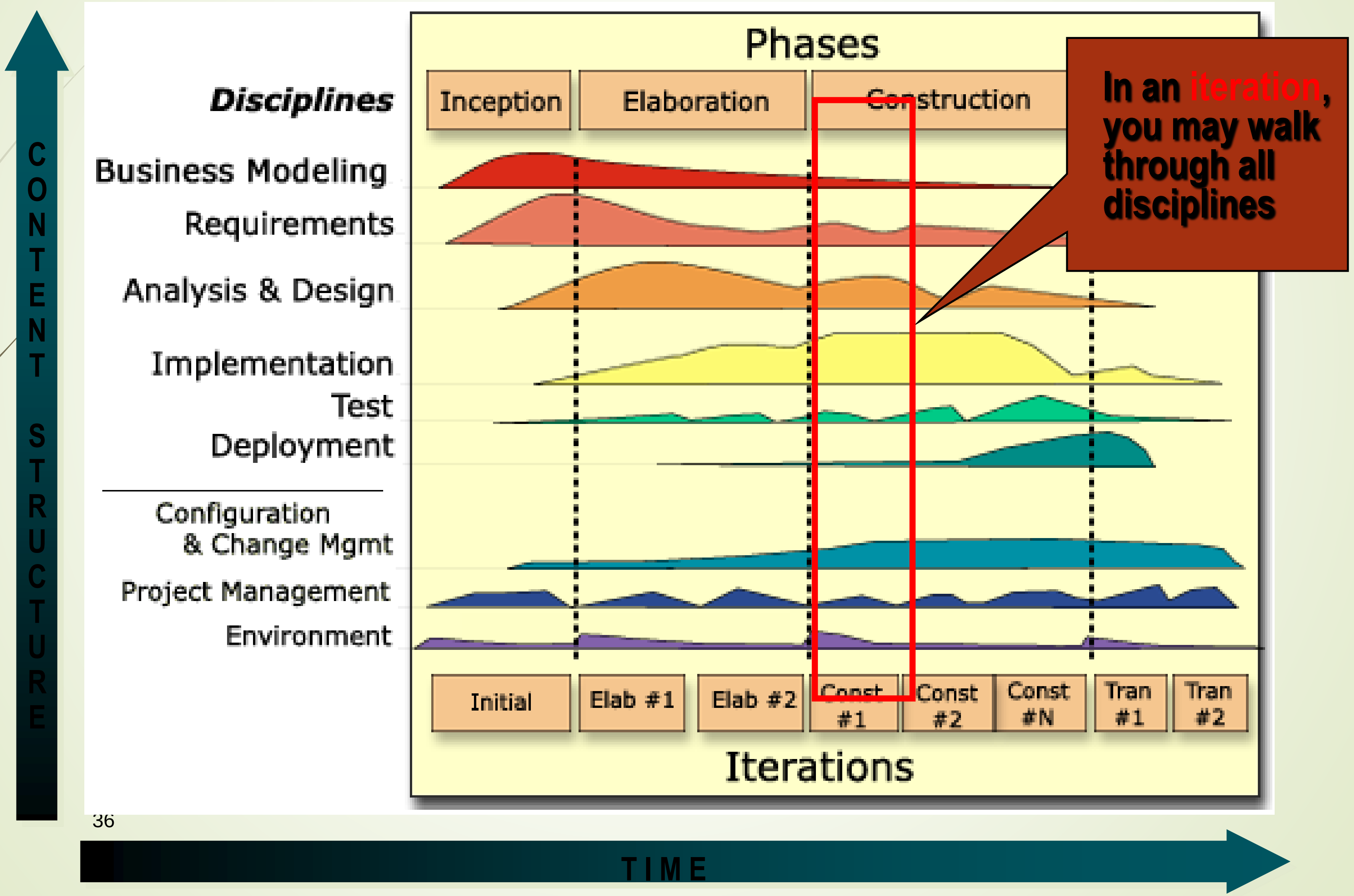


- **Earliest iterations** address **highest risks**.
- Each iteration produces an **executable release**.
- Each iteration includes **integration, test, and assessment**.
- Objective Milestones: **short-term focus; short term successes** !

Iterative Development Characteristics

- **Critical risks are resolved** before making large investments.
- **Initial iterations enable early user feedback:**
  - Easy to resolve problems early.
  - Encourages user feedback in meaningful ways
- **Testing and integration are continuous** – assures successful integration (parts all fit).
- Objective milestones provide **short-term focus**.
- **Progress** is measured by **assessing implementations.**
- **Partial implementations** can be **deployed**.
- **No big-bang approach !**

UP Lifecycle Graph – Showing Iterations



Unified Process Iterations and Phases

Executable Releases

Inception	Elaboration		Construction			Transition	
Preliminary Iteration	Architect. Iteration	Architect. Iteration	Devel. Iteration	Devel. Iteration	Devel. Iteration	Transition Iteration	Transition Iteration

An **iteration** is a distinct sequence of activities with an **established plan** and **evaluation criteria**, resulting in an ‘**executable release**.’

Problems Addressed by Iterative Development

Root Causes

- ☑ **Insufficient requirements**
- ☑ **Ambiguous communications**
- ☐ Brittle architectures
- ☑ **Overwhelming complexity**
- ☑ **Subjective assessment**
- ☑ **Undetected inconsistencies**
- ☑ **Poor testing**
- ☑ **Waterfall development**
- ☐ Uncontrolled change
- ☐ Insufficient automation

Solutions

Enables and encourages user feedback

Serious understanding encouraged in the life cycle

Development focuses on critical issues – break it down!

Objective assessment thru testing and assessment

Inconsistencies detected early

Testing starts earlier – continuous!

Risks identified and addressed early - via planned iterations!

Iterations Are Time-boxed

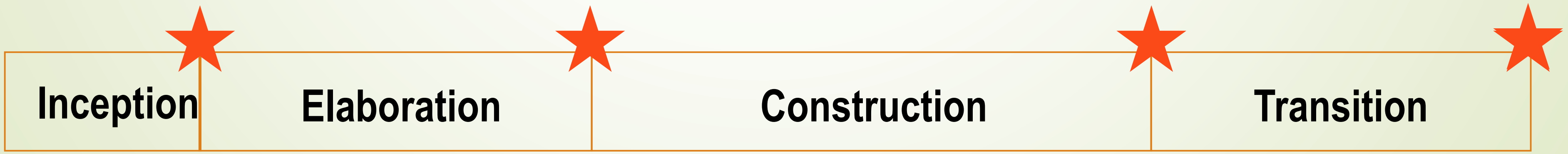
- Work is undertaken within an **iteration**.
- The iteration plan **defines** the **artifacts** to be delivered, **roles** and **activities**.
- An iteration is clearly **measurable**.
- Iterations are **risk-driven**.
- Iterations are **planned**.
- Iterations are **assessed** !
- Generally, **initial iterations** (in Construction) based on **high risk and core functionalities** !

Progress is made against MILESTONES

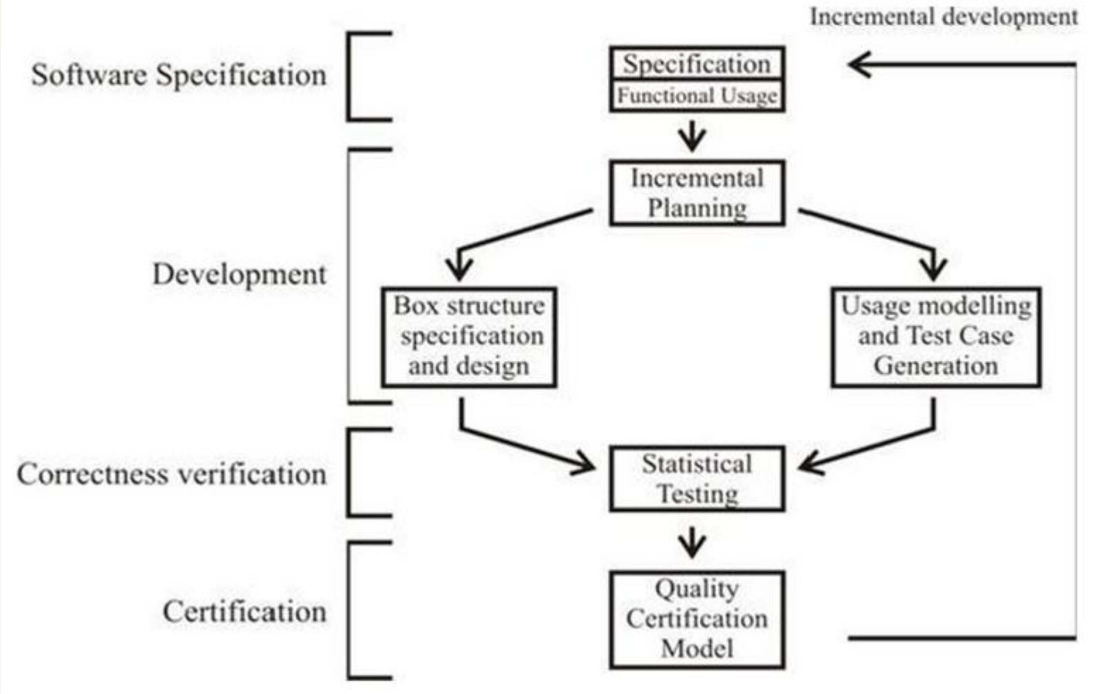
- In the **Unified Process**:
  - Each phase is defined by a **milestone**.
  - Progress is made by passing **milestones**.
  - Milestones measure **success.**

- **Phases** - NOT TIMEBOXED.
- **Iterations** **ARE TIMEBOXED**.

Major Milestones



Cleanroom Methodology



- ►Set of principles and practices for software specification, design and testing - **to improve quality, to increase productivity & to reduce cost,**
- ►Emphasis on **defect prevention** rather than defect removal,
- ►Use of **statistical Quality Assurance (QA) testing**
- ►It has another variant called – **Chinese Wall Methodology.**

Thank You !

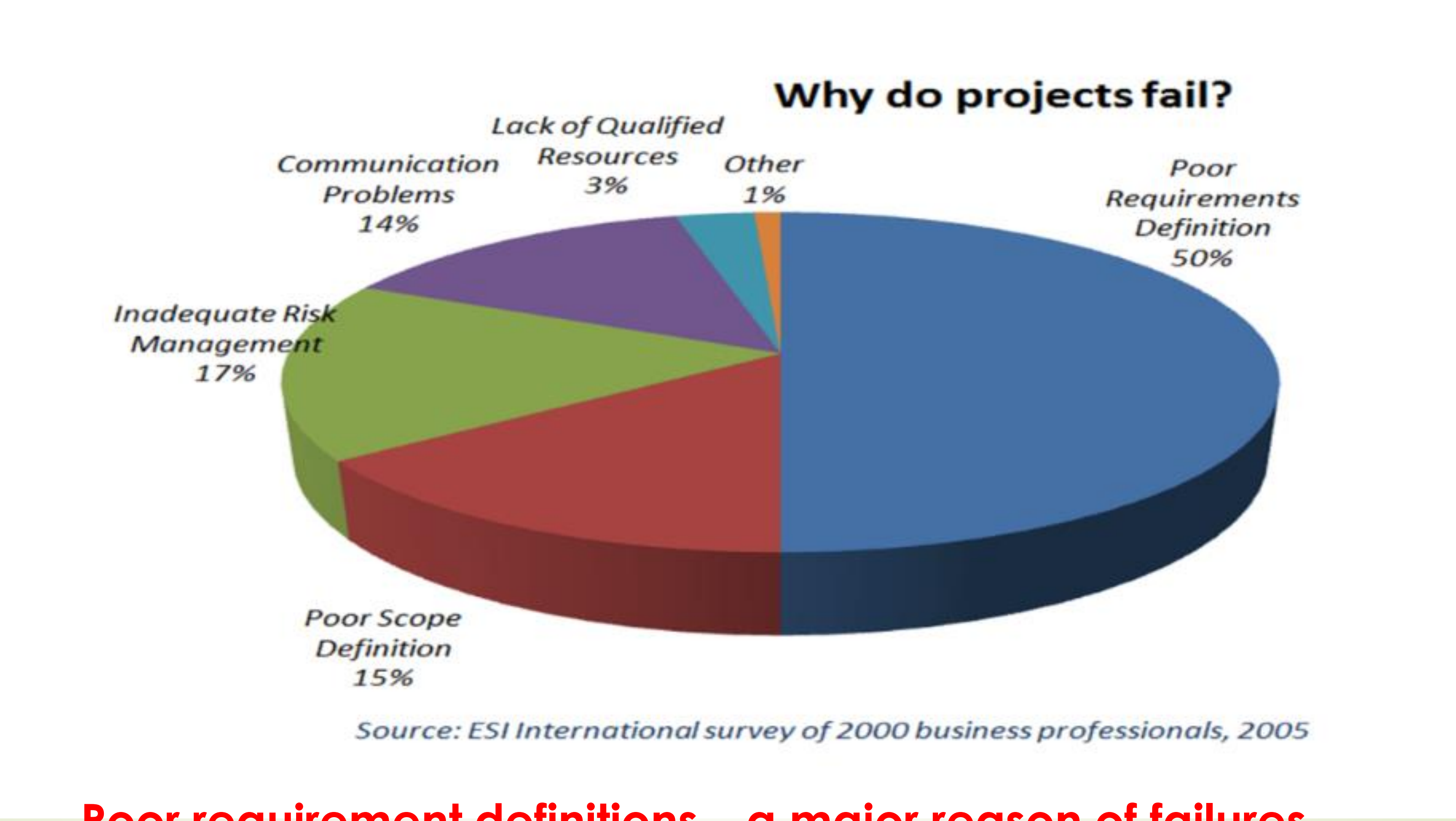
Software Requirements

Dr. Ganesh Bhutkar

Vishwakarma Institute of Technology, Pune, India

Course: Software Modelling And Design (SMAD) – CS3218

Why do Projects fail?



Poor requirement definitions – a major reason of failures

Reference: http://apppm.man.dtu.dk/index.php/File:Why_projects_fail.png

What are Software Requirements?

- The requirements for a computing system are the **description of what the system should do, the service or services that it provides and the constraints on its operation**.
- The **IEEE Standard Glossary** of Software Engineering Terminology defines a **requirement** as:
  - *A **condition or capability** needed by a user to solve a problem or achieve an objective,*
  - A **condition or capability** that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document,
  - A **documented representation** of a condition or capability as in 1 or 2.

User and System Requirements

User requirement definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20 mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

User and System Requirements

User Requirements Vs. System Requirements

User Requirements	System Requirements
▪ Written for customers . ▪ Statements in natural language. ▪ Describe the services that system provides and its operational constraints. ▪ May include diagrams or tables. ▪ Should describe functional and non-functional requirements. ▪ Should be understandable by system users who don't have detailed technical knowledge.	▪ Statements that set out detailed descriptions of the system's functions, services and operational constraints. ▪ Defines what should be implemented so may be part of a contract between client and contractor. ▪ Intended to be a basis for designing the system. ▪ Can be illustrated using system models.
We provide a definition for a user requirement.	**We provide a <u>specification</u> for a system requirement.**

Functional and Non-Functional Requirements

- **Functional Requirements**
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.

- **Non-Functional Requirements**
  - Constraints on the services or functions offered by the system such as timing constraints, constraints due to quality guidelines, standards or legal aspects.
  - Often apply to the system as a whole rather than individual features or services.

- **Domain Requirements**
  - Constraints on the system from the domain of operation

Functional Requirements

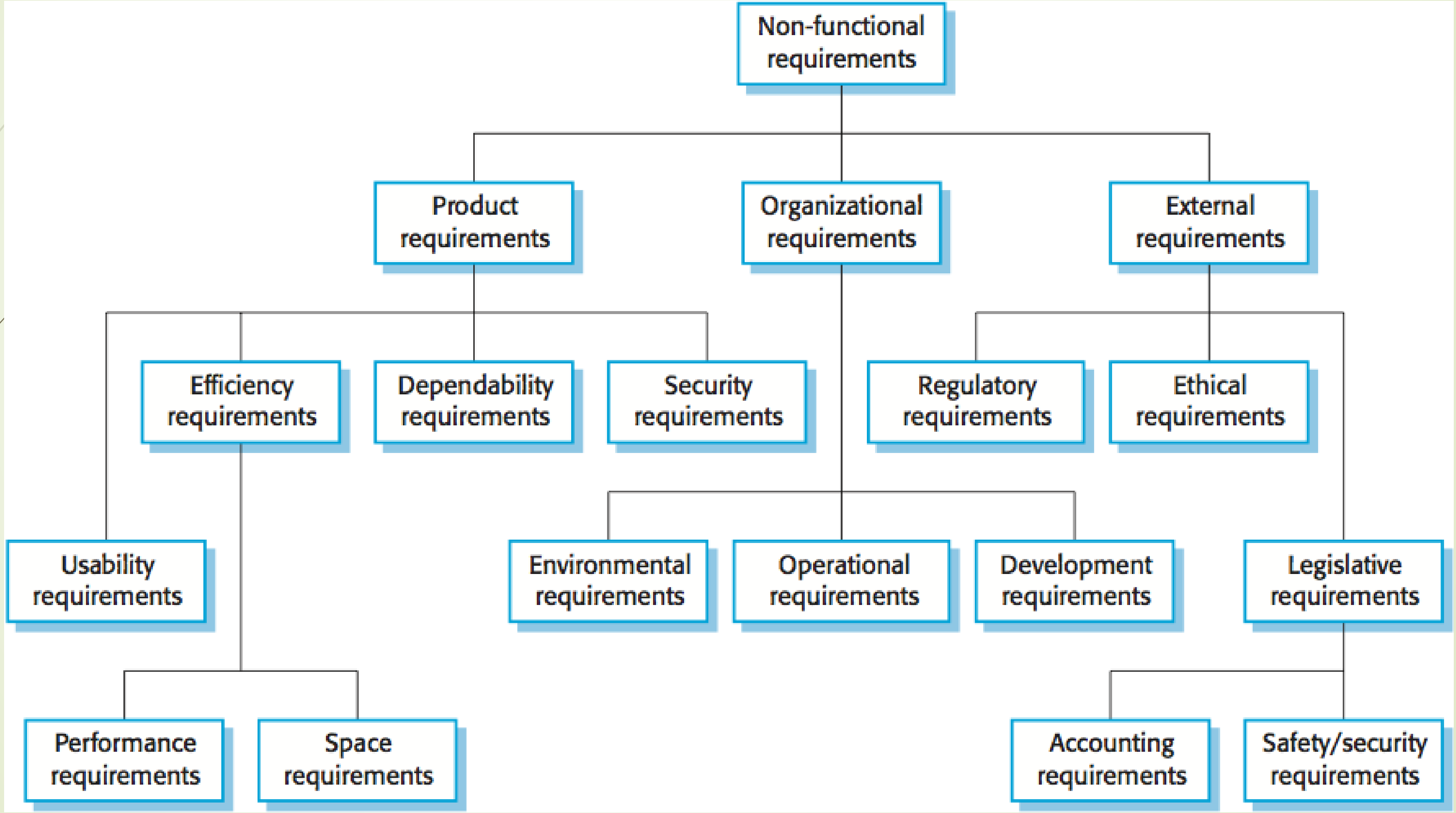
- Describe **functionality or system services**.
- Depend on the type of software, expected users and the type of system where the software is used.
- **Functional user requirements** may be **high-level statements of what the system should do**.
- **Functional system requirements** should describe **the system services in detail**.
- Essentially, **these are the 'whats' of the system** that we often refer to. These are not 'all that there is,' but these should describe the overall functionality of the system.

Functional Requirement No.	Function Requirement Description
FR 1	User should be able to enter Sales Data
FR 2	Sales Reports should be generated every 24 hours
FR 3	API interface to Invoice System

Non-Functional Requirements

- They define **system properties and constraints.**
- e.g. **Reliability**, **Response Time**, **Maintainability**, **Usability**, **Scalability**, **Portability and storage requirements**.
- Constraints are **I/O device capability**, **system representations** etc.
- Process requirements may also be specified mandating a particular **IDE**, **programming language or development method**.
- Non-functional requirements may be **more critical** than functional requirements. If these are not met, the system may be useless.

Types of Non-Functional Requirements



Metrics for specifying Non- Functional Requirements

Property	Measure
Speed	Processed transactions / Seconds User / Event response time Screen refresh time
Size	MBytes / GByte Number of ROM chips
Usability	Time to learn Speed of performance Rate of errors / failures Retention overtime User satisfaction
Reliability	Mean Time To Failure (MTTF) Probability of unavailability Rate of failure occurrence Availability
Robustness	Mean Time To Restart (after failure…) (MTTR) Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Domain Requirements - Examples

- **Train Control System**
  - It should consider the **braking mechanism / characteristics in different weather conditions**.
- **Restaurant Ordering System**
  - Order will be placed through a **touch screen at each table**.
  - **Menus** should be **in multiple languages**. e. g. English, Marathi & Gujrathi.
  - **Wireless communication** is required with kitchen.
- **Banking System**
  - Most banks do not allow over draw on most accounts. Only few accounts like **current account has over draw or over draft facility**.

Domain Requirements Problems

- **Understandability**
  - Requirements are expressed in the **language / terms of the application domain**.
  - This is often **not understood by software engineers** developing the system.
- **Implicitness**
  - **Domain specialists** understand the area so well that they **do not think of making the domain requirements explicit**.
  - And this is often a major **problem in communications**.

Requirements Engineering (RE)

- It is the process of **defining**, **documenting and maintaining the requirements**.
- **Phases in RE**
  - ***(1) Requirements Elicitation:*** *It is a* process through which the customers, buyers, or users of a software system **discover, reveal, articulate, and understand their requirements.**
  - ***(2) Requirements Analysis and Negotiation:*** It is a process of **examining** the requirements that have been elicited; **prioritizing** the elicited requirements & **Classify** the requirements.
  - ***(3) Requirements Specification:*** *It is a* process of recording the requirements in one or more forms, including natural language. It leads to **formal, symbolic or graphical representation(s) with necessary qualitative & quantitative details** of the requirements.

Requirements Analysis

- It is the process of **determining user expectations for a new or modified product / project**.
- These features, called requirements, must be **quantifiable, relevant and detailed**.
- Requirements are often called **functional specifications** in Software Engineering.
- Requirements analysis is an **important aspect of project management**.
- For Analyzing Requirements take the following steps:
  - Make Requirements **S.M.A.R.T.**
  - **MoSCoW** the Requirements
  - **Classify** the Requirements

Make them S.M.A.R.T.

Specific Requirements

- Without ambiguity, using consistent terminology, simple and at the appropriate level of detail.
- Let’s consider this requirement: ***The new system shall be able to manage project schedule.***
- Is the requirement specific? The Answer is NO.
- What is meant by the verb Manage? Does it mean **Create – Retrieve – Update – Delete or something more**?

Measurable Requirements

- Is it possible to put a number to the requirement?
- This is especially true for non-functional requirements.
- Let's consider the requirement: ***The system shall have great usability.***
- How do we measure great usability?
- One way to measure is through the **User Satisfaction, Success Rate or Completion Rate.**

Achievable Requirements

- By an achievable requirement means that it could be **reasonably accomplished under the given conditions and within the bounds of human knowledge.**
- For example: ***'The system shall be 95% reliable and 99% available'.*** OR ***'The system shall have a minimum response to a query of 1 second irrespective of system load'***.
- The consequence of attempting to meet these requirements is that the system may be **accepted**, based on measurable performance.

Relevant Requirements

- It means the requirements are **realistic, given the resources**.
- Aspects to consider include:
  - Do we have the required **staffing**?
  - Do we have the **skills**?
  - Do we have access to the **development infrastructure needed**?
  - Do we have **enough time** in hand to implement the requirement?
- Let's consider this requirement: ***Let's implement SAP ERP Application in next 1 month.***
- Any ERP project takes significant amount of planning and preparation. **There would not have been any ERP implementation which is less than 3 months duration**.

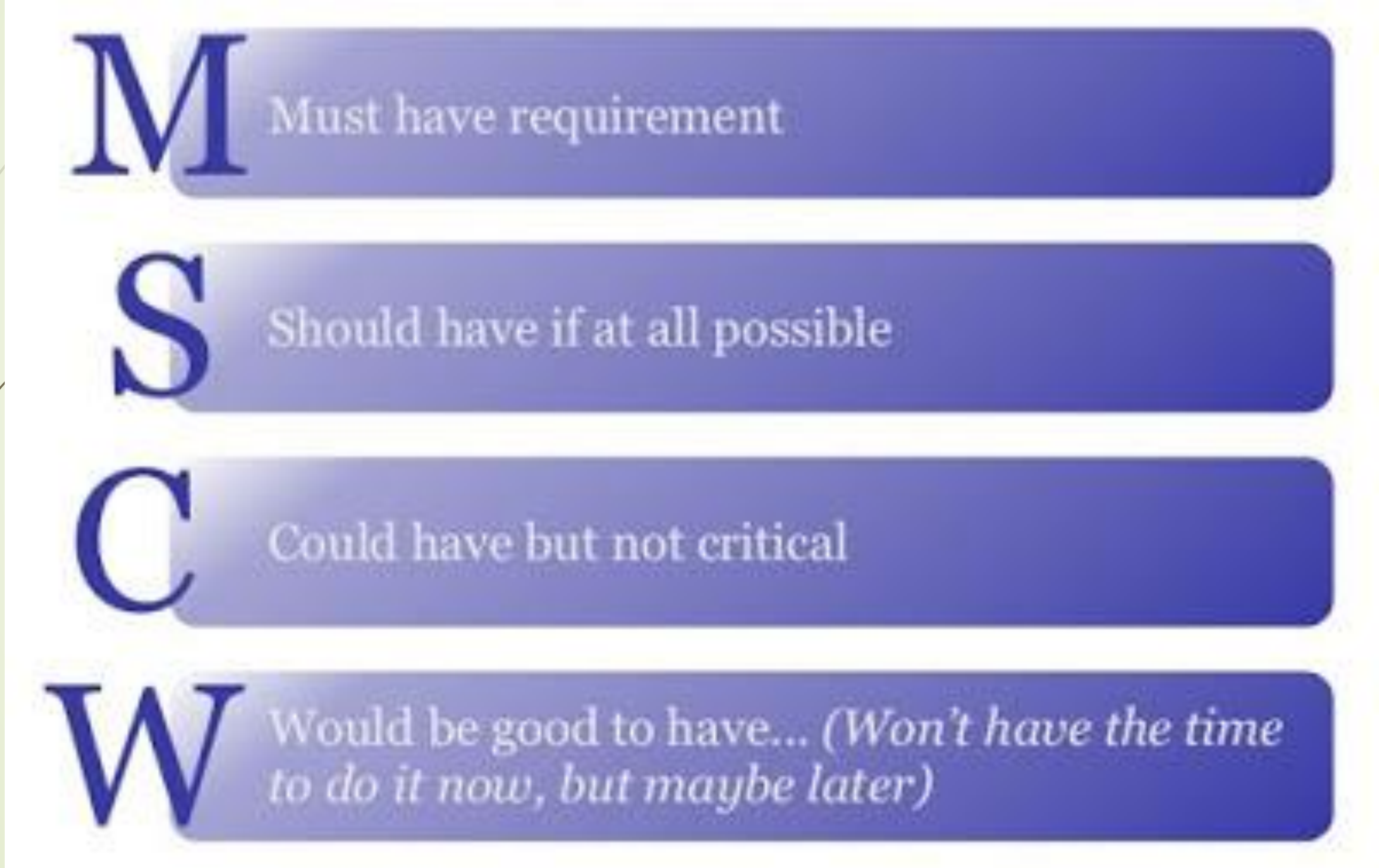
Time-oriented Requirements

- Answers the question, **'When will it be done?'**
- Sometimes a **task** / **phase** may only have an **end point or due date**.
- Sometimes that end point or due date is the actual end of the task, or sometimes the end point of one task is the start point of another.
- Sometimes, a **task has several milestones or check points** to help you or others assess how well something is going before it is finished so that corrections or modifications can be made as needed to make sure the end result meets expectations.

MoSCoW the Requirements

- MoSCoW is a method for **prioritizing requirements**.
- The requirements are prioritized to **prevent them from becoming to expensive or unrealistic**.
- The main goal is to come up with requirements that add the most **value for the customers / users and their organization**.
- The project requirements are divided into one of the following categories:
  - **Must-Haves**
  - **Should-Haves**
  - **Could-Haves**
  - **Would-Haves**

MoSCoW the Requirements (Contd…)



M – Must haves

- They are a **necessity for a workable product** and there is **no alternative**.
- Without meeting these requirements, the project **won't** be use-able.
- **Examples:**
- *The **HR system** "must" **store employee attendance record**.*

S – Should haves

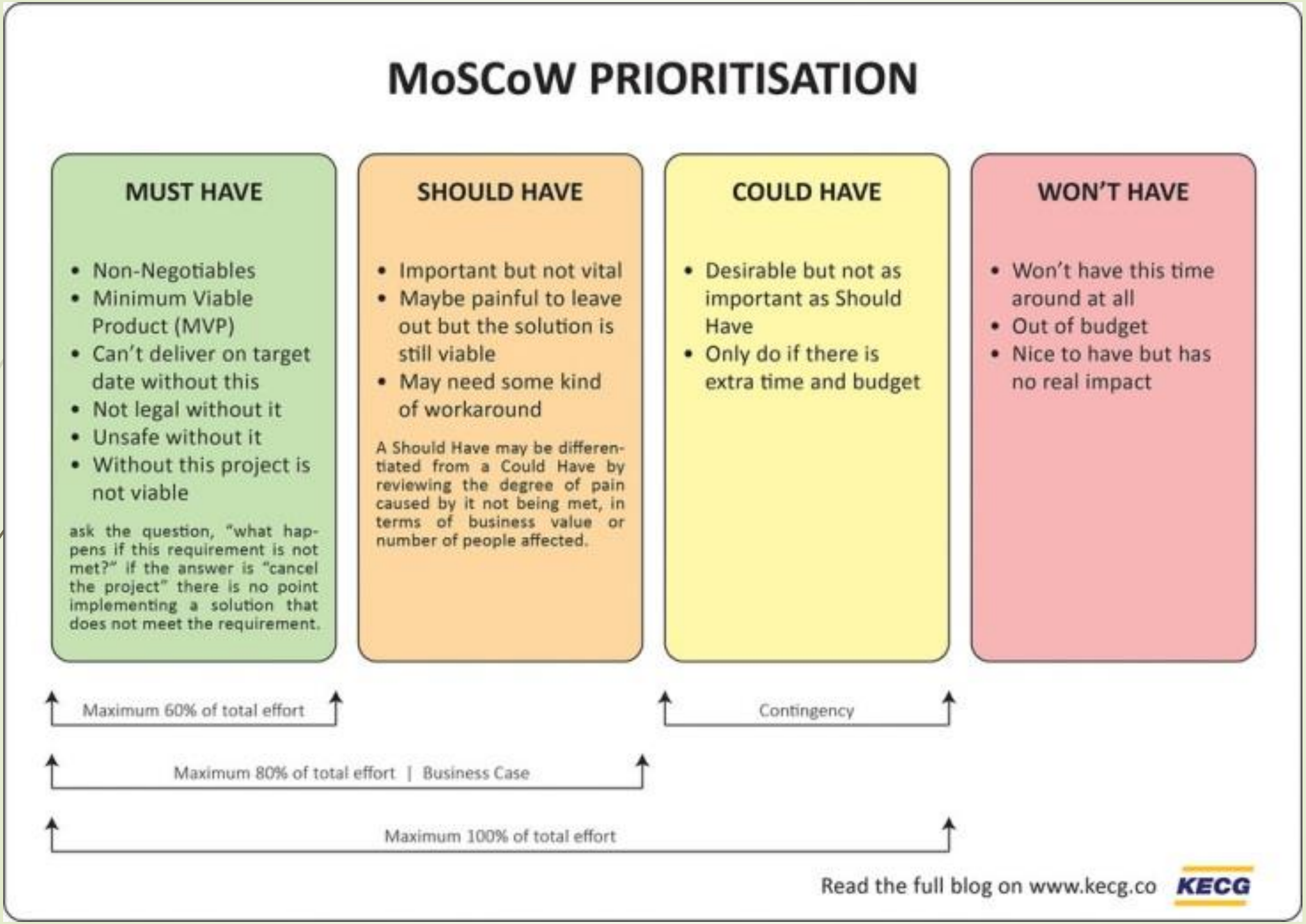
- These are **additional and much desired requirements** that have a high priority, but are **not essential** for a usable end product.
- When they are met, they will only **add to the value of the product**.
- Depending on the available time, you can always return to these requirements at a later time.
- **Examples:**
- *The **HR system** "should" **allow printing of employee attendance record**.*

C – Could haves

- The ‘Could haves’ have a **lower priority** than the ‘Should haves’.
- This option will **only be included if there really is more than enough time** to make it work.
- This category is also referred to as **‘nice to have’**; they’re more a wish than an absolute requirement.
- **Examples:**
- *The **HR system** “could” send out **notifications on pending leave dates**.*

W – Won’t haves (and would haves)

- These are about **wishes for the future** that are often impossible to realize or cost a lot of time.
- ‘Would haves’ are often followed upon at a later stage after the initial project is finished.
- **Examples:**
- *The **HR system “won’t” support remote access** but may do so in the next release.*


MoSCoW PRIORITISATION
MUST HAVE
• Non-Negotiables
• Minimum Viable Product (MVP)
• Can't deliver on target date without this
• Not legal without it
• Unsafe without it
• Without this project is not viable
ask the question, "what happens if this requirement is not met?" if the answer is "cancel the project" there is no point implementing a solution that does not meet the requirement.
SHOULD HAVE
• Important but not vital
• Maybe painful to leave out but the solution is still viable
• May need some kind of workaround
A Should Have may be differentiated from a Could Have by reviewing the degree of pain caused by it not being met, in terms of business value or number of people affected.
COULD HAVE
• Desirable but not as important as Should Have
• Only do if there is extra time and budget
WON'T HAVE
• Won't have this time around at all
• Out of budget
• Nice to have but has no real impact
Maximum 60% of total effort
Contingency
Maximum 80% of total effort | Business Case
Maximum 100% of total effort
Read the full blog on www.kecg.co
KECG

Classify the Requirements

- The requirements are gathered in the ***Requirements Elicitation Phase***
- The requirements are analyzed in the ***Requirements Analysis and Negotiation Phase:***
  - Examined against **SMART Objectives**
  - Prioritized by **MoSCoW**
  - Then, are classified into following categories:
    - **Functional Requirements**
    - **Non-functional requirements**
    - **Domain requirements**

Thank You !

Agile Development Methodology

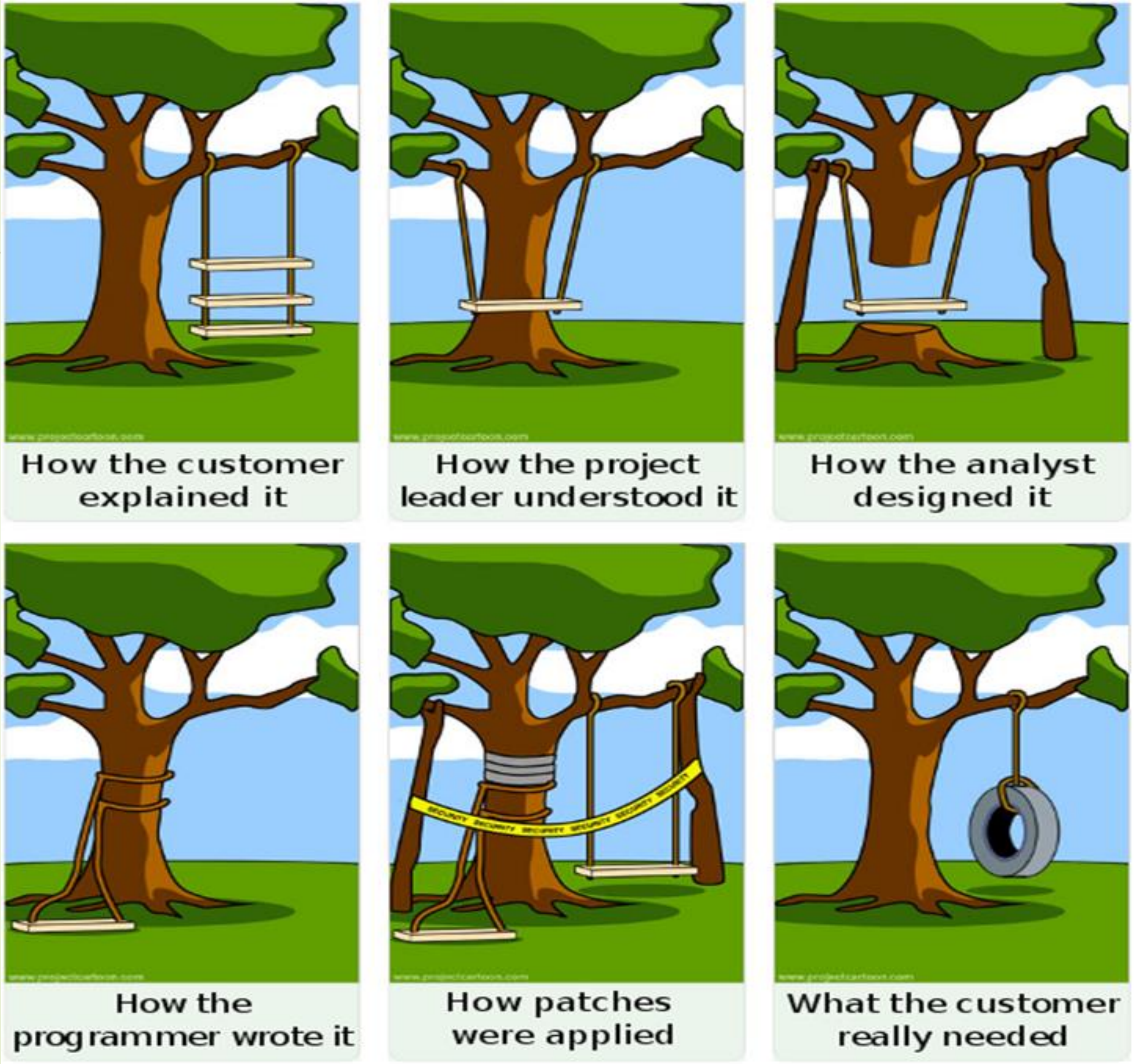
Dr. Ganesh Bhutkar

Vishwakarma Institute of Technology, Pune, India

Course: Software Modelling And Design (SMAD) – CS3218

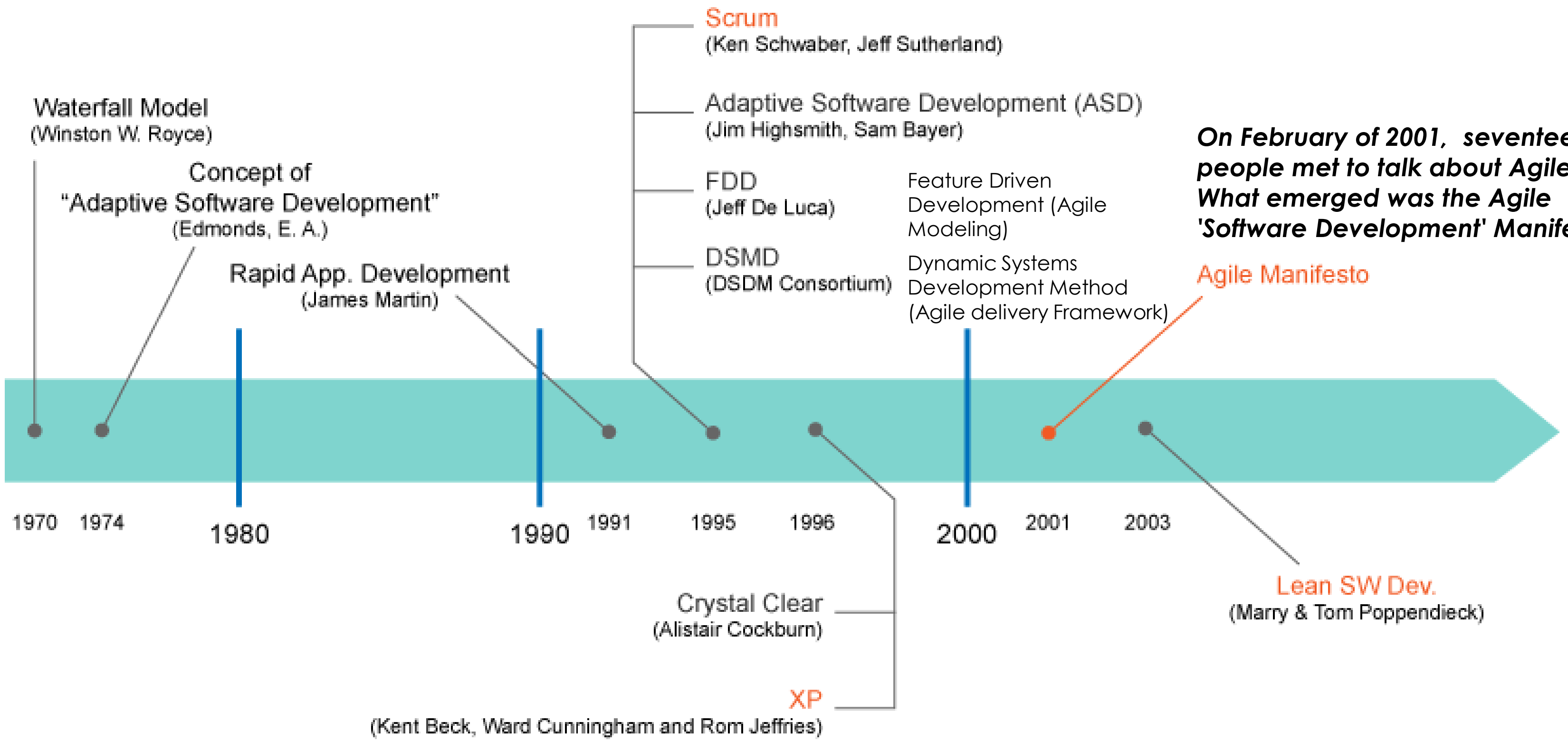
Common Fears of Developers

- The project will produce the **wrong / undesirable product**.
- The project will produce a **product of inferior quality**.
- The project will be **late on time**.
- We'll have to work **80 hour weeks**.
- We'll have to **break commitments**.
- We **won't be having fun** (Tough time).


How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
How patches were applied
What the customer really needed

Agile History Timeline

On February of 2001, seventeen people met to talk about Agile. What emerged was the Agile 'Software Development' Manifesto



Hirotaka Takeuchi 1986 ⟷ **1993 Jeff Sutherland, Jeff McKenna, John Scumniotales**

Scumniotales

What Is Agile?

- **Agile:** **Readiness for motion, nimbleness, activity or dexterity**

- **Agility:** **Ability to both create and respond to change in order to make a profit** in a turbulent / dynamic business environment

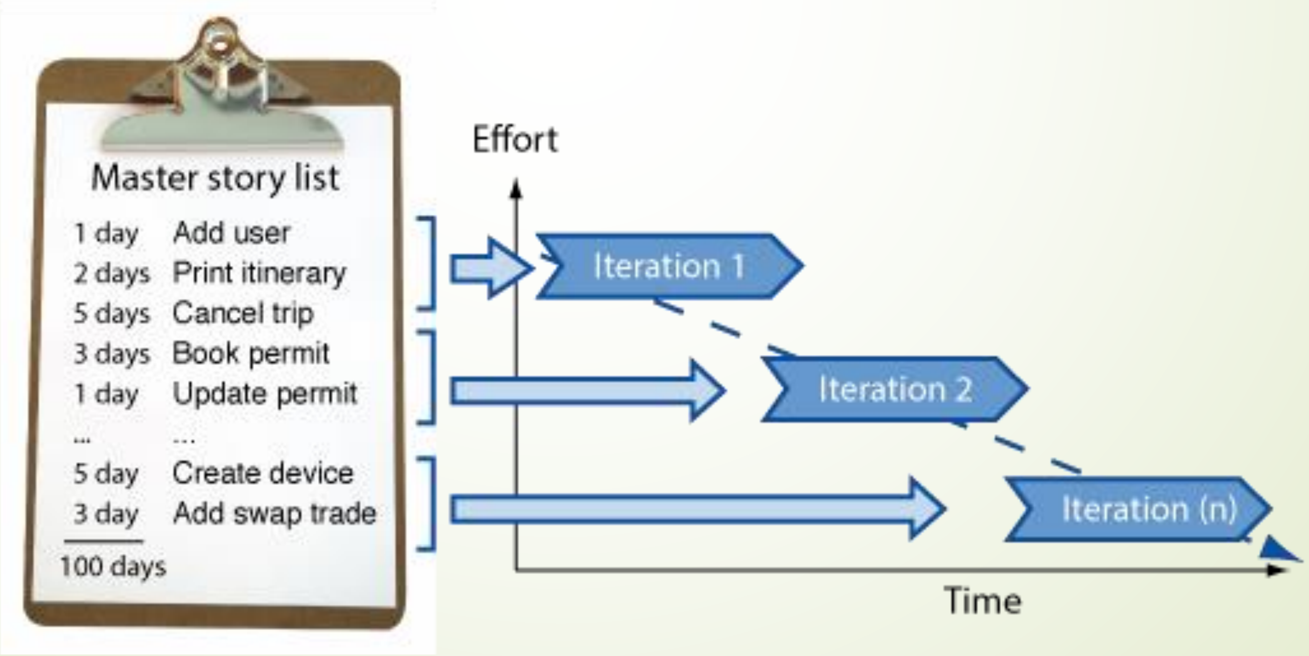
- **Companies / Organizations** need to determine the **amount of agility they need, to be competitive**.

Evolution of Project Management

Simple Comparison	
Traditional Project Management	**Modern Project Management (Agile)**
Elements impacting execution of a project: a) Planning b) Control	**Elements impacting execution of a project**: a) Competitive environment b) Creativity and Innovation c) Planning d) Control
Project Success is measured by having: a) Well defined requirements b) Approved budget c) On-Time Delivery	**Project Success is measured by**: a) Delivery of Business Value (what does customer want) Less important are: a) Having well defined requirements b) Budget constraints c) Set Schedule Today's world has a much higher level of uncertainty which makes it difficult to always start a project with well-defined requirements
Requires a Project Manager	**Project Manager Role is distributed among**: a) Product Owner b) Scrum Master c) Project Team

Understanding Agile

- Agile is a **time-boxed and iterative approach of software development that builds software incrementally** from the start of the project, instead of trying to deliver it all at once near the end.
- It works by **breaking projects down into little bits of user functionality called *User Stories***, prioritizing them, and then continuously delivering them in **short two week cycles called *Iterations*.**



Agile Alliance

- Several individuals, **The Agile Alliance**,
  - motivated to constrain activities
  - such that certain outputs and artifacts are **predictably** produced.
  - Around 2000, these notables got together to address common development problems.

- Goal: outline values and principles to allow software teams to
  - **develop quickly** and
  - **respond to change.**

Agile Alliance (Conti…)

- These activities arose in large part to **runaway processes**.
  - Failure to achieve certain goals was met with 'more process.' **Schedules slipped; budgets bloated, and processes became even larger**.
- The Agile Alliance created a statement of **values**: termed the **manifesto** of the Agile Alliance (2001).
- They then developed the **12 Principles of Agility.**

Agile Manifesto (Principles of Agility)

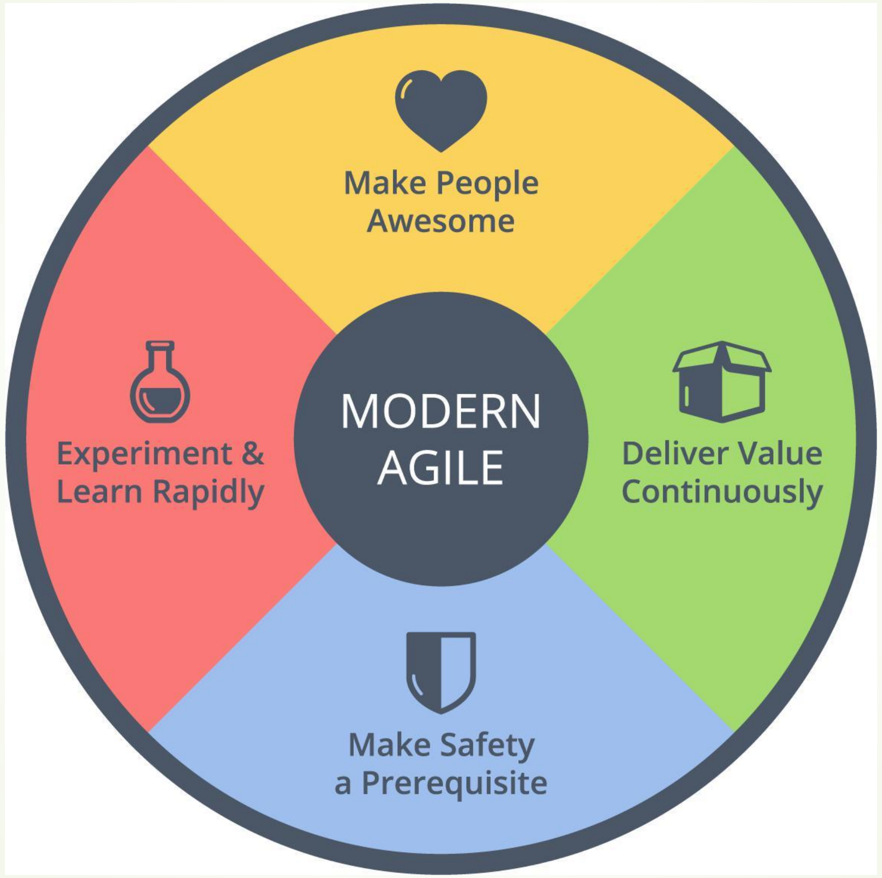
1. Customer Satisfaction through continuous delivery
2. Welcome changing requirements
3. Deliver working software frequently
4. Collaboration between business members and developers
5. Build projects around motivated individuals
6. Efficiency through face-to-face conversation
7. Working software is the primary measure of progress
8. Agile processes promote sustainable development
9. Technical excellence and good design enhances agility
10. Promote Simplicity
11. Self-organizing teams
12. Self-improving team philosophy

United Nations (UN)

Agile Guiding Principles

Experiment & Learn Rapidly
Is a guiding principle of Modern Agile because it protects us from wasting time and helps us discover success faster

Make Safety a Prerequisite
Means establishing safety before engaging in potentially hazardous work



Make People Awesome
Amazon has made Customer Obsession a guiding principle since 1997 and it shows. If you make customers awesome, they tend to be natural promoters of your products or services

Deliver Value Continuously
Anything valuable that hasn't been delivered isn't helping anyone. How might we deliver the right outcomes faster

Manifesto for Agile Software Development

- We are uncovering better ways of developing software by doing it and helping others do it.

- Through this work, we have come to the value:

1. **Individuals and interactions over processes and tools**
2. **Working software over comprehensive documentation**
3. **Customer collaboration over contract negotiation**
4. **Responding to change over following a plan**

Value 1: Individuals and Interactions over Processes and Tools

- **Strong players:** a must, but can fail if don't work together.
- **Strong player:** not necessarily an 'ace;' work well with others!
  - Communication and interacting is **more important** than raw talent.
- **'Right' tools** are vital to smooth functioning of a team.
- **Start small**. Find a free tool and use until you can demo you've outgrown it. Don't assume bigger is better. Start with white board; flat files before going to a huge database.
- **Building a team** more important than **building environment.**
  - Some managers build the environment and expect the team to fall together. Doesn't work.
  - Let the team build the environment on the **basis of need**.

Value 2: Working Software over Comprehensive Documentation

- **Code:** not ideal medium for communicating rationale and system structure.
  - Team needs to produce human readable documents describing system and design decision rationale.
- **Too much documentation is worse than too little**.
  - Take time; more to keep in sync with code.
- **Short rationale and structure document**.
  - Keep this in sync; Only highest level structure in the system kept.

Value 2: Working Software over Comprehensive Documentation (Conti...)

- **How to train newbees** if short & sweet?
  - Work closely with them.
  - Transfer knowledge by sitting with them; make part of team via close training and interaction
- **Two essentials** for transferring info to new team members:
  - **Code** is the only unambiguous source of information.
  - **Team** holds every-changing roadmap of systems in their heads; cannot put on paper.
  - **Best way** to transfer info - **interact with them**.
- **Fatal flaw**: Pursue documentation instead of software
- **Rule**: Produce no document unless need is immediate and significant.

Value 3: Customer Collaboration over Contract Negotiation

- **Not possible to describe software requirements up front** and leave someone else to develop it within cost and on time.
- **Customers cannot just** cite needs and **go away**.
- Successful projects require **customer feedback on a regular and frequent basis** – and not dependent upon a contract or SOW.

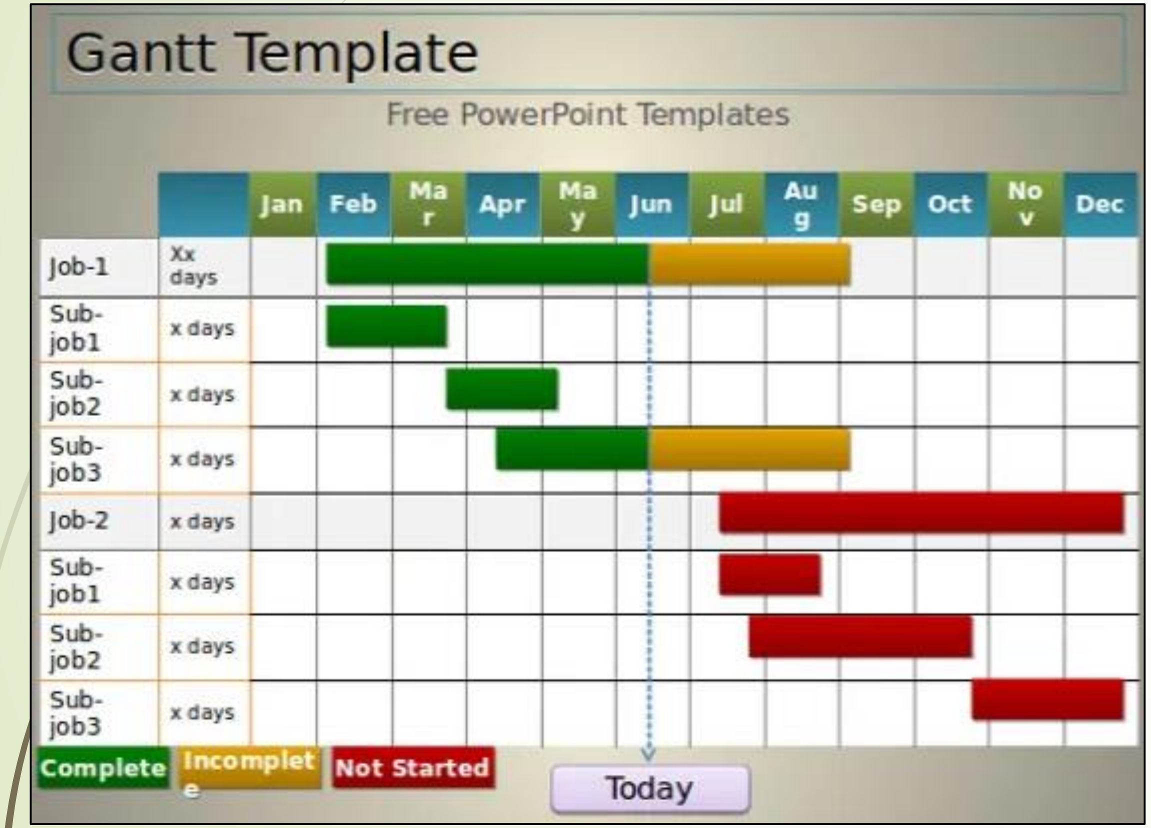
Value 3: Customer Collaboration over Contract Negotiation (Conti…)

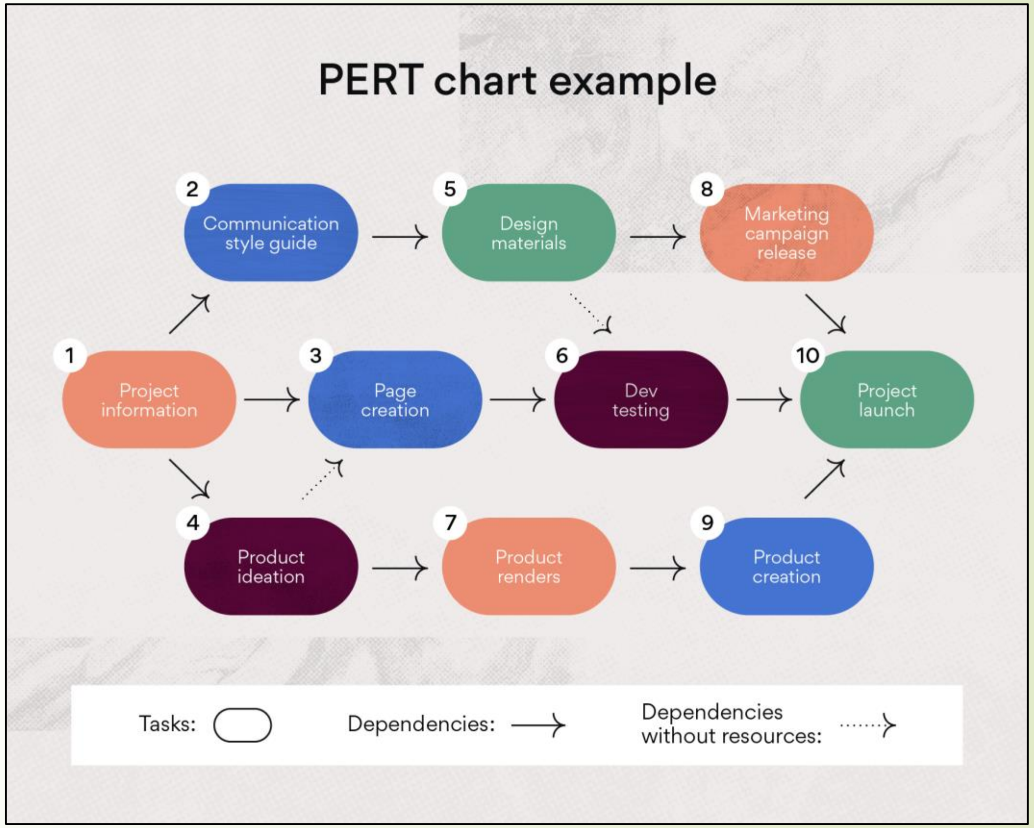
- **Best contracts are NOT** those **specifying requirements, schedule and cost.**
  - Become meaningless shortly.
- Far better are **contracts that govern the way the development team and customer will work together**.
- Key is intense collaboration with customer and a contract that governed **collaboration rather than details** of scope and schedule
  - Details ideally not specified in contract.
  - With frequent deliverables and feedback, acceptance tests never an issue.

Value 4: Responding to Change over Following a Plan

- **Our plans and the ability to respond to changes is critical !**
- **Course of a project cannot be predicted far into the future.**
  - **Too many variables; not many good ways at estimating cost.**
- Tempting to create a **PERT / CPM or Gantt chart** for whole project.
  - This does Not give novice managers control.
  - Can track individual tasks, compare to actual dates / planned dates and react to discrepancies.
  - The structure of the chart will degrades…
  - **As developers gain knowledge of the system and as customer gains knowledge about their needs, some tasks will become unnecessary; others will be discovered and will be added to 'the list.'**
  - **In short, the plan will undergo changes in *shape*, not just dates.**

Gantt & PERT Charts





Value 4: Responding to Change over Following a Plan (Conti…)

- **Better planning strategy** - **make detailed plans for the next few weeks, very rough plans for the next few months, and extremely crude plans beyond that.**
- Need to know what we will be working on the next few weeks; roughly for the next few months; a vague idea what system will do after a year.
- **Only invest in a detailed plan for immediate tasks**; once plan is made, difficult to change due to momentum and commitment.
  - But **rest of plan remains flexible**. The lower resolution parts of the plan can be changed with relative ease.

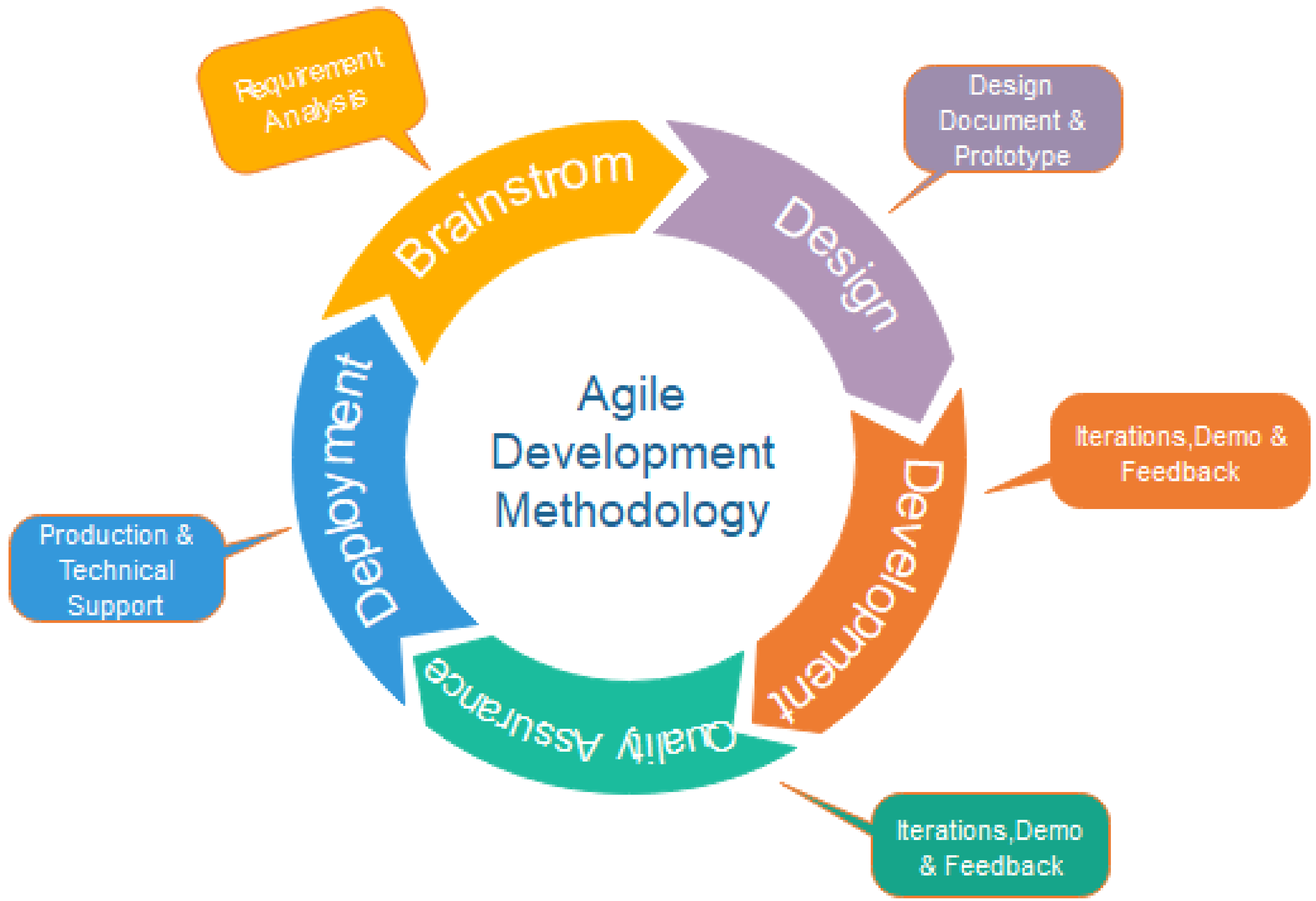


Fig. Agile Model

Problems with Agile Methodology

- It can be **difficult to keep the interest of customers / users** who are involved in the process.
- **Team members may be unsuited to the intense involvement** that characterizes agile methods.
- **Prioritizing changes can be difficult** where there are multiple stakeholders.
- Maintaining **simplicity requires extra work**.
- Because of their focus on small, tightly-integrated teams, there are **problems in scaling agile methods to large systems**.
- **Less emphasis on documentation - harder to maintain** when you get a new team for maintenance.

Thank You !

The Scrum

Dr. Ganesh Bhutkar

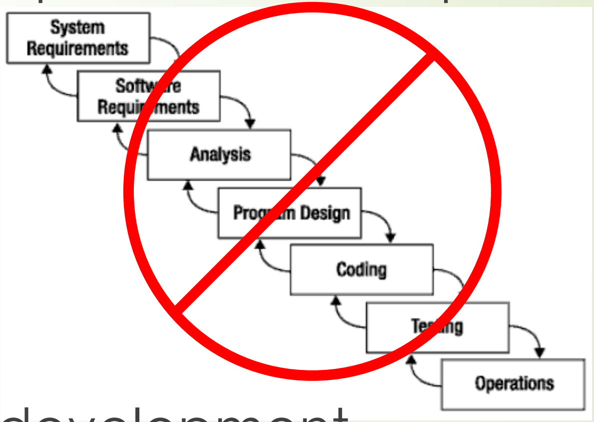
Vishwakarma Institute of Technology, Pune, India

Course: Software Modelling And Design (SMAD) – CS3218

What is Scrum?

Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems.

- Is an agile, lightweight process
- Can manage and control software and product development
- Uses iterative & incremental practices
- Reduces time to benefits

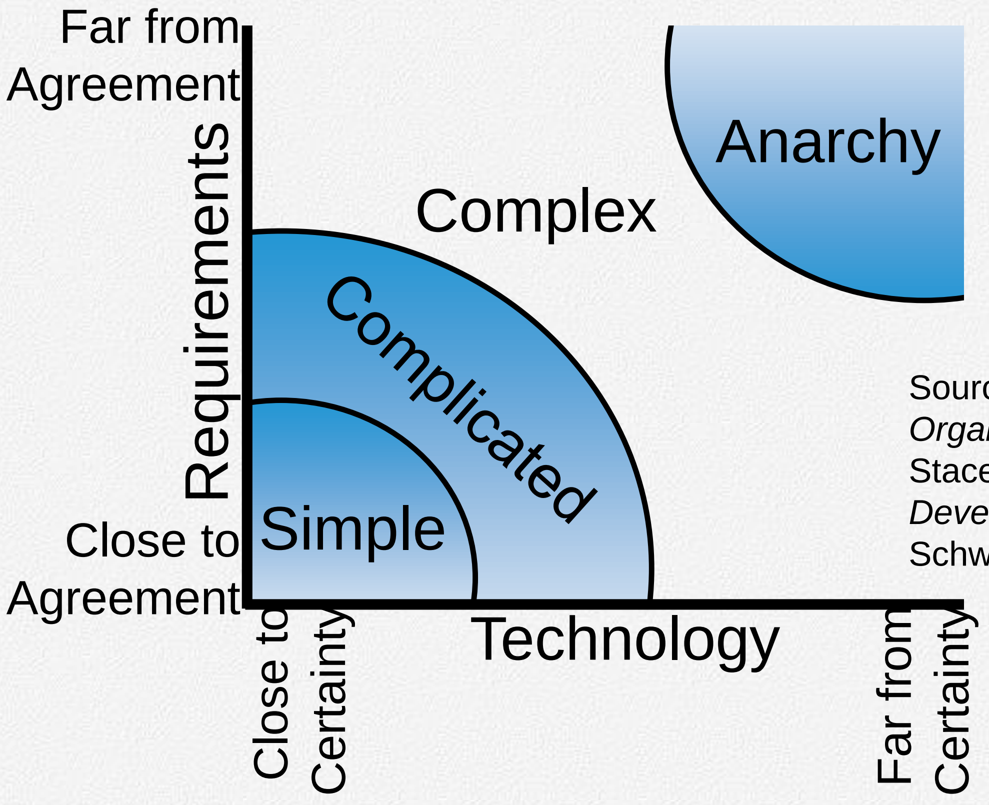


- Embraces adaptive, empirical systems development
- Embraces the opposite of the waterfall approach…

Scrum Origins

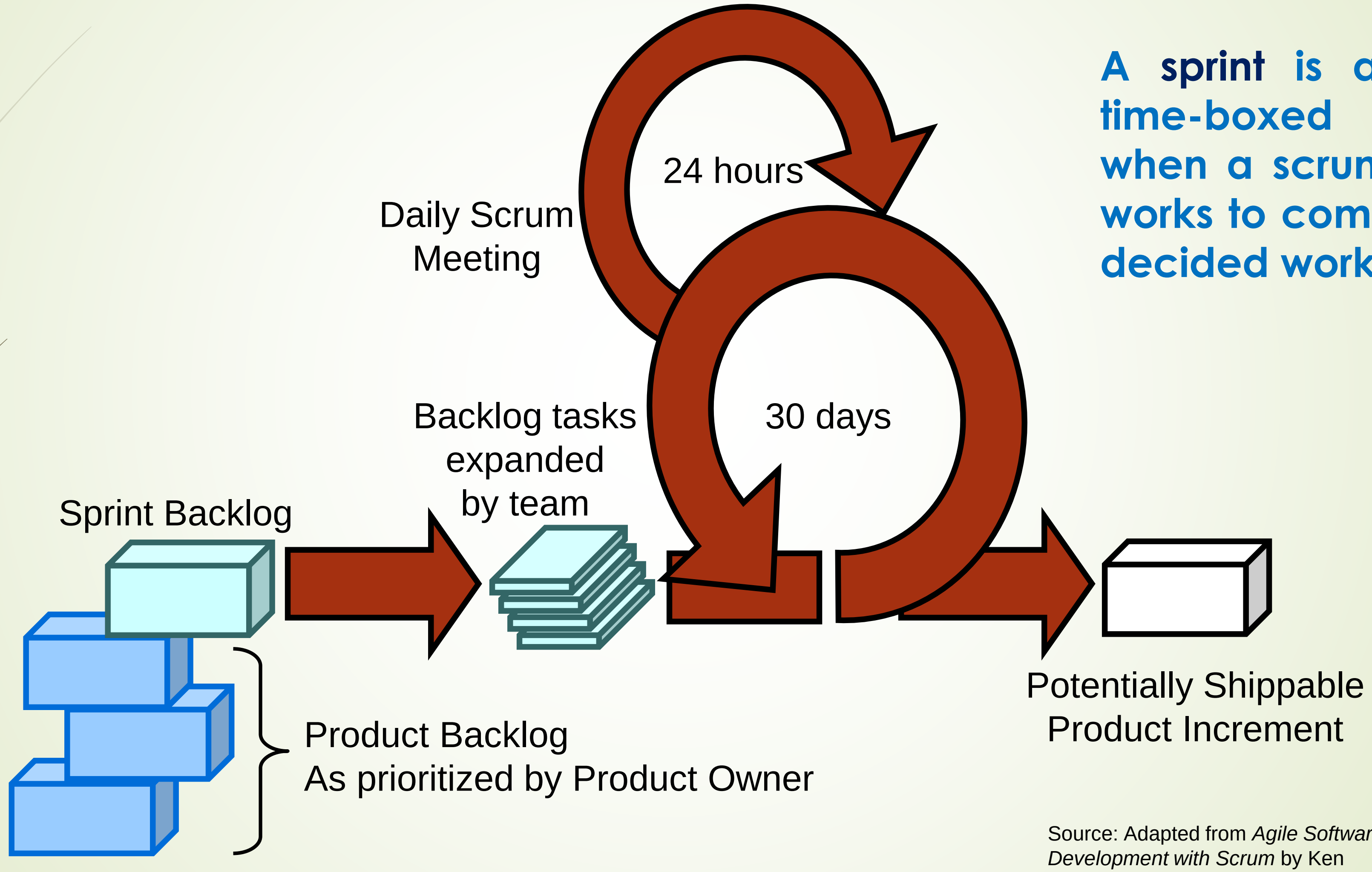
- **Jeff Sutherland**
  - Initial scrums at Easel Corp in **1993**
  - IDX and 500+ people doing Scrum
- **Ken Schwaber**
  - Scrum presented at OOPSLA 96 with Sutherland
  - Author of three books on Scrum
- **Mike Beedle**
  - Scrum patterns in PLOPD4
- **Ken Schwaber and Mike Cohn**
  - Co-founded **Scrum Alliance in 2002**, initially within Agile Alliance

Project Noise Level



Source: *Strategic Management and Organizational Dynamics* by Ralph Stacey in *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle.

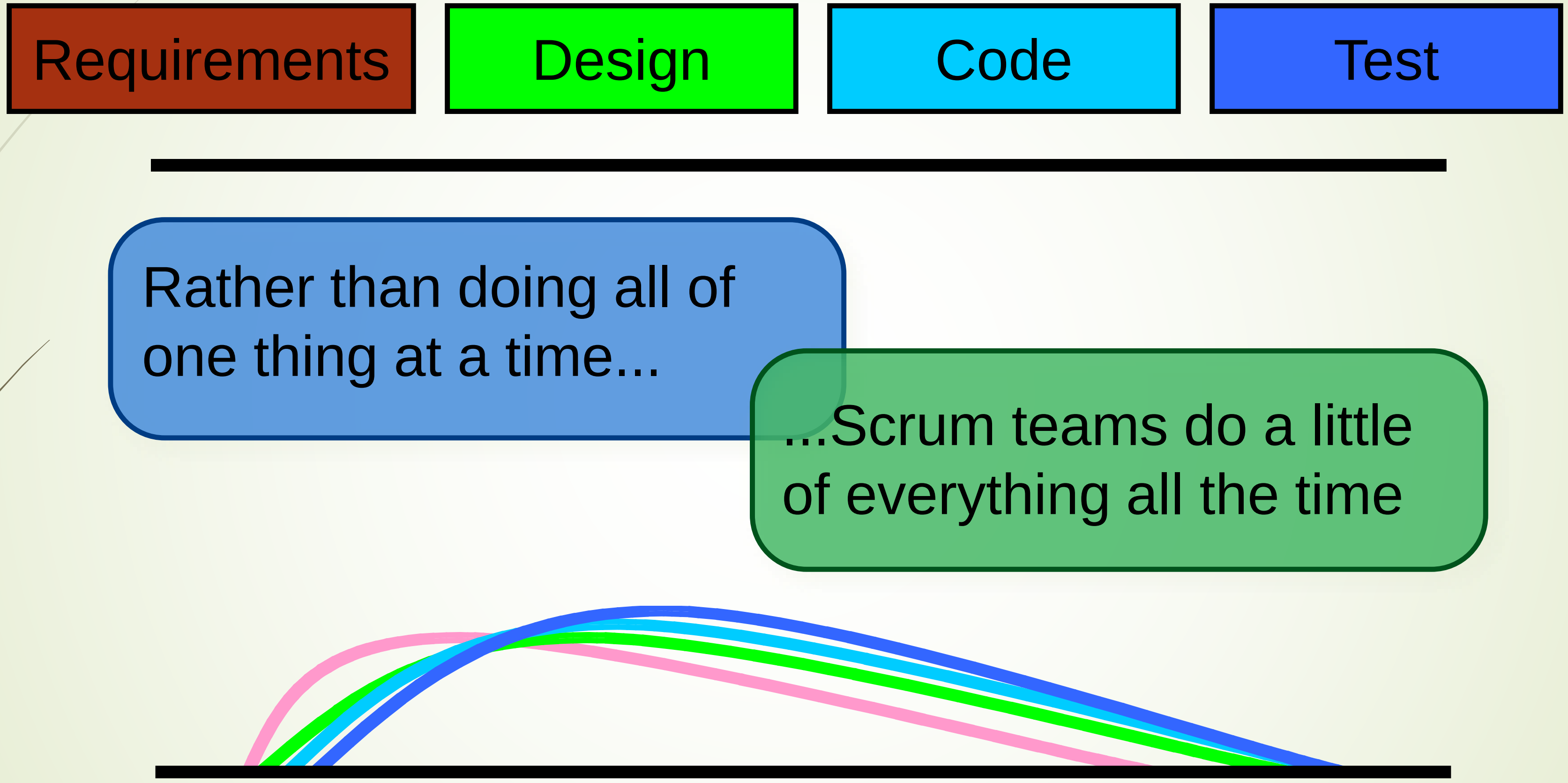
Scrum at a Glance



A **sprint** is a short, time-boxed period when a scrum team works to complete a decided work.

Source: Adapted from *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle.

Sequential vs. Overlap



Scrum Framework

Roles

- Product owner
- Scrum master
- Team

Ceremonies

- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

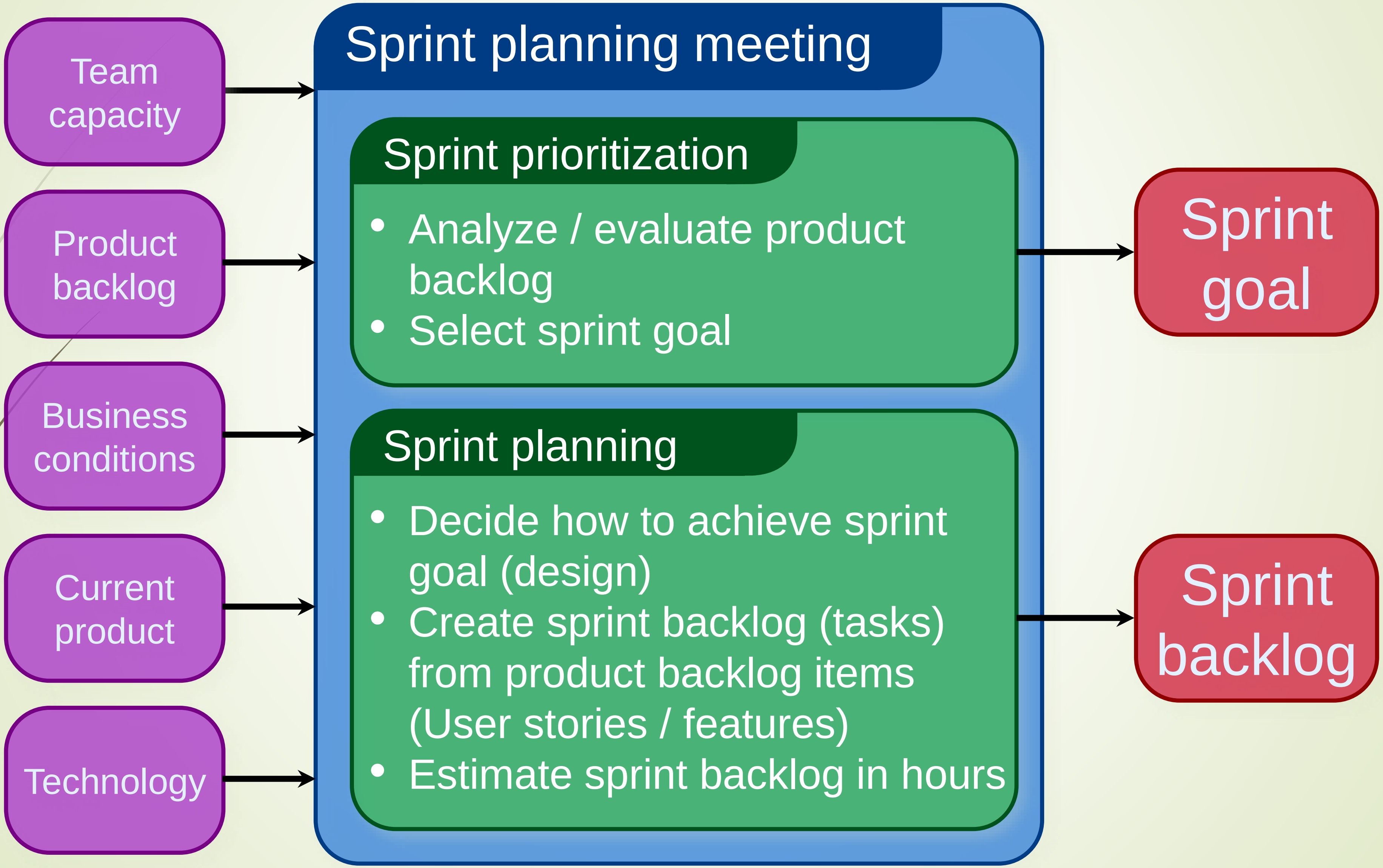
Scrum Roles

- **Product Owner**
  - Possibly a **Product Manager or Project Sponsor**
  - Decides features, release date, prioritization, $$$

- **Scrum Master**
  - Typically a **Team Leader or Project Manager**
  - Responsible for enacting Scrum values and practices
  - Remove impediments / politics, keeps everyone productive

- **Project Team**
  - **5-10 members**; Teams are self-organizing
  - Cross-functional: QA, Programmers, UI Designers, etc.
  - Membership can change only between sprints

Sprint Planning



Daily Scrum Meeting

- **Parameters**
  - **Daily, ~15 minutes, Stand-up**
  - Anyone late pays a $1 fee

- **Not for problem solving**
  - Whole world is invited
  - **Only team members, Scrum Master & product owner can talk**
  - Helps avoid other unnecessary meetings

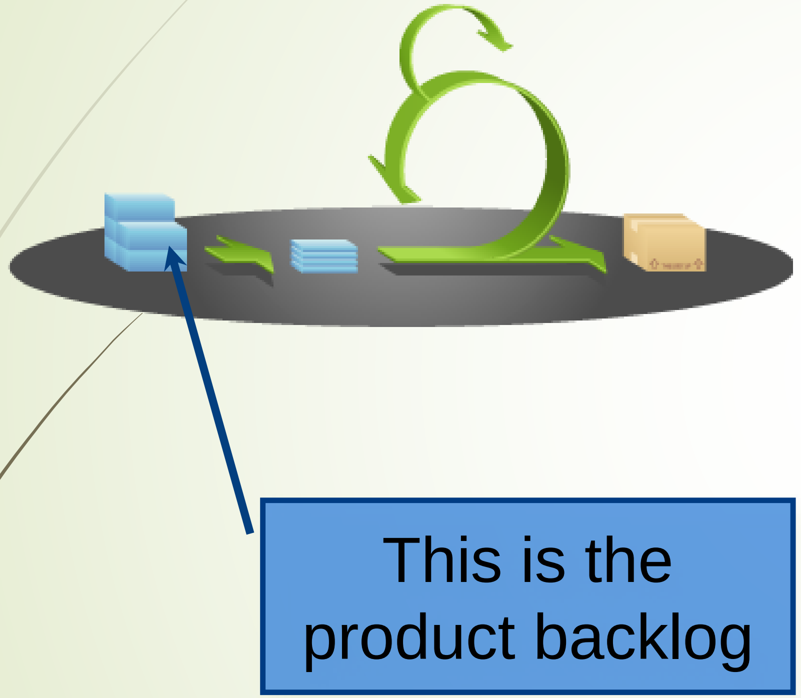
- **Three questions answered by each team member:**
  1. What did you do **yesterday**?
  2. What will you do **today**?
  3. What **obstacles** are in your way?

Scrum's Artifacts

- Scrum has remarkably few artifacts
  - **Product Backlog**
  - **Sprint Backlog**
  - **Burndown Charts**

- Can be managed using just an **Excel spreadsheet**
  - More **advanced / complicated tools** exist:
    - **Expensive**
    - **Web-based** – no good for Scrum Master/project manager who travels
    - **Still under development**

Product Backlog



- A list of **requirements & all desired work** on project
- Ideally expressed as a **list of User Stories** along with **'Story Points'**, such that each item has value to users or customers of the product
- **Prioritized by the product owner**
- **Reprioritized at start of each sprint**

User Stories

- Instead of Use Cases, Agile project owners do **'User Stories'**
  - **Who** (user role) – Is this a customer, employee, admin, etc.?
  - **What** (goal) – What functionality must be achieved/developed?
  - **Why** (reason) – Why does user want to accomplish this goal?

  As a **[user role]**, I want to **[goal]**, so I can **[reason]**.

- **Example:**
  - **"As a user, I want to log in, so I can access subscriber content."**

- **Story points**: Rating of effort needed to implement this story
  - **Common Scales: 1-10**

Sample Product Backlog

Backlog Item	Estimate
Allow a guest to make a reservation	3 (Story Points)
As a guest, I want to cancel a reservation.	05
As a guest, I want to change the dates of a reservation.	03
As a hotel employee, I can run RevPAR reports (revenue-per-available-room)	08
Improve exception handling	08
...	10
...	10

Sample Product Backlog (Conti…)

Product Backlog
Estimating System Upgrade

Sprint	ID		Backlog Item	Owner	Estimate (days)	Remaining (days)
1	1	Minor	Remove user kludge in .dpr file	BC	1	1
1	2	Minor	Remove cMap/cMenu/cMenuSize from disciplines.pas	BC	1	1
1	3	Minor	Create "Legacy" discipline node with old civils and E&I content	BC	1	1
1	4	Major	Augment each tbl operation to support network operation	BC	10	10
1	5	Major	Extend Engineering Design estimate items to include summaries	BC	2	2
1	6	Super	Supervision/Guidance	CAM	4	4
	7	Minor	Remove Custodian property from AppConfig class in globals.pas	BC	1	
	8	Minor	Remove LOC_ constants in globals.pas and main.pas	BC	1	
	9	Minor	New E&I section doesn't have lblCaption set	BC	1	
	10	Minor	Delay in main.releaseform doesn't appear to be required	BC	1	
	11	Minor	Undo modifications to Other Major Equipment in formExcel.pas	BC	1	
	12	Minor	AJACS form to be centred on the screen	BC	1	
	13	Major	Extend DUnit tests to all 40 disciplines	BC	6	

Sprint Backlog

- Individuals sign up for **work of their own choosing.**
  - **Work is never assigned.**
- Estimated **work remaining is updated daily**.
- **Any team member can add, delete or change sprint backlog.**
- **Work** for the sprint **emerges.**
- If work is unclear, **define a sprint backlog item with a larger amount of time** and break it down later.
- **Update work remaining** as more becomes known.

Sample Sprint Backlog

Tasks	Mon	Tue	Wed	Thu	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	4	
Test the middle tier	8	16	16	11	8
Write online help	12				
Write the Foo class	8	8	8	8	8
Add error logging			8	4	

Sample Sprint Backlog

Sprint 1

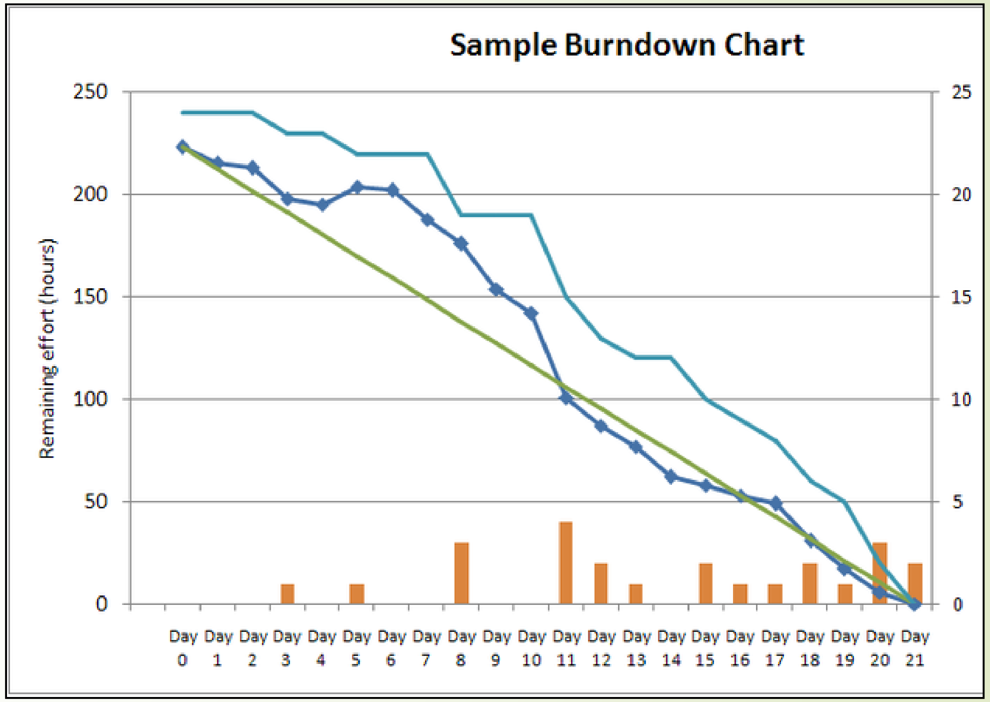
01/11/2004				Sprint Day	1	2	3	4	5	6	7
					Mo	Tu	We	Th	Fr	Sa	Su
	19 days work in this sprint			Hours remaining	152	152	152	152	152	152	152
	Backlog Item	Backlog Item	Owner	Estimate							
1	Minor	Remove user kludge in .dpr file	BC	8	8	8	8	8	8	8	8
2	Minor	Remove cMap/cMenu/cMenuSize from disciplines.pas	BC	8	8	8	8	8	8	8	8
3	Minor	Create "Legacy" discipline node with old civils and E&I content	BC	8	8	8	8	8	8	8	8
4	Major	Augment each tbl operation to support network operation	BC	80	80	80	80	80	80	80	80
5	Major	Extend Engineering Design estimate items to include summaries	BC	16	16	16	16	16	16	16	16
6	Super	Supervision/Guidance	CAM	32	32	32	32	32	32	32	32

Sprint 1

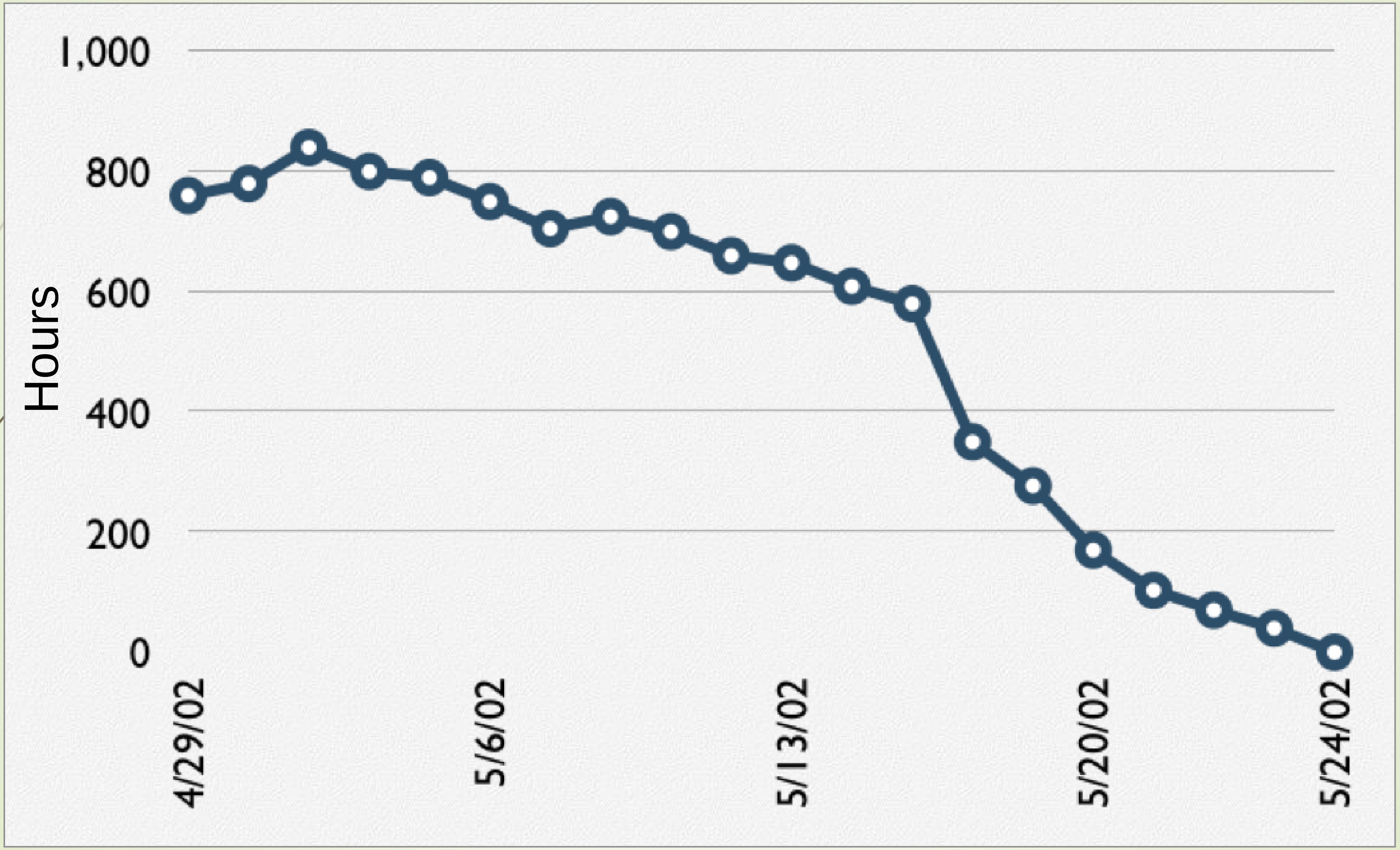
01/11/2004				Sprint Day	1	2	3	4	5	6	7
					Mo	Tu	We	Th	Fr	Sa	Su
	19 days work in this sprint			Hours remaining	152	150	140	130	118	118	118
	Backlog Item	Backlog Item	Owner	Estimate							
1	Minor	Remove user kludge in .dpr file	BC	8	8	8	4	2	0		
2	Minor	Remove cMap/cMenu/cMenuSize from disciplines.pas	BC	8	8	8	4	0			
3	Minor	Create "Legacy" discipline node with old civils and E&I content	BC	8	8	8	8	6	0		
4	Major	Augment each tbl operation to support network operation	BC	80	80	80	80	80	78	78	78
5	Major	Extend Engineering Design estimate items to include summaries	BC	16	16	16	16	16	16	16	16
6	Super	Supervision/Guidance	CAM	32	32	30	28	26	24	24	24

Sprint Burndown Chart

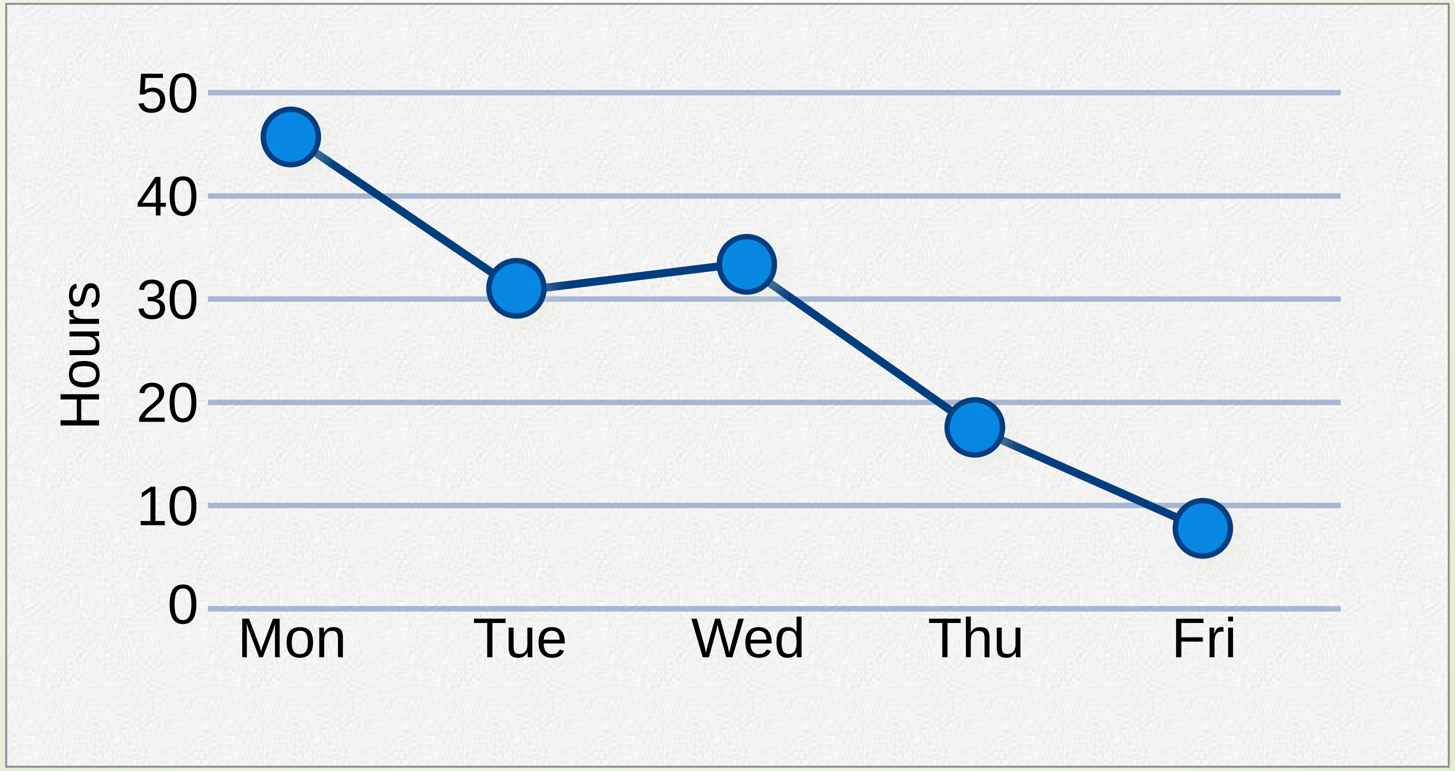
- A **display of what work has been completed and what is left** to complete -
  - one for **each developer or work item**
  - **updated every day**
  - make best guess about hours / points completed each day
- *Variation:* Release burndown chart
  - shows **overall progress**
  - updated at end of each sprint



Sample Burndown Chart



Tasks	Mon	Tue	Wed	Thu	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	7	
Test the middle tier	8	16	16	11	8
Write online help	12				



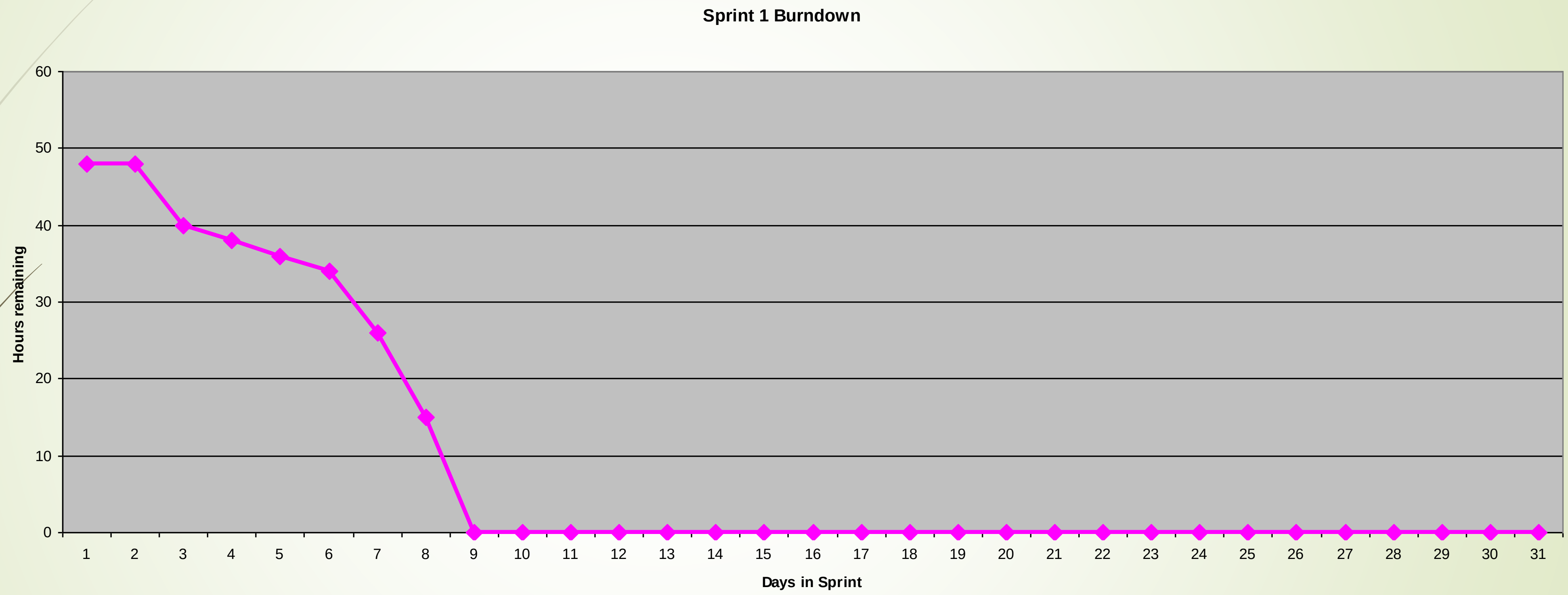
Burndown Example: 1

No work being performed

Burndown Example: 2

Work being performed, but not fast enough

Burndown Example: 3



Work being performed, but too fast!

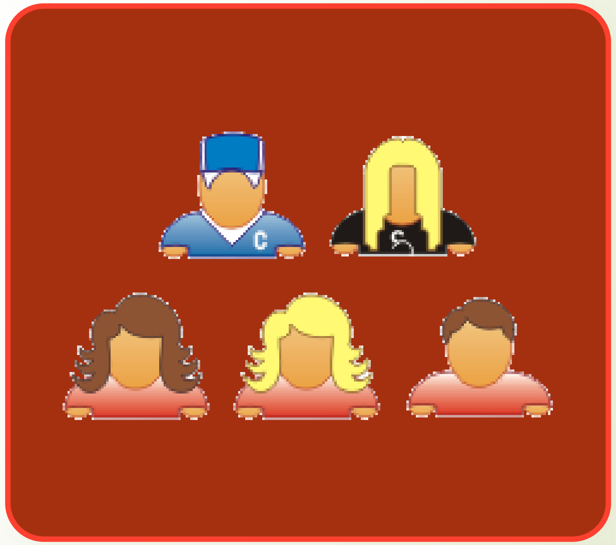
The Sprint Review

- Team presents **what it accomplished** during the sprint.
- Typically takes the form of a **demo of new features or underlying architecture**.
- **Informal**
  - **2-hour prep time rule**
  - **No slides**
- Whole **team participates**
- The **world is invited**

Scalability

- Typical individual **team size is upto 7 ± 2 people**. (Miller's Principle)
  - Scalability comes from teams of teams.

- **Factors in scaling**
  - **Type of application**
  - **Team size**
  - **Team dispersion**
  - **Project duration**



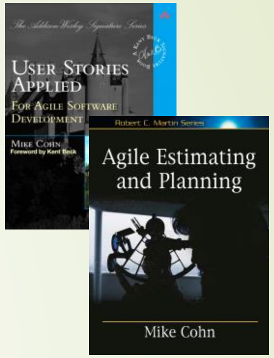
- Scrum has been used on **multiple 500+ person projects**.

Scaling: Scrum of Scrums

Scrum vs. Other Models

	Scrum	Traditional Project Management
Emphasis is on	People	Processes
Documentation	Minimal—only as required	Comprehensive
Process style	Iterative	Linear
Upfront planning	Low	High
Prioritization of Requirements	Based on business value and regularly updated	Fixed in the Project Plan
Quality assurance	Customer centric	Process centric
Organization	Self-organized	Managed
Management style	Decentralized	Centralized
Change	Updates to Productized Product Backlog	Formal Change Management System
Leadership	Collaborative, Servant Leadership	Command and control
Performance measurement	Business value	Plan conformity
Return on Investment	Early/throughout project life	End of project life
Customer involvement	High throughout the project	Varies depending on the project lifecycle

References



- Mike Cohn, Mountain Goat Software
  www.mountaingoatsoftware.com
- *Scrum and The Enterprise* by Ken Schwaber
- *Succeeding with Agile* by Mike Cohn
- *Agile Software Development Ecosystems* by Jim Highsmith
- *Agile Software Development with Scrum* by K. Schwaber and M. Beedle
- *User Stories Applied for Agile Software Development* by Mike Cohn
- Marty Stepp - http://www.cs.washington.edu/403/
- www.agilescrum.com/
- www.objectmentor.com
- jeffsutherland.com/
- www.controlchaos.com/scrumwp.htm
- agilealliance.com/articles/articles/InventingScrum.pdf

Thank You !

Design Patterns

PRESENTED BY SANGEETA MEHTA

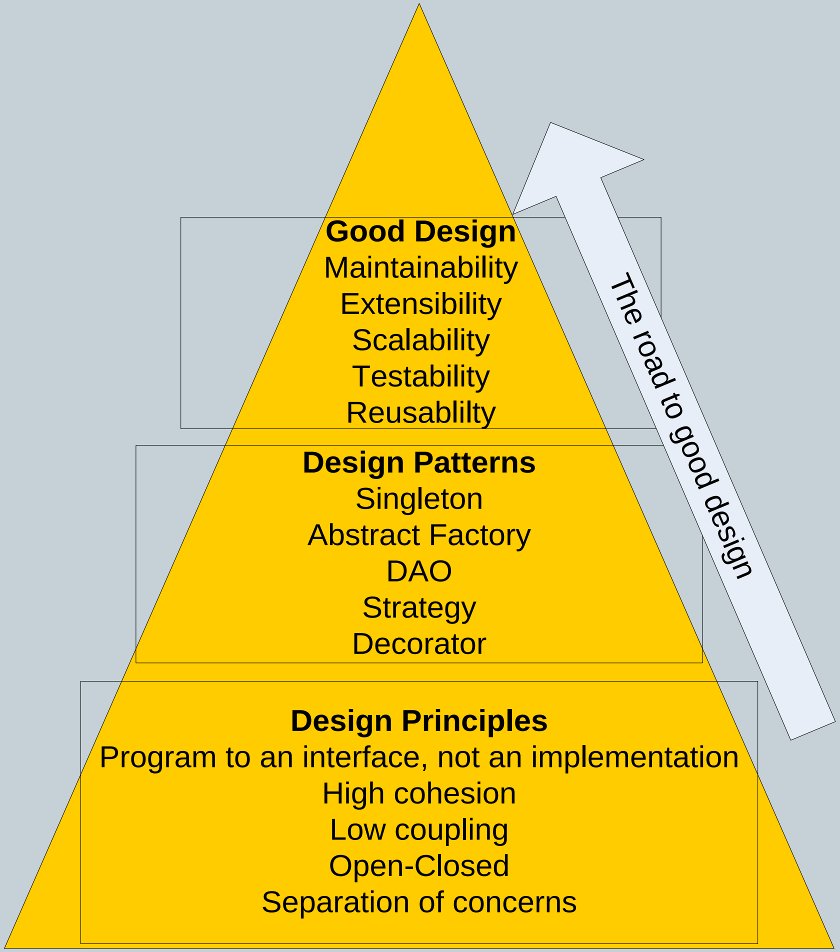
EECS810
UNIVERSITY OF KANSAS
OCTOBER 2008

Contents

What are Design Patterns?

- What Are Design Patterns?
  - Wikipedia definition
    - "a design pattern is a general repeatable solution to a commonly occurring problem in software design"
  - Quote from Christopher Alexander
    - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (GoF,1995)

Why use Design Patterns?



Why use Design Patterns?

- Design Objectives
  - Good Design (the "ilities")
    - High readability and maintainability
    - High extensibility
    - High scalability
    - High testability
    - High reusability

Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
    - Describes the pattern
    - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
    - Describes when to apply the pattern
    - Answers - What is the pattern trying to solve?

Elements of a Design Pattern (cont.)

- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
  - Results of applying the pattern
  - Benefits and Costs
  - Subjective depending on concrete scenarios

Design Patterns Classification

A Pattern can be classified as

- Creational
- Structural
- Behavioral

Pros/Cons of Design Patterns

- Pros:
  - Add **consistency** to designs by solving similar problems the same way, independent of language.
  - Add **clarity** to design and design communication by enabling a common vocabulary.
  - Improve **time** to solution by providing templates which serve as foundations for good design.
  - Improve **reuse** through composition.

Pros/Cons of Design Patterns

- Cons:
  - Some patterns come with **negative consequences** (i.e. object proliferation, performance reduction or additional layers)
  - Consequences are **subjective** depending on concrete scenarios.
  - Patterns are subject to **different interpretations, misinterpretations**, and philosophies.
  - Patterns can be **overused and abused** → Anti-Patterns

Popular Design Patterns

- Let's take a look
  - Strategy
  - Observer
  - Singleton
  - Decorator
  - Proxy
  - Façade
  - Adapter

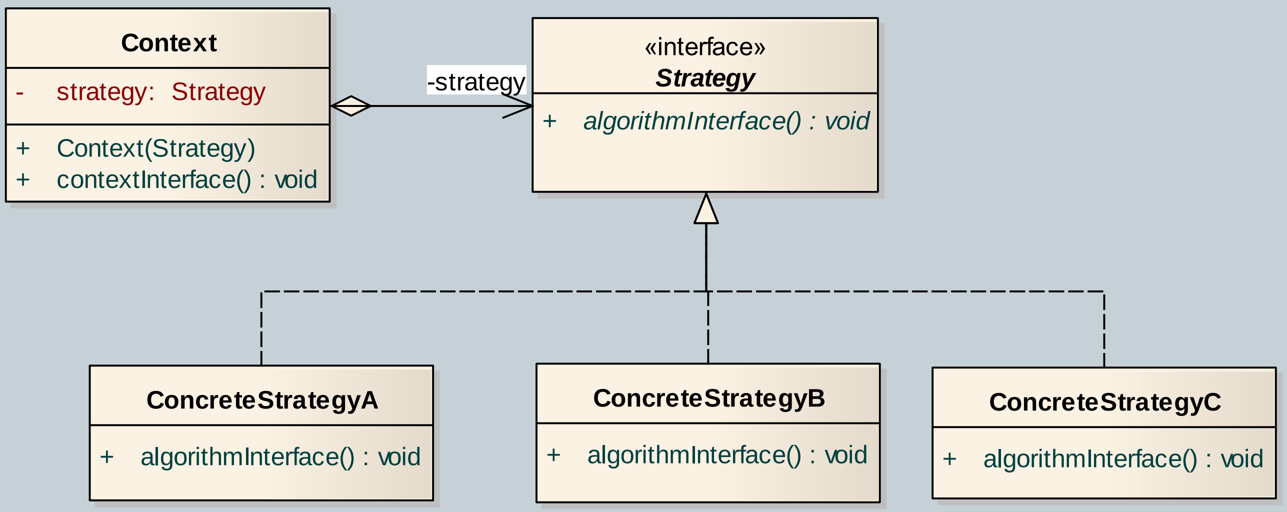
Strategy Definition

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

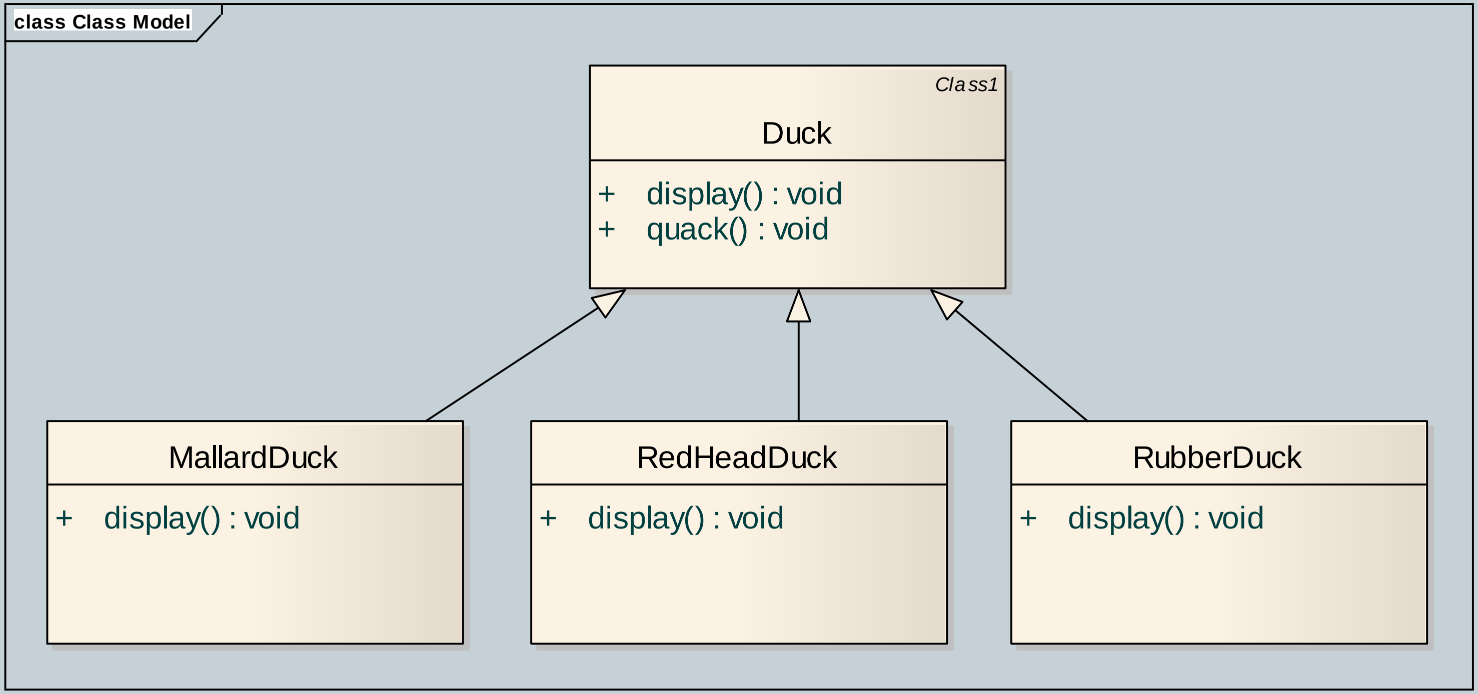
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interface, not an implementation
- Favor composition over inheritance

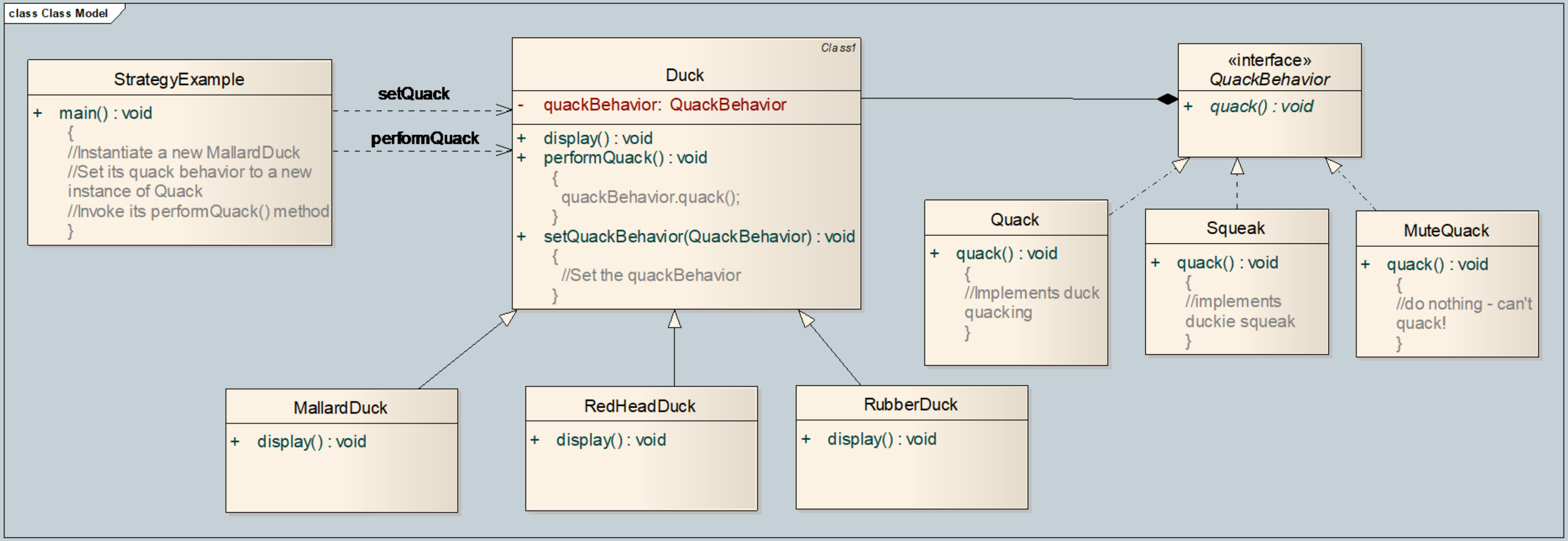
Strategy - Class diagram



Strategy - Problem



Strategy - Solution



Strategy

- Pros
  - Provides encapsulation
  - Hides implementation
  - Allows behavior change at runtime
- Cons
  - Results in complex, hard to understand code if overused

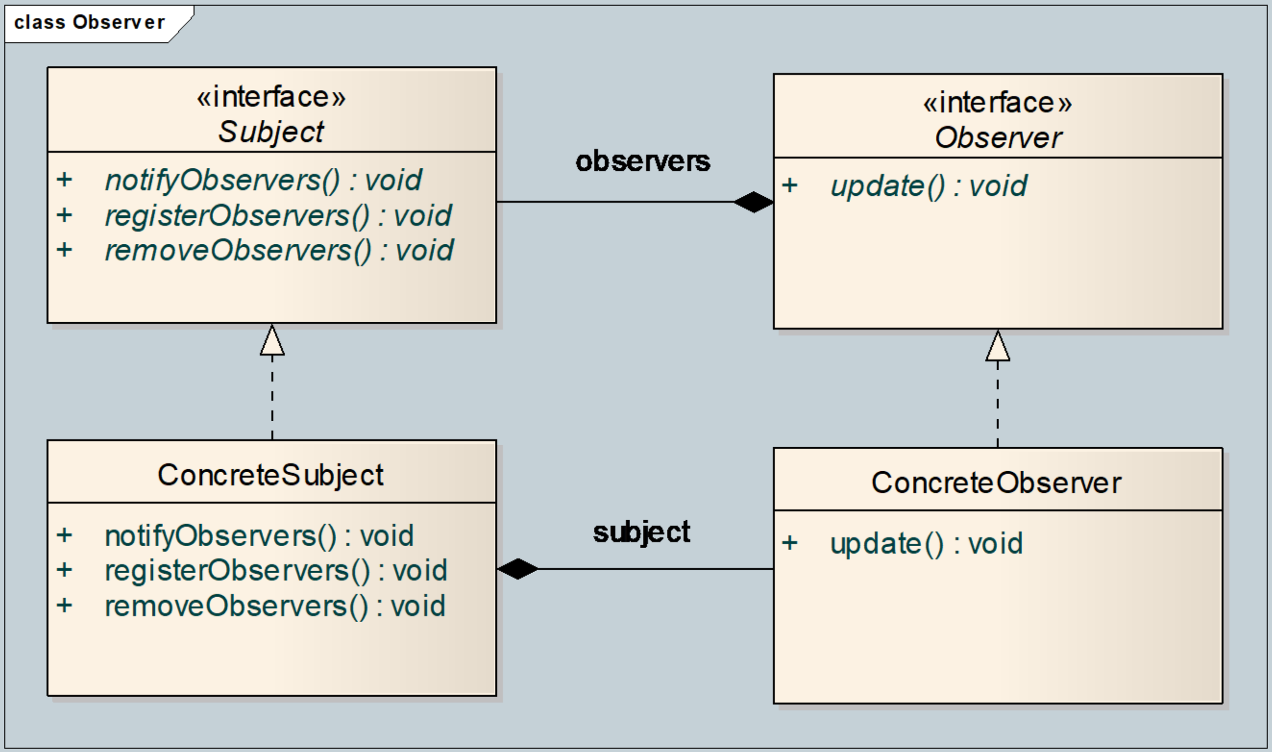
Observer Definition

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

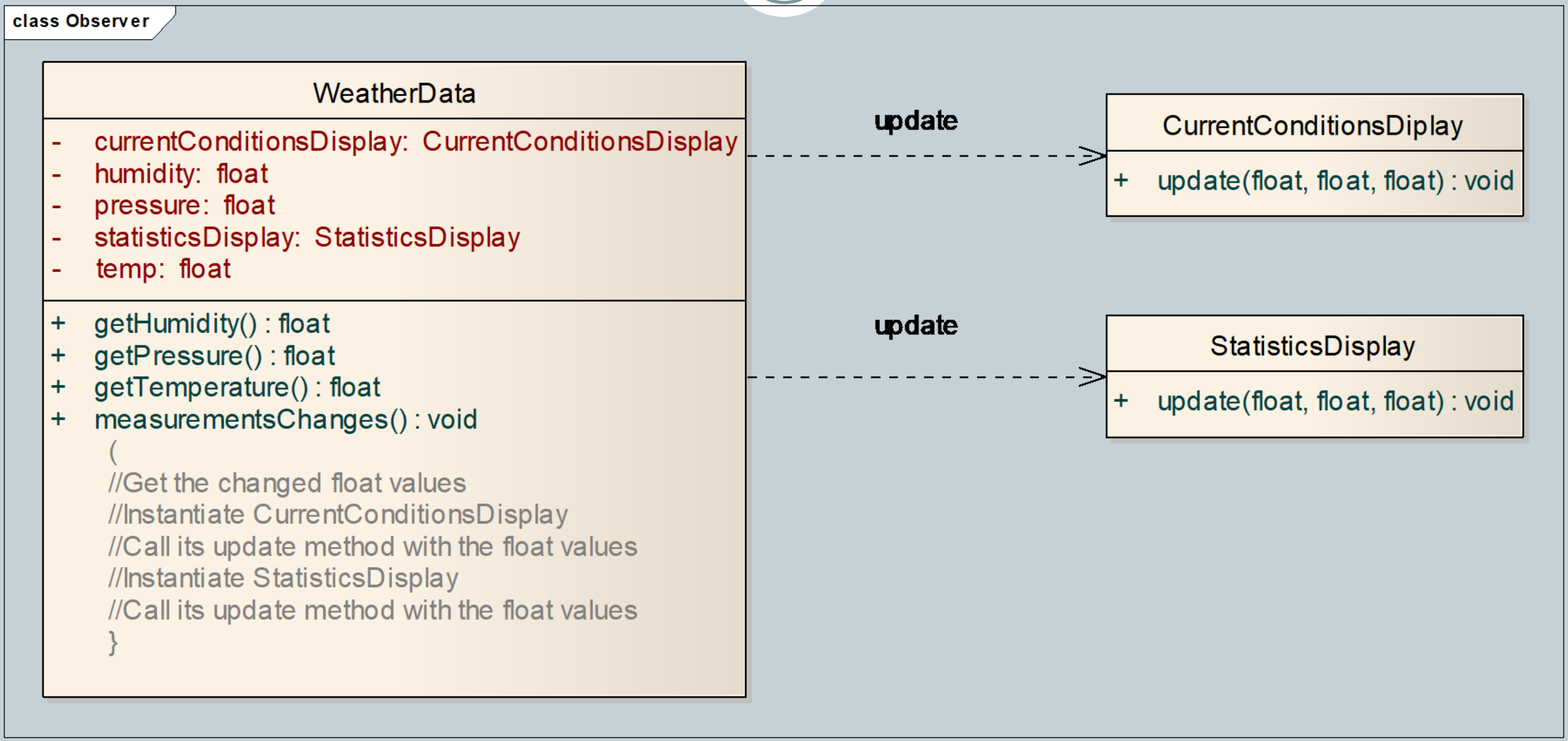
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact

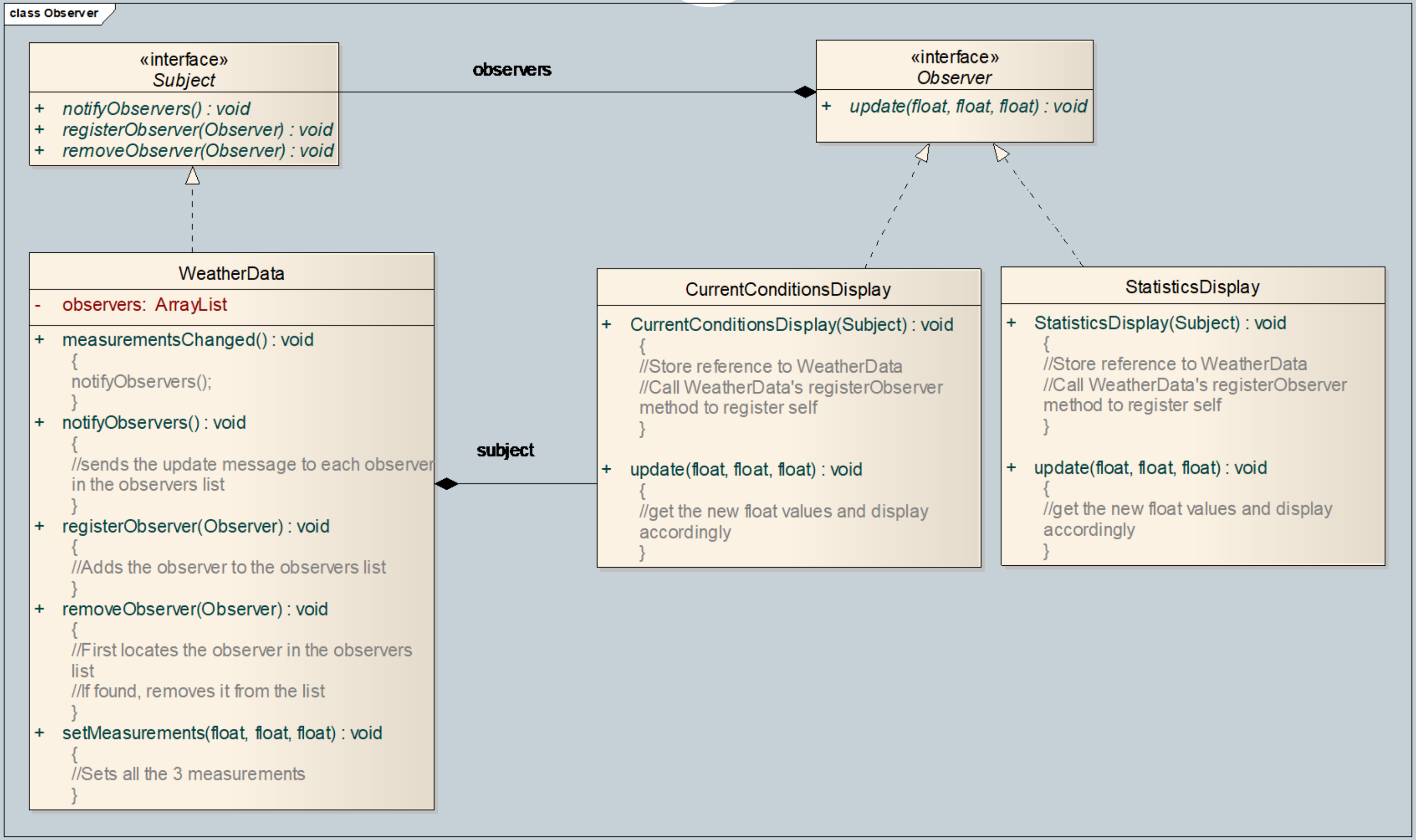
Observer - Class diagram



Observer - Problem



Observer - Solution



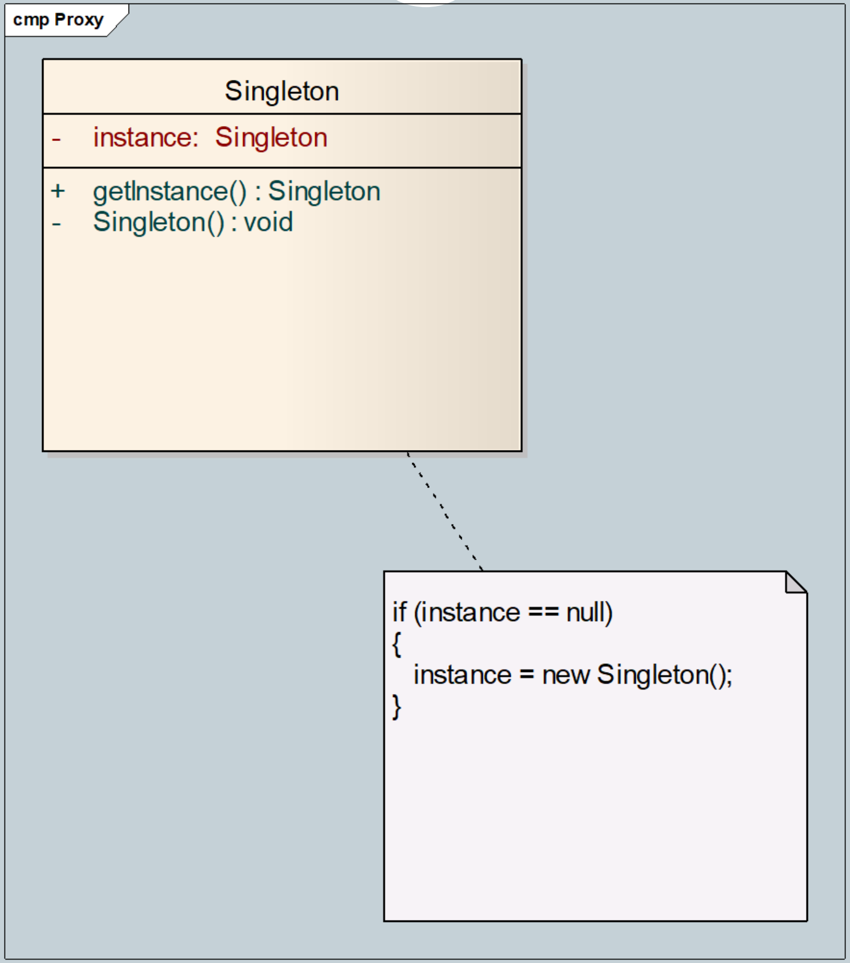
Observer

- Pros
  - Abstracts coupling between Subject and Observer
  - Supports broadcast communication
  - Supports unexpected updates
  - Enables reusability of subjects and observers independently of each other
- Cons
  - Exposes the Observer to the Subject (with push)
  - Exposes the Subject to the Observer (with pull)

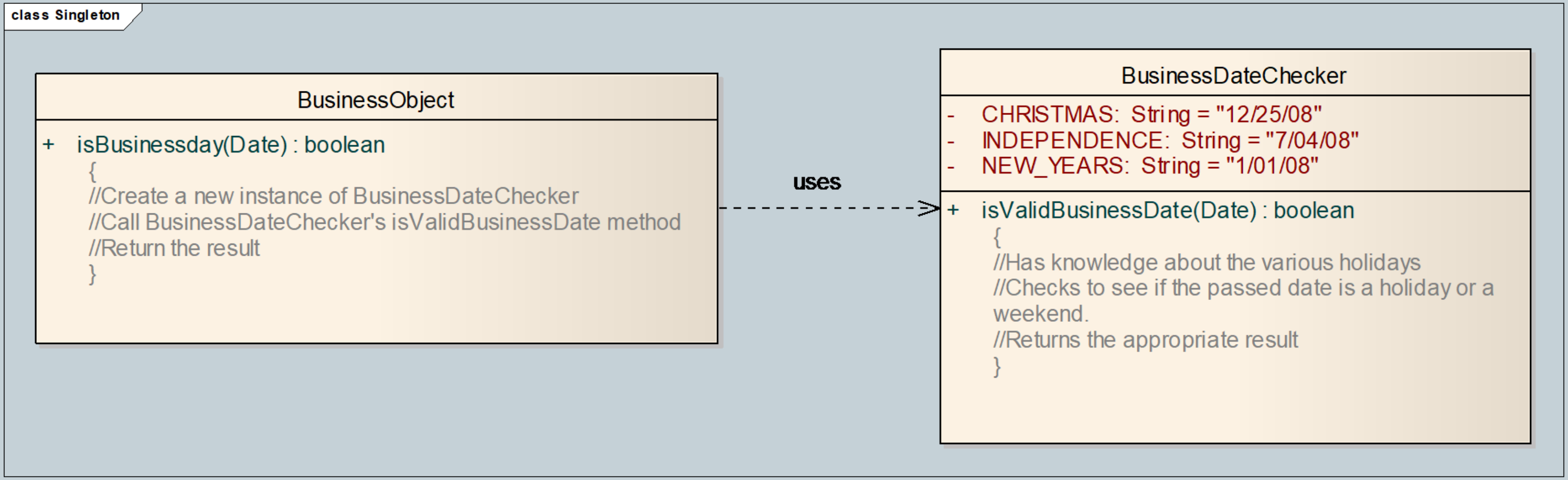
Singleton Definition

Ensure a class only has one instance and provide a global point of access to it.

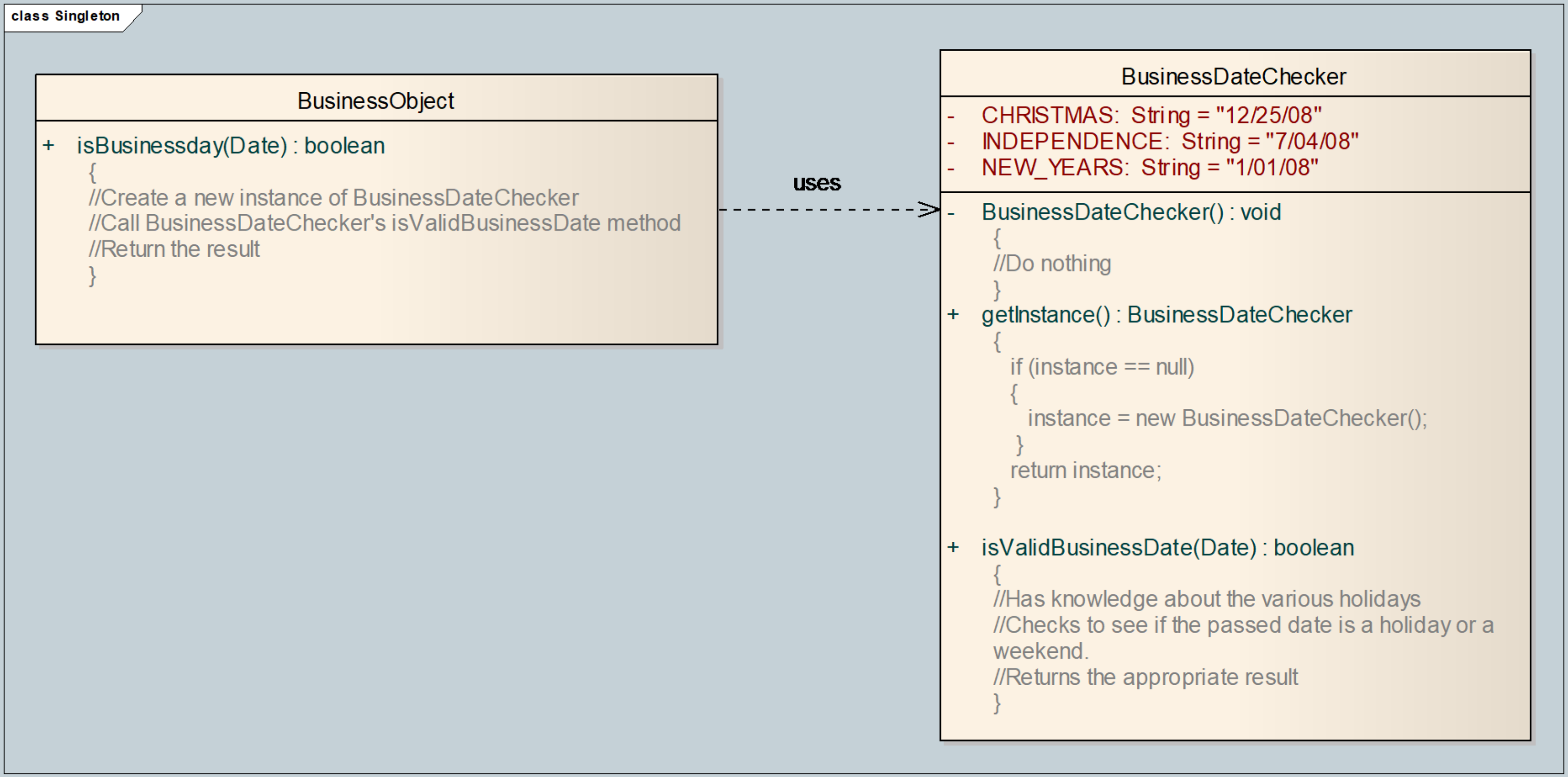
Singleton - Class diagram



Singleton - Problem



Singleton - Solution



Singleton

cmp Proxy

```java
public class Singleton {
        private static Singleton instance = null;
        protected Singleton() {
            //Exists only to defeat instantiation.
        }

        public static Singleton getInstance() {
            if(instance == null) {
              instance = new Singleton();
            }

           return instance;
}
```

```java
public class SingletonInstantiator {
    public SingletonInstantiator() {
      Singleton instance = Singleton.getInstance();
      Singleton anotherInstance = new Singleton();
      ......
 }
```

Singleton

- Pros
  - Increases performance
  - Prevents memory wastage
  - Increases global data sharing

- Cons
  - Results in multithreading issues

Patterns & Definitions - Group 1

- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

Patterns & Definitions - Group 1

- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

Patterns & Definitions - Group 1

- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

Patterns & Definitions - Group 1

- Strategy
- Observer
- Singleton

- Allows objects to be notified when state changes
- Ensures one and only one instance of an object is created
- Encapsulates inter-changeable behavior and uses delegation to decide which to use

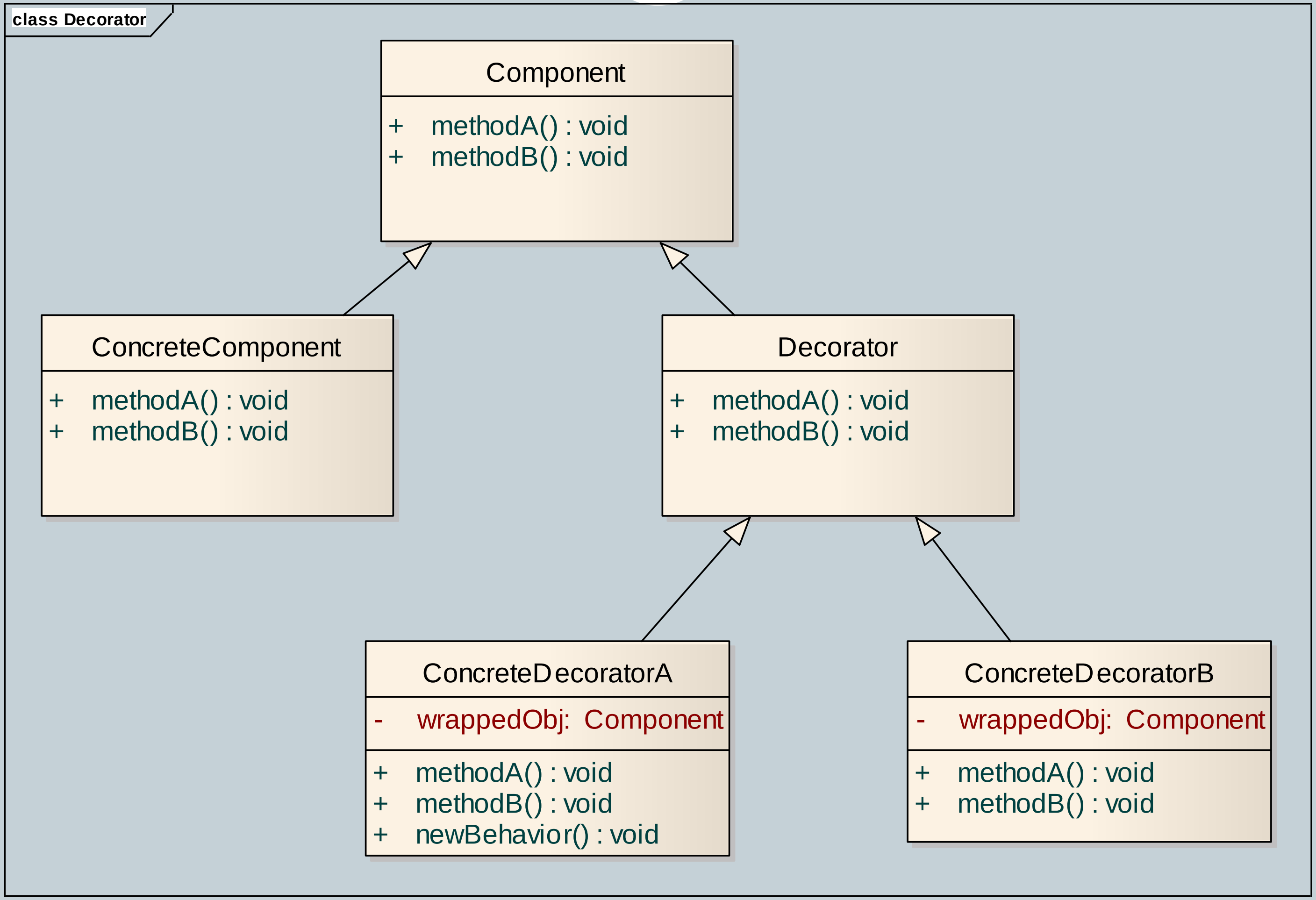
Decorator Definition

Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

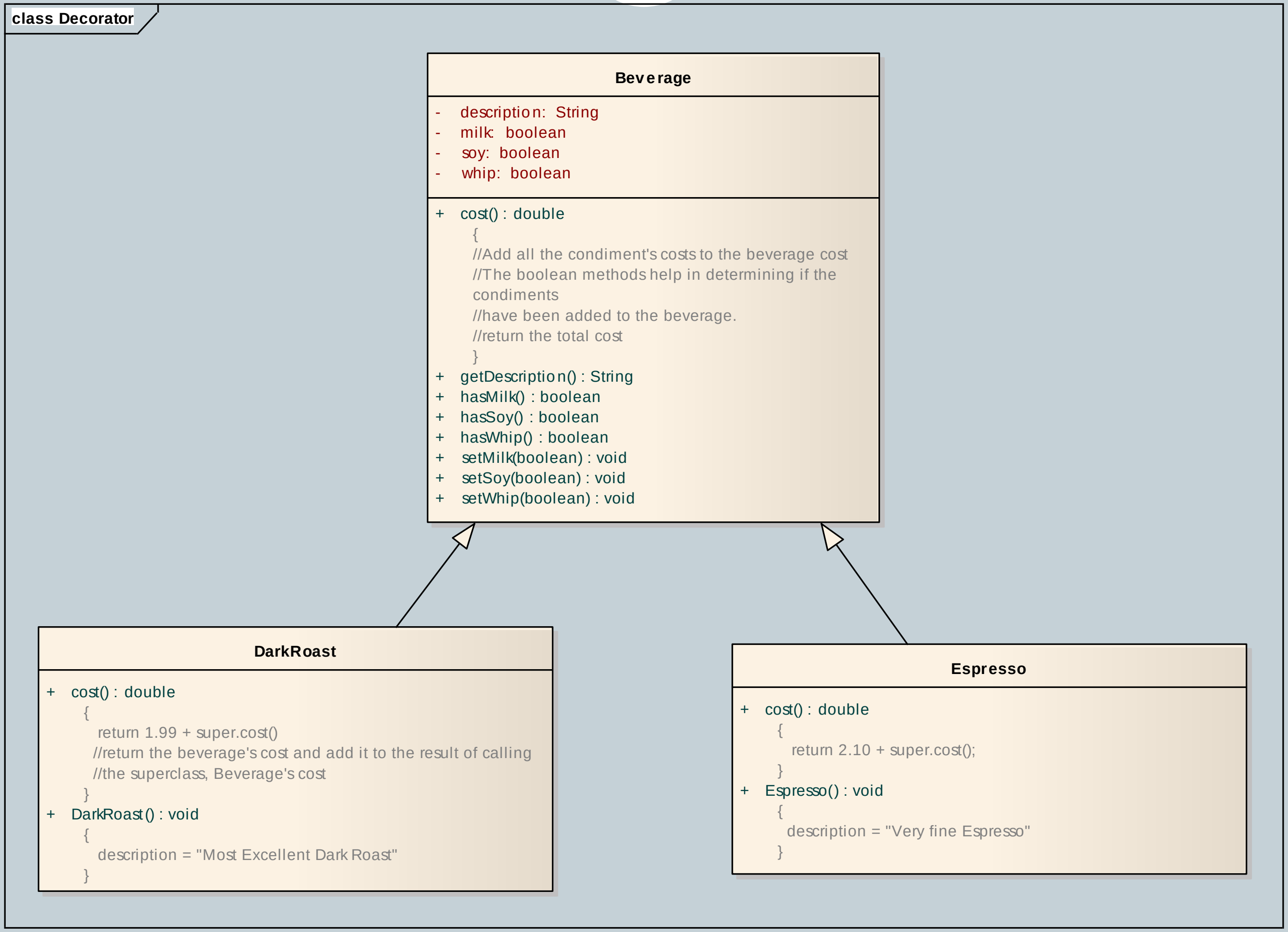
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification

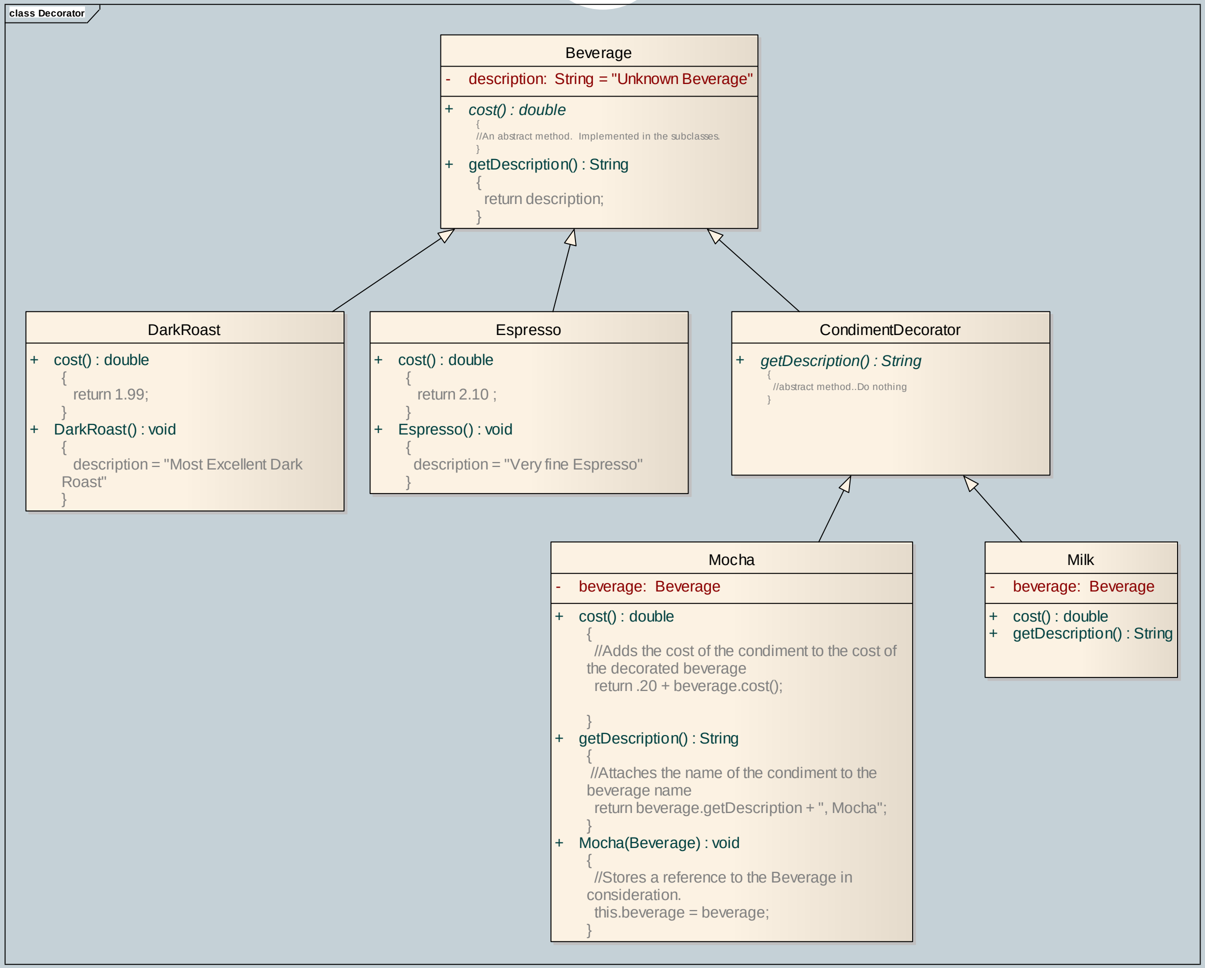
Decorator - Class diagram



Decorator - Problem



Decorator - Solution



Decorator

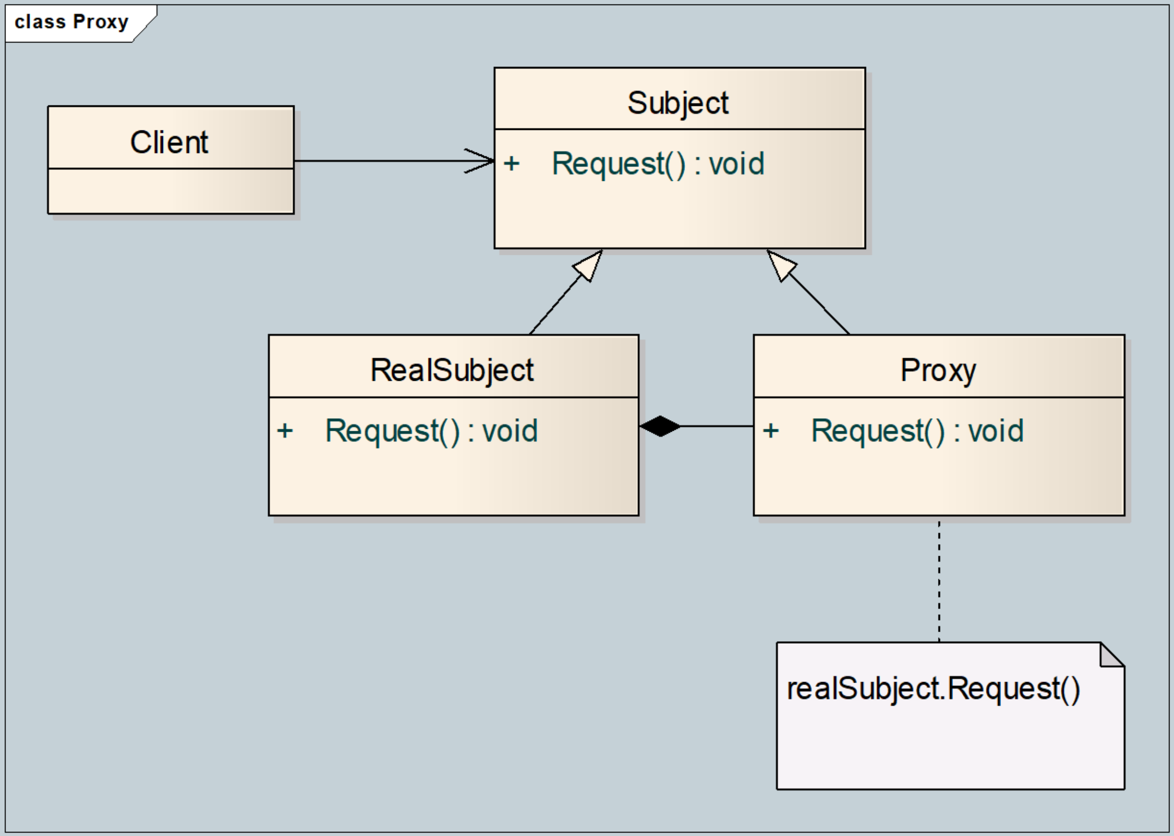
- Pros
  - Extends class functionality at runtime
  - Helps in building flexible systems
  - Works great if coded against the abstract component type

- Cons
  - Results in problems if there is code that relies on the concrete component’s type

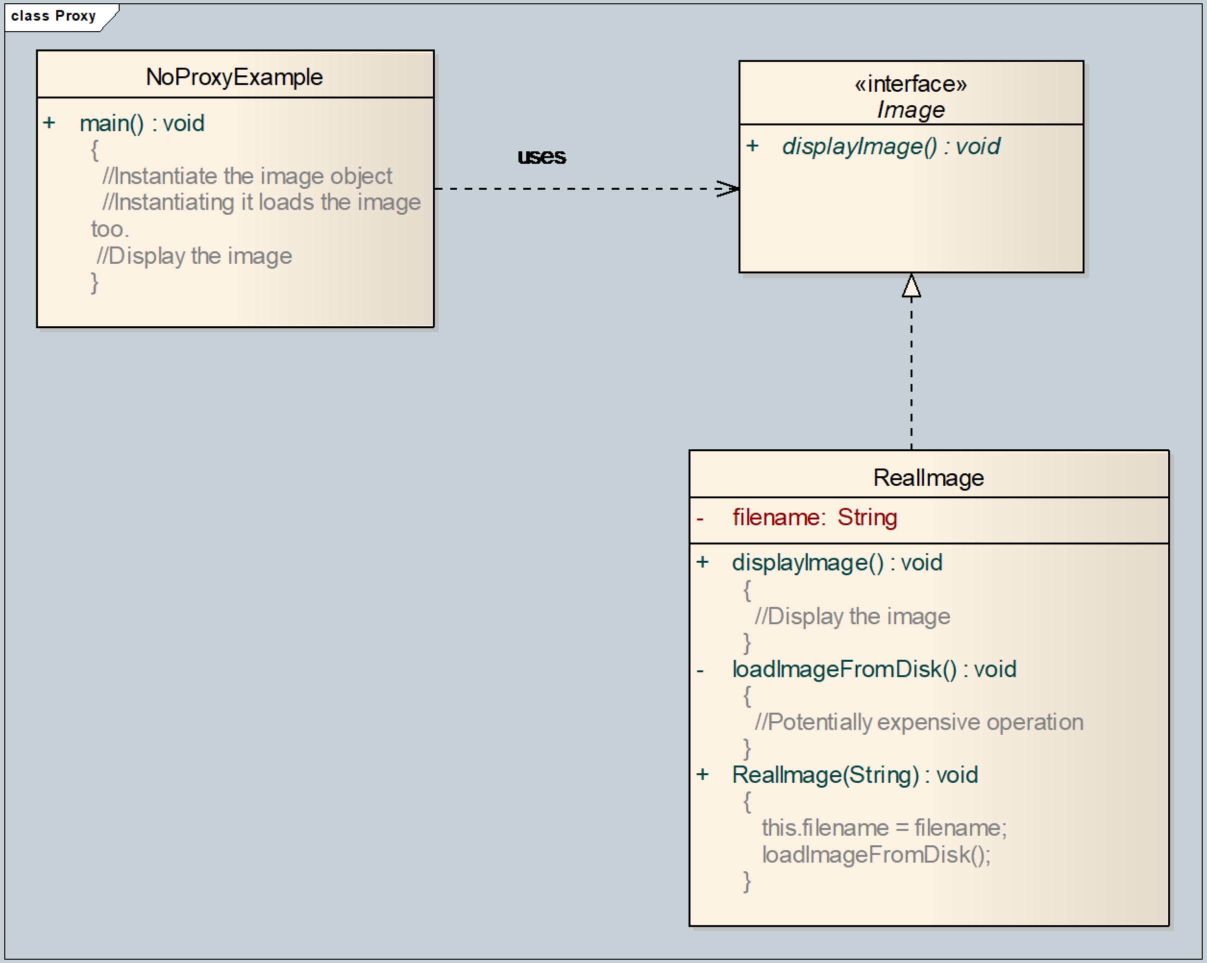
Proxy Definition

Provides a surrogate or placeholder for another object to control access to it

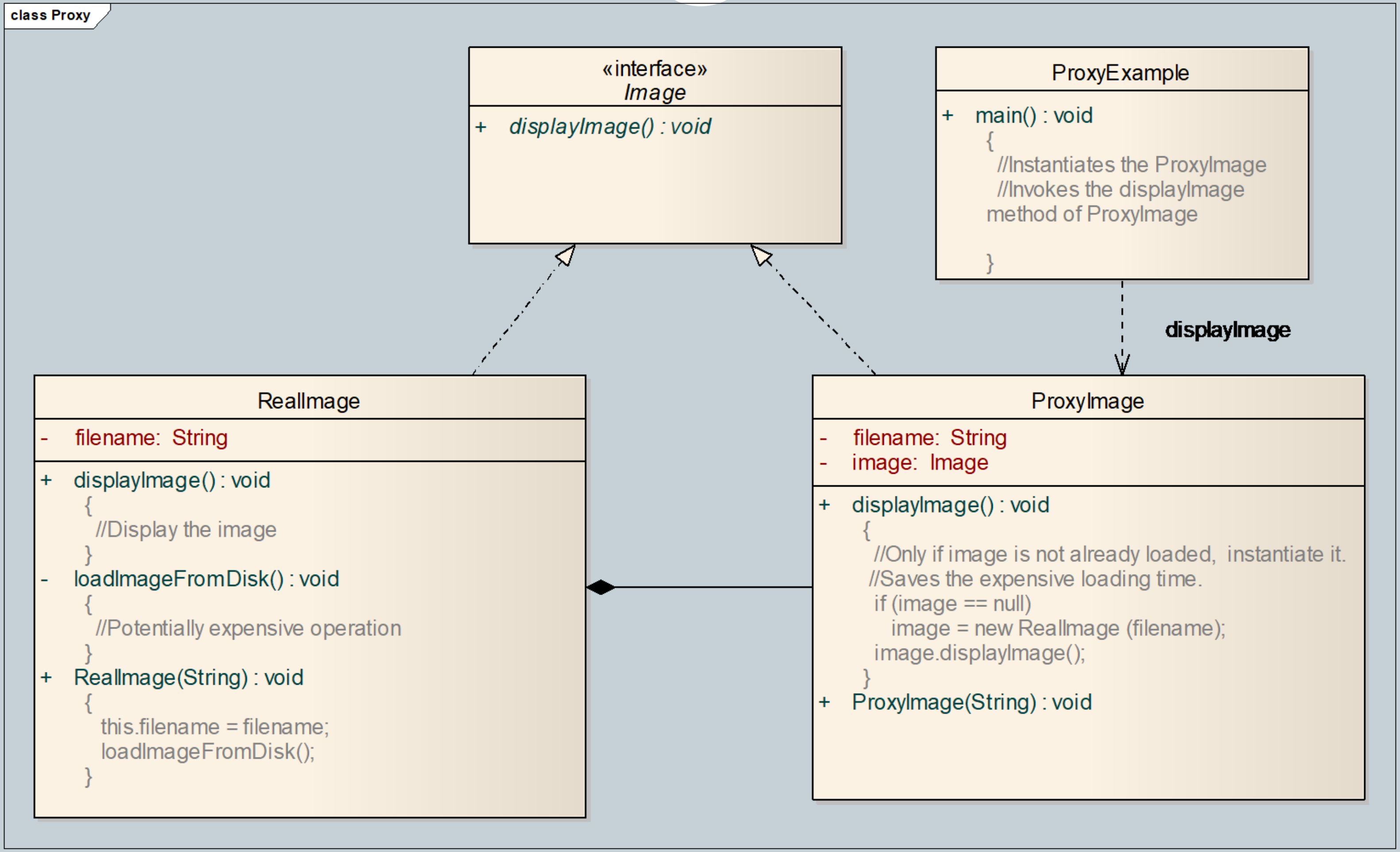
Proxy - Class diagram



Proxy - Problem



Proxy - Solution



Proxy

- Pros
  - Prevents memory wastage
  - Creates expensive objects on demand
- Cons
  - Adds complexity when trying to ensure freshness

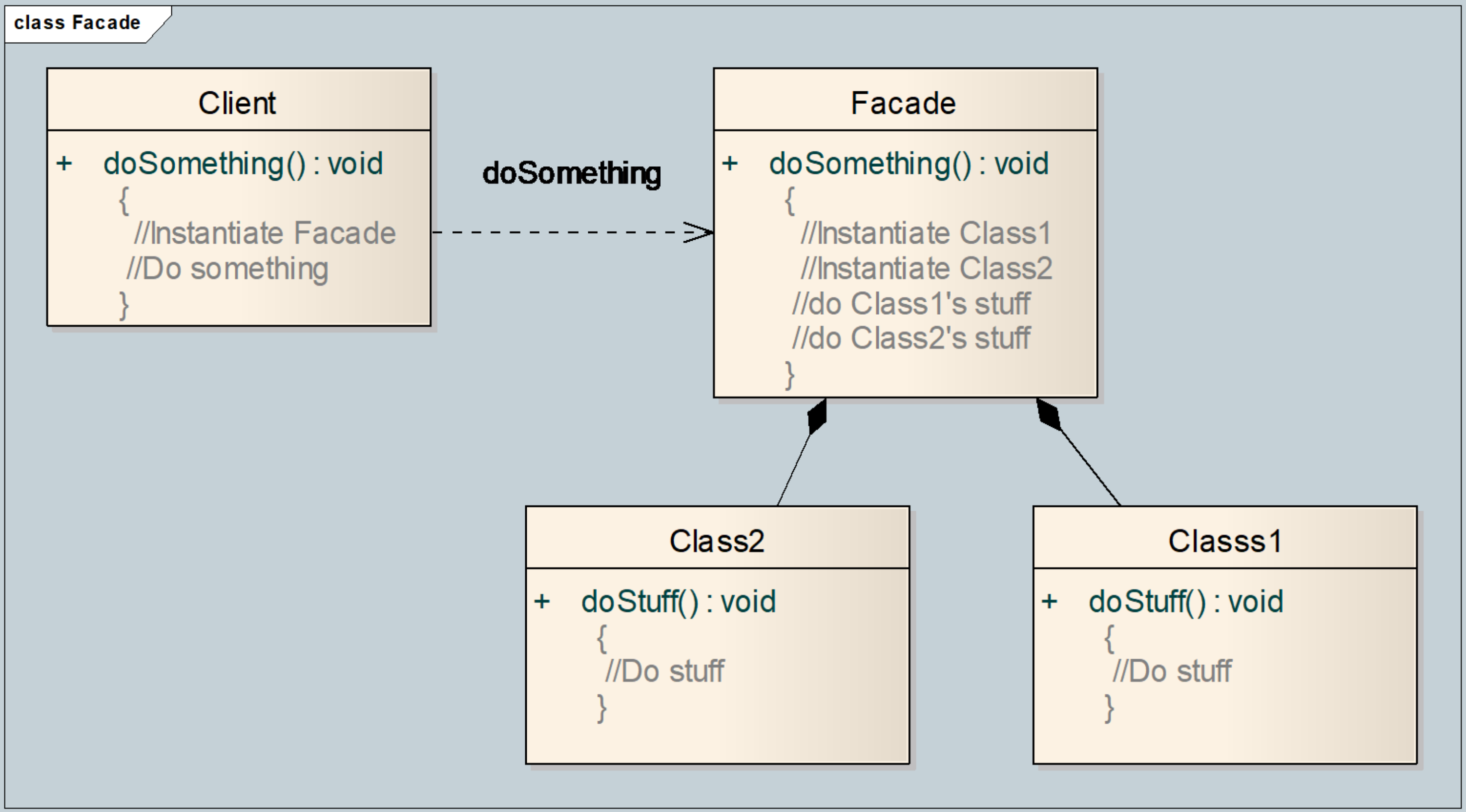
Facade Definition

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.

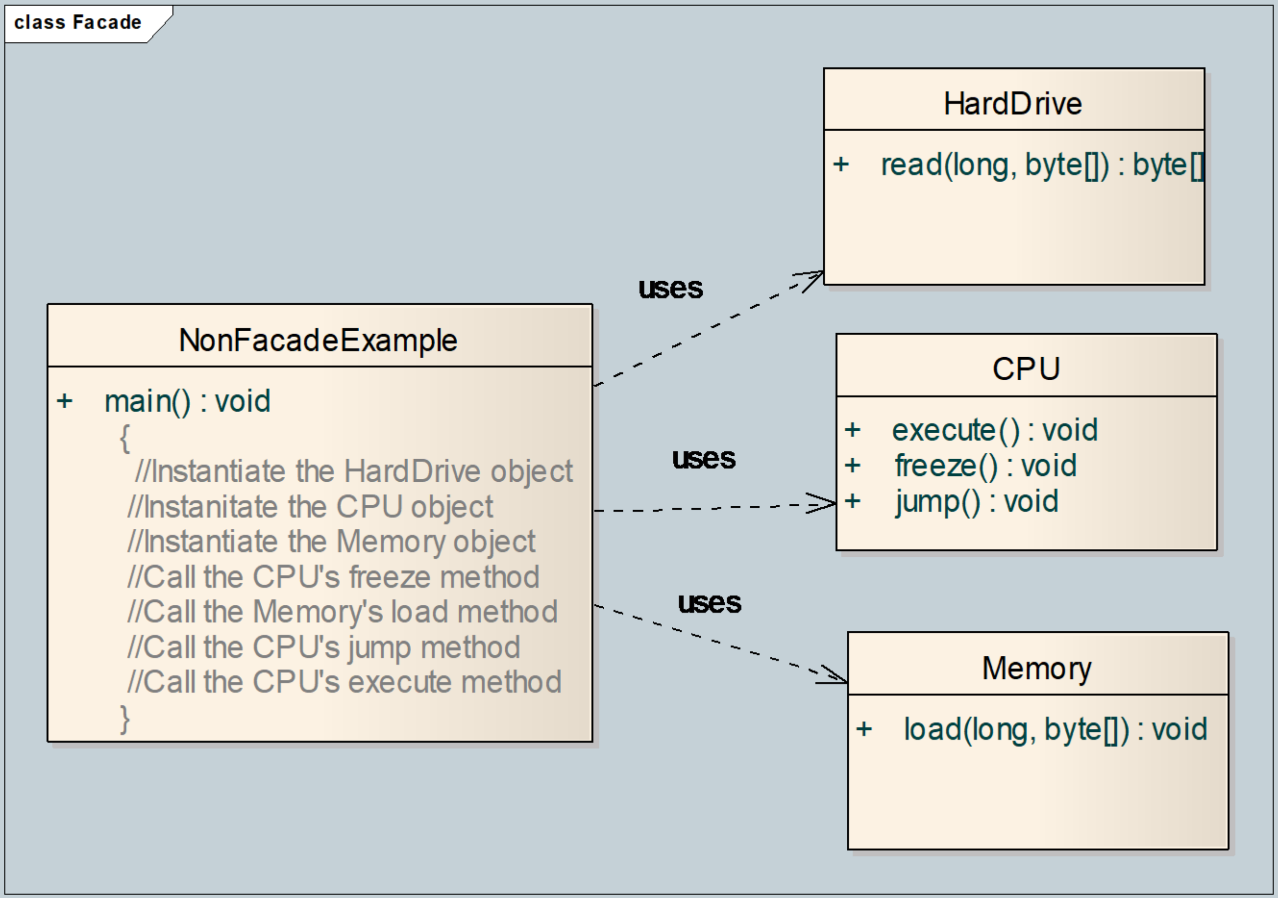
Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends

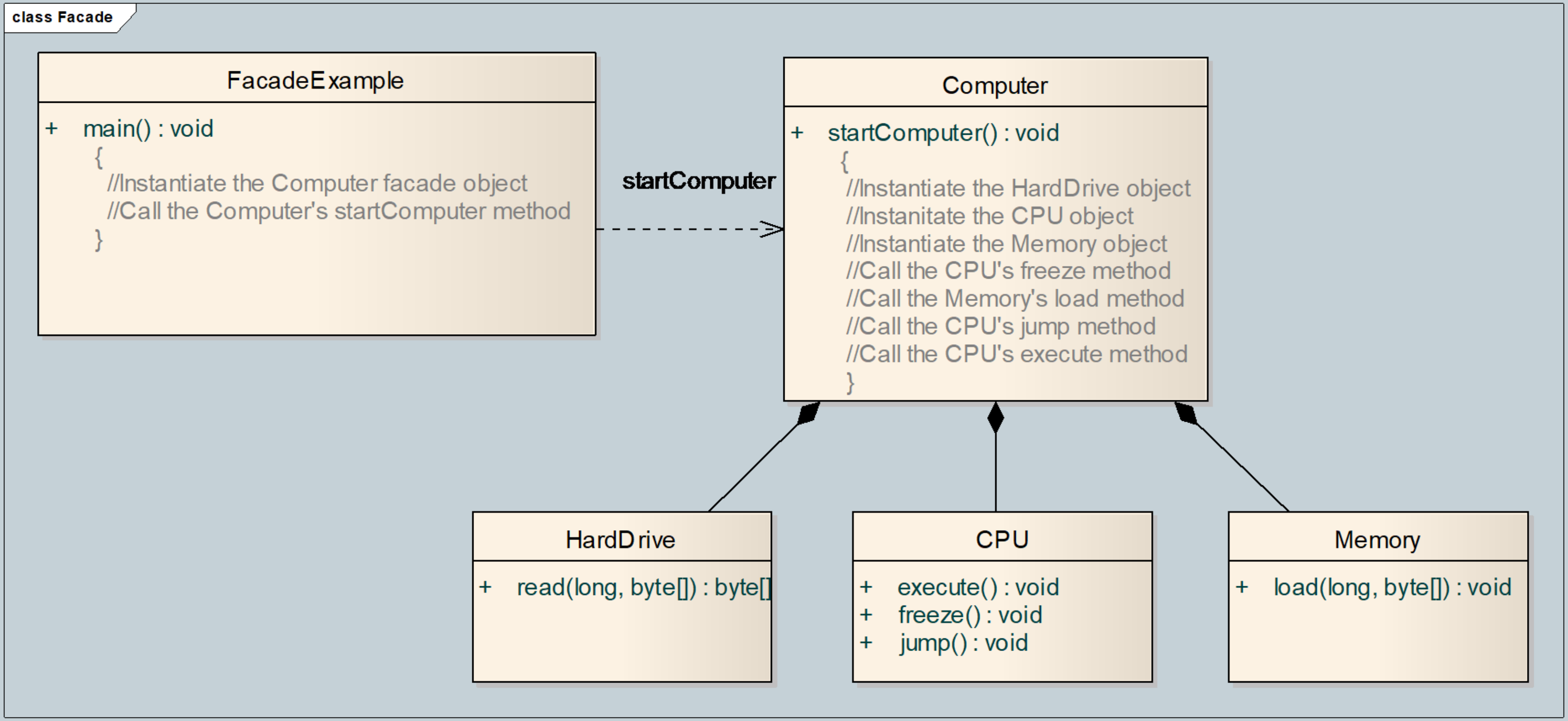
Façade - Class diagram



Façade - Problem



Façade - Solution



Facade

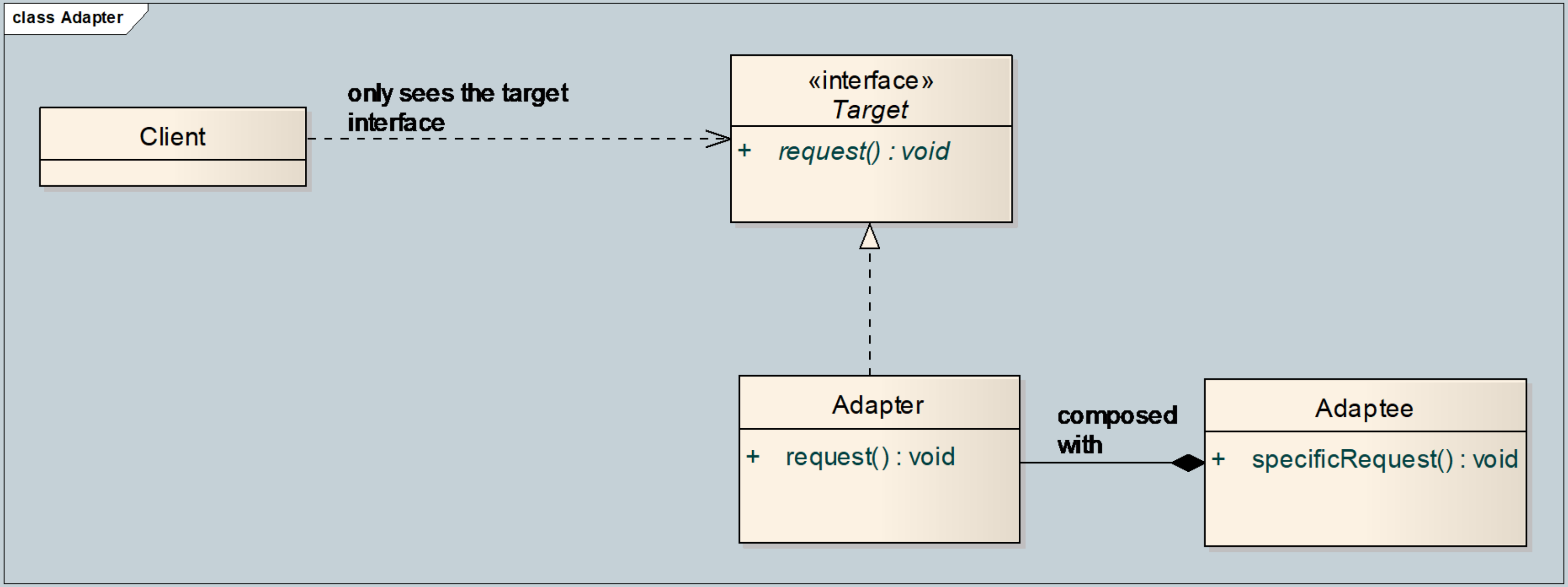
- Pros
  - Makes code easier to use and understand
  - Reduces dependencies on classes
  - Decouples a client from a complex system

- Cons
  - Results in more rework for improperly designed Façade class
  - Increases complexity and decreases runtime performance for large number of Façade classes

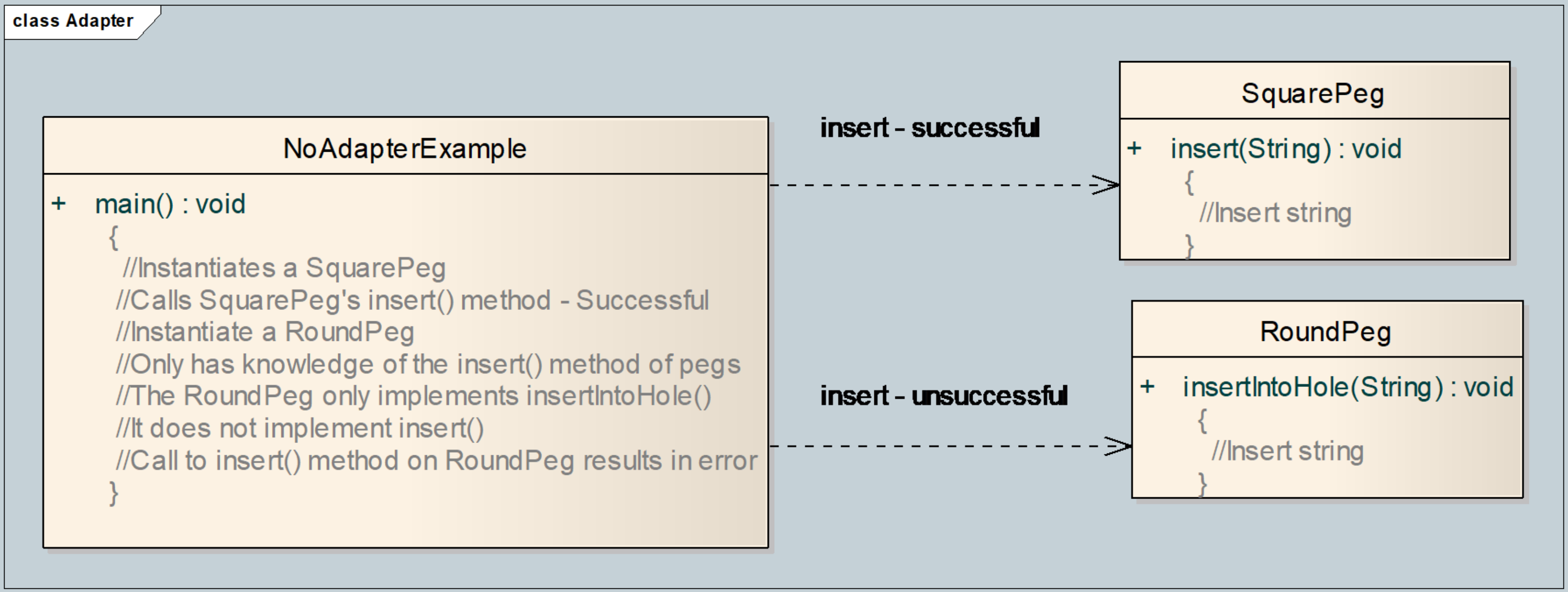
Adapter Definition

Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

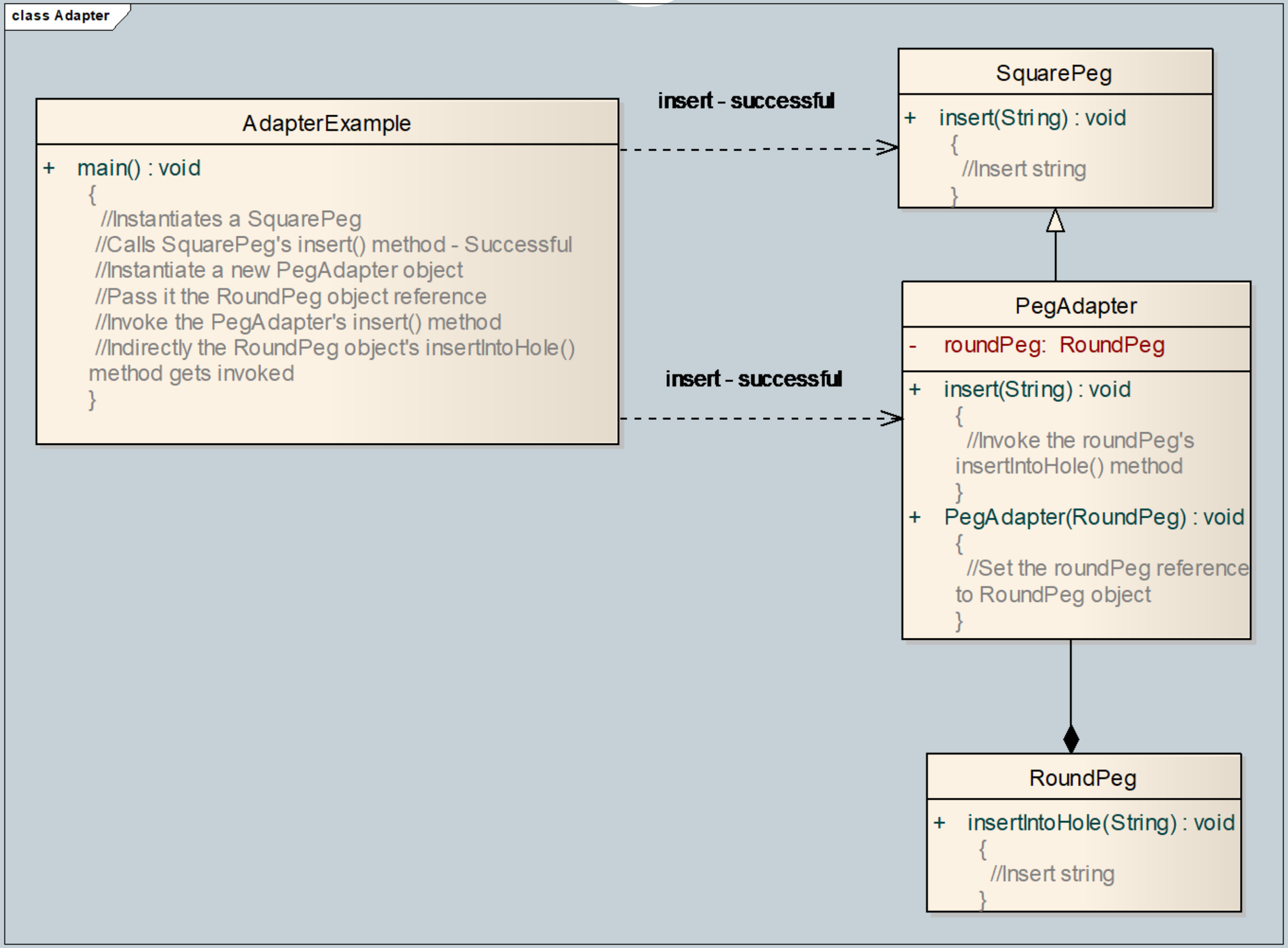
Adapter - Class diagram



Adapter - Problem



Adapter - Solution



Adapter

- Pros
  - Increases code reuse
  - Encapsulates the interface change
  - Handles legacy code

- Cons
  - Increases complexity for large number of changes

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Patterns & Definitions - Group 2

- Decorator
- Proxy
- Façade
- Adapter

- Simplifies the interface of a set of classes
- Wraps an object and provides an interface to it
- Wraps an object to provide new behavior
- Wraps an object to control access to it

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral
- Creational

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral
- Creational
- Structural

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral
- Creational
- Structural
- Structural

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral
- Creational
- Structural
- Structural
- Structural

Pattern Classification

- Strategy
- Observer
- Singleton
- Decorator
- Proxy
- Façade
- Adapter

- Behavioral
- Behavioral
- Creational
- Structural
- Structural
- Structural
- Structural

Conclusion - Design Principles

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface, not an implementation
- Favor composition over inheritance
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Principle of least knowledge – talk only to your immediate friends

References

- Design Patterns: Elements of Reusable Object-Oriented Software. Gamma, Helm, Johnson, and Vlissides (GoF). Addison-Wesley, 1995.
- Head First Design Patterns. Freeman and Freeman. O'REILLY, 2004.
- Design Patterns Explained. Shalloway and Trott. Addison-Wesley, 2002.
- Patterns of Enterprise Application Architecture. Fowler. Addison-Wesley, 2003.
- Core J2EE Pattern Catalog, Sun Developer Network, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
- Object Oriented Software Construction. Meyer, Prentice Hall, 1988.

References

- Wikipedia, The Free Encyclopedia
  - http://en.wikipedia.org/wiki/Singleton_pattern
  - http://en.wikipedia.org/wiki/Observer_pattern
  - http://en.wikipedia.org/wiki/Strategy_pattern
  - http://en.wikipedia.org/wiki/Decorator_pattern
  - http://en.wikipedia.org/wiki/Design_Patterns
  - http://en.wikipedia.org/wiki/Anti-pattern
  - http://en.wikipedia.org/wiki/Open/closed_principle
  - http://c2.com/ppr/wiki/JavaIdioms/JavaIdioms.html

Thank You!

CS 425/625 Software Engineering

Project Management

Based on Chapter 5 of the textbook [SE-8]

Ian Sommerville, Software Engineering, 8^{th} Ed., Addison-Wesley, 2006

and on Ch5 PPT presentation from http://www.software-engin.com/

Outline

- Introduction
- Project Planning
- Project Scheduling
- Risk Management

Introduction.

- *Software project management* is aimed to ensure that the software is delivered on time, within budget and schedule constraints, and satisfies the requirements of the client
- Management of software projects is different from other types of management because:
  - Software is not tangible
  - Software processes are relatively new and still “under trial”
  - Larger software projects are usually “one-off” projects
  - Computer technology evolves very rapidly

.Introduction

- Management activities:
  - Writing proposals
  - Planning the project
  - Scheduling the project
  - Estimating the cost of the project
  - Monitoring and reviewing the project's progress
  - Selecting, hiring, and evaluating personnel
  - Writing reports and giving presentations

Project Planning…

- A *project plan* should be drawn at the start of the project. This plan drives the project and needs to be continuously adjusted
- The role of the project manager is to anticipate possible problems and be prepared with solutions for these problems
- Other plans that need be developed:
  - Quality plan
  - Validation and verification plan
  - Configuration management plan
  - Maintenance plan
  - Staff development plan

.Project Planning..

- The planning process [Fig 5.2, SE-8]

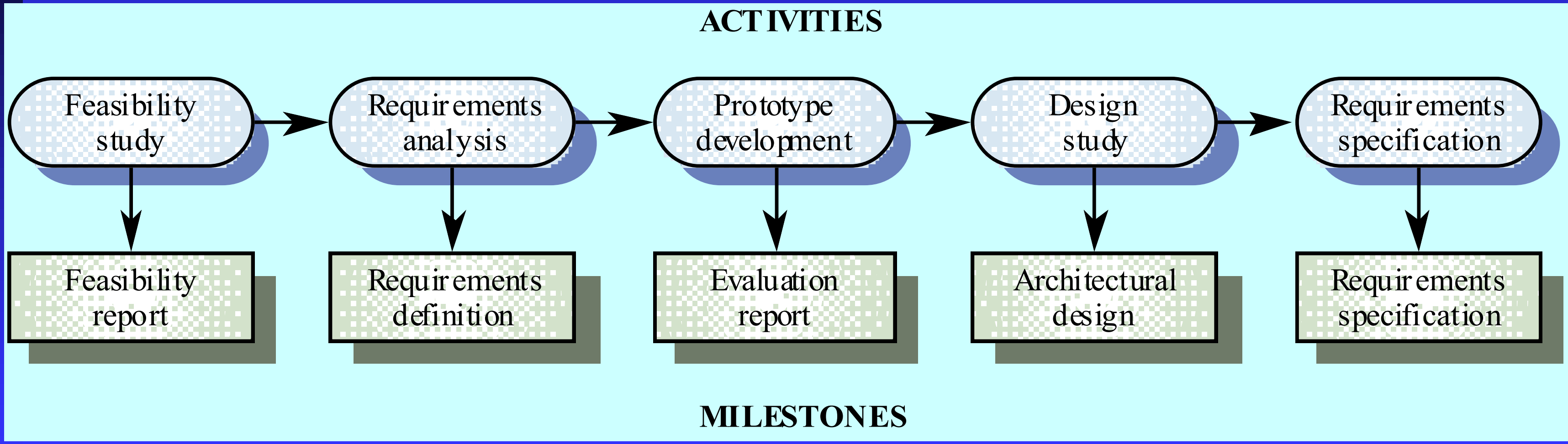
```
Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled   loop
        Draw up project schedule
        Initiate activities according to schedule
        Wait ( for a while )
        Review project progress
        Revise estimates of project parameters
        Update the project schedule
        Re-negotiate project constraints and deliverables
        if ( problems arise ) then
              Initiate technical review and possible revision
        end if
end loop
```

..Project Planning.

- The structure of the project plan:
  - Introduction (objectives, constraints)
  - Project organization (team structure, personnel involved, roles)
  - Risk analysis (types of risk, probabilities, solutions to prevent or reduce the risk)
  - Hardware and software resources needed (prices, delivery schedule)
  - Work breakdown (activities, milestones, deliverables)
  - Project schedule (dependencies between activities/tasks, work assignments, time allocated per task)
  - Monitoring and reporting mechanisms (reports, dates)

…Project Planning

- *Milestone* = end-point of a specific, distinct software process activity or task (for each milestone a report should be presented to the management)
- *Deliverable* = project result delivered to the client
- In order to establish milestones the phases of the software process need be divided in basic activities/tasks. Example for requirements engineering [Fig. 5.3, SE-8]

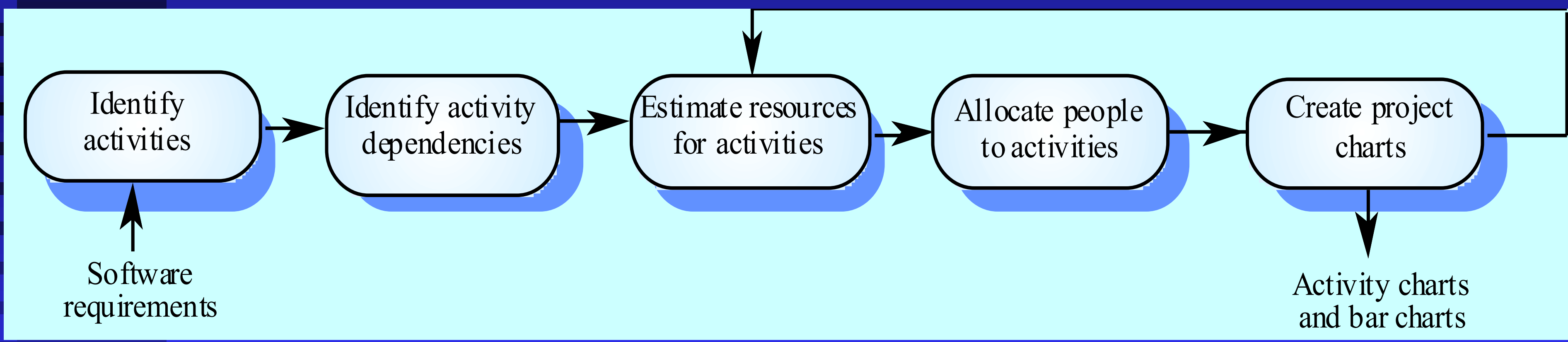


Project Scheduling……

- Software managers:
  - Divide the project in activities/tasks
  - Estimate time and resources needed to finish the project
  - Allocate resources to tasks
  - Try to employ efficiently all the project personnel
  - Minimize dependencies between tasks and teams
  - Prepare contingency plans
  - Rely on experience and intuition

.Project Scheduling…..

- The scheduling process [Fig. 5.4, SE-8]



..Project Scheduling….

- Graphical notations used in software project scheduling:
  - Tables: summary description of tasks
  - *Bar charts*: show schedule against the time
  - *Activity charts*: graphs that depict dependencies between tasks and indicate the *critical path* (the longest path in the activity graph)

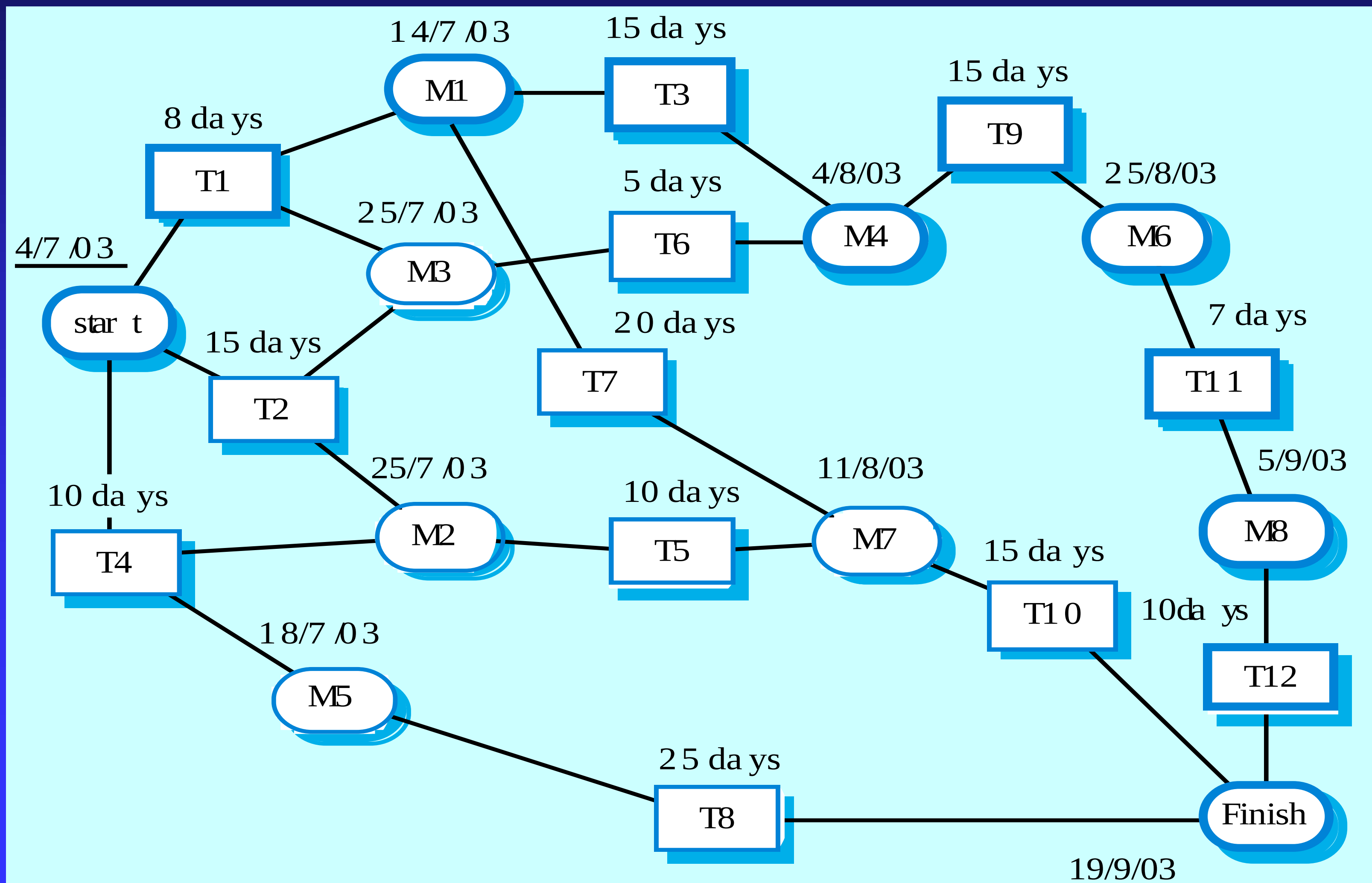
…Project Scheduling…

- Example of tabular description [Fig. 5.5, SE-8]:

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

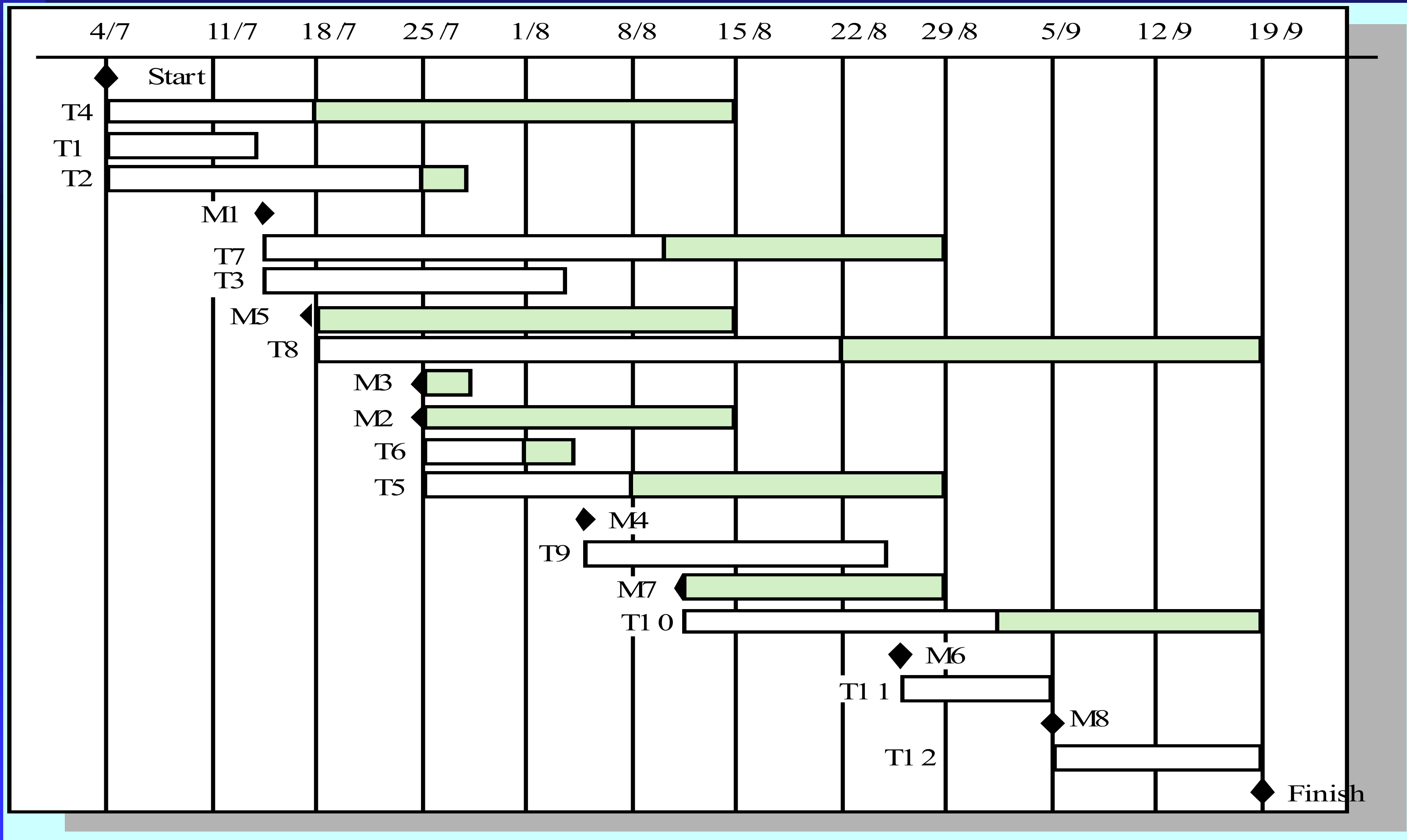
….Project Scheduling..

- Example of activity chart [Fig. 5.6, SE-8]



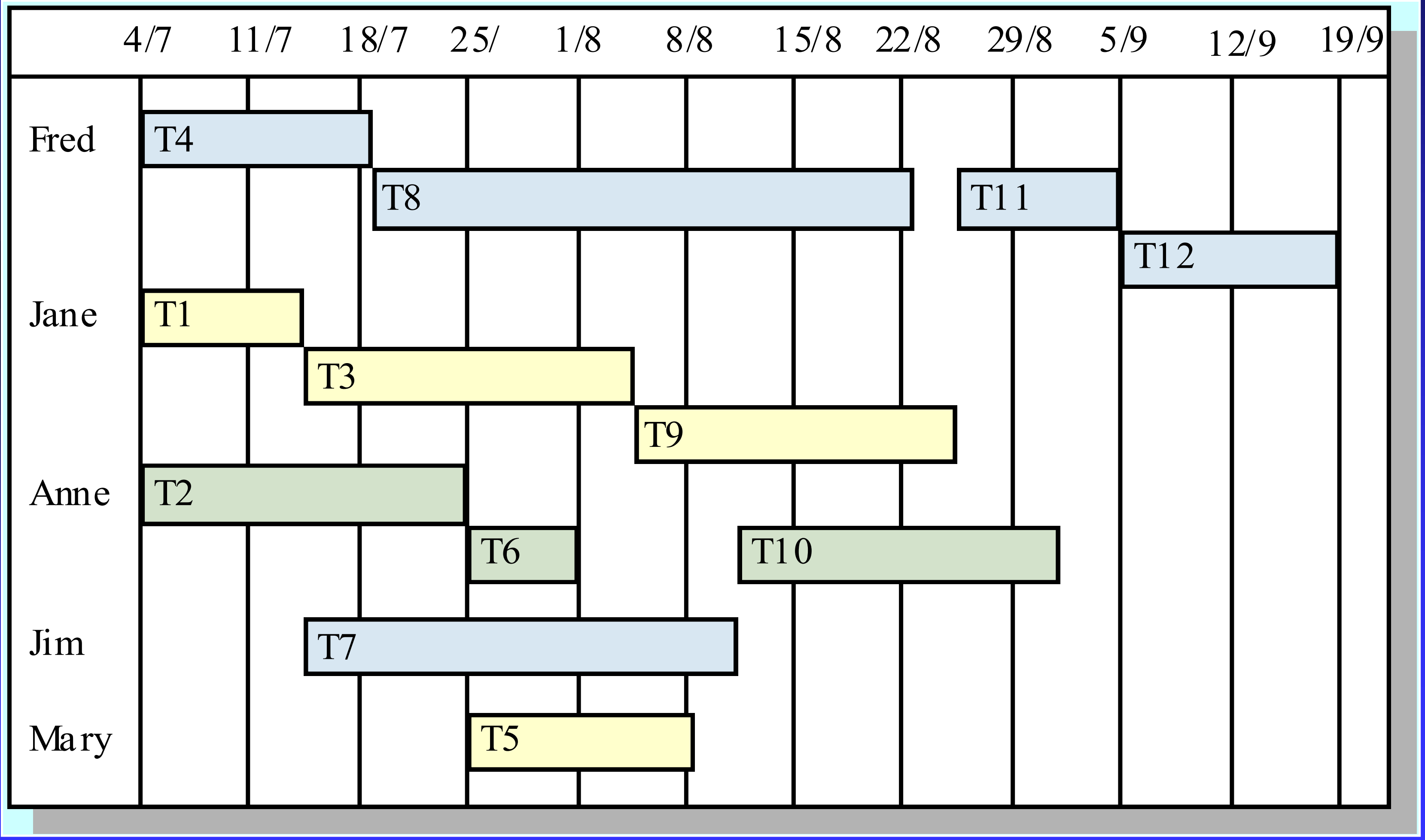
.....Project Scheduling.

- Example of bar chart [Fig. 5.7, SE-8]



……Project Scheduling

- Staff allocation chart [Fig. 5.8, SE-8]



Risk Management…….

- *Risk* = some adverse circumstance that may happen and affect negatively the project, the product, and/or the business
- Categories of risk:
  - Project risks
  - Product risks
  - Business risks
- *Risk management* means anticipating risks and preparing plans to reduce their effect

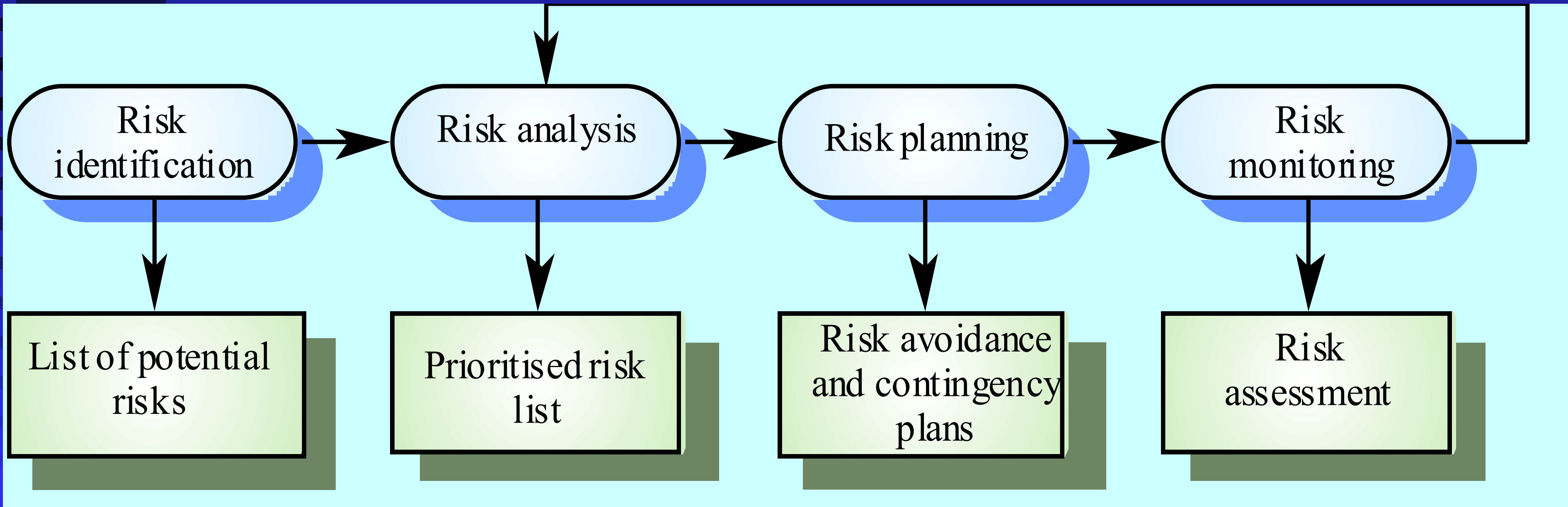
.Risk Management……

- Examples of risks in the software process [Fig. 5.9, SE-8]

Risk	**Affects**	**Description**
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

..Risk Management…..

- The risk management activities [Fig. 5.10, SE-8]



…Risk Management….

- Types of risk in *risk identification* [Fig. 5.11, SE-8]

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team member, job availability
Organisational	Organisational gossip, lack of action by senior management
Tools	Reluctance by team members to use tools, complaints about CASE tools, demands for higher-powered workstations
Requirements	Many requirements change requests, customer complaints
Estimation	Failure to meet agreed schedule, failure to clear reported defects

….Risk Management…

- *Risk analysis*:
  - Estimate risk probability:
    - Very low (< 10%)
    - Low (10-25%)
    - Moderate (25-50%)
    - High (50-75%)
    - Very high (> 75%)
  - Establish risk seriousness:
    - Insignificant
    - Tolerable
    - Serious
    - Catastrophic

…..Risk Management..

- *Risk planning* means preparing a strategy to deal with each of the risks identified
- Classes of strategies:
  - Avoidance strategies: the probability of the risk will be diminished
  - Minimization strategies: the effect of the risk will be reduced
  - Contingency strategies: plans for the worst case scenarios

……Risk Management.

- Examples of risk management strategies [Fig. 5.13, SE-8]

Risk	Strategy
Organisational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate buying-in components.
Staff illness	Reorganise team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.

Risk	Strategy
Requirements changes	Derive traceability information to assess requirements change impact, maximise information hiding in the design.
Organisational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying in components, investigate use of a program generator

…….Risk Management

- *Risk monitoring*:
  - Frequently re-assess the risks
    - Changes in risk probability?
    - Changes in risk gravity?
  - Take into consideration risk factors
  - Discuss key risks at each management project progress meeting

Software Project Management

Software Cost Estimation

Software Cost Estimation

- Predicting the resources required for a software development process

Software cost components

- Hardware and software costs
- Travel and training costs
- Effort costs (the dominant factor in most projects)
  - salaries of engineers involved in the project
  - Social and insurance costs
- Effort costs must take overheads into account
  - costs of building, heating, lighting
  - costs of networking and communications
  - costs of shared facilities (e.g library, staff restaurant, etc.)

Costing and pricing

- Estimates are made to discover the cost, to the developer, of producing a software system
- There is not a simple relationship between the development cost and the price charged to the customer
- Broader organisational, economic, political and business considerations influence the price charged

Software pricing factors

Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the opportunity of more profit later. The experience gained may allow new products to be developed.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices may be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a small profit or break even than to go out of business.

Measurement problems

- Estimating the size of the measure
- Estimating the total number of programmer months which have elapsed
- Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate

Lines of code

- What's a line of code?
  - The measure was first proposed when programs were typed on cards with one line per card
  - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line
- What programs should be counted as part of the system?
- Assumes linear relationship between system size and volume of documentation

Function points

- Based on a combination of program characteristics
  - external inputs and outputs
  - user interactions
  - external interfaces
  - files used by the system
- A weight is associated with each of these
- The function point count is computed by multiplying each raw count by the weight and summing all values

Function points

- Function point count modified by complexity of the project
- FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language
  - LOC = AVC * number of function points
  - AVC is a language-dependent factor varying from 200-300 for assemble language to 2-40 for a 4GL
- FPs are very subjective. They depend on the estimator.
  - Automatic function-point counting is impossible

Estimation techniques

- There is no simple way to make an accurate estimate of the effort required to develop a software system
  - Initial estimates are based on inadequate information in a user requirements definition
  - The software may run on unfamiliar computers or use new technology
  - The people in the project may be unknown
- Project cost estimates may be self-fulfilling
  - The estimate defines the budget and the product is adjusted to meet the budget

Estimation techniques

- Algorithmic cost modelling
- Expert judgement
- Estimation by analogy
- Parkinson's Law
- Pricing to win

Expert judgement

- One or more experts in both software development and the application domain use their experience to predict software costs. Process iterates until some consensus is reached.
- Advantages: Relatively cheap estimation method. Can be accurate if experts have direct experience of similar systems
- Disadvantages: Very inaccurate if there are no experts!

Estimation methods

- Each method has strengths and weaknesses
- Estimation should be based on several methods
- If these do not return approximately the same result, there is insufficient information available
- Some action should be taken to find out more in order to make more accurate estimates
- Pricing to win is sometimes the only applicable method

Experience-based estimates

- Estimating is primarily experience-based
- However, new methods and technologies may make estimating based on experience inaccurate
  - Object oriented rather than function-oriented development
  - Client-server systems rather than mainframe systems
  - Off the shelf components
  - Component-based software engineering
  - CASE tools and program generators

Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers
  - $\text{Effort} = A \times \text{Size}^B \times M$
  - A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes
- Most commonly used product attribute for cost estimation is code size
- Most models are basically similar but with different values for A, B and M

Estimation accuracy

- The size of a software system can only be known accurately when it is finished
- Several factors influence the final size
  - Use of COTS and components
  - Programming language
  - Distribution of system
- As the development process progresses then the size estimate becomes more accurate

The COCOMO model
Constructive Cost Estimation

- An empirical model based on project experience
- Well-documented, 'independent' model which is not tied to a specific software vendor
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2
- COCOMO 2 takes into account different approaches to software development, reuse, etc.

COCOMO 81

Project complexity	Formula	Description
Simple	$PM = 2.4\ (KDSI)^{1.05} \times M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0\ (KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6\ (KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

KDSI - Thousands of Delivered Source Instruction

COCOMO 2 levels

- COCOMO 2 is a 3 level model that allows increasingly detailed estimates to be prepared as development progresses
- Early prototyping level
  - Estimates based on object points and a simple formula is used for effort estimation
- Early design level
  - Estimates based on function points that are then translated to LOC
- Post-architecture level
  - Estimates based on lines of source code

Object point productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NOP/month)	4	7	13	25	50

Exponent scale factors

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience, Extra high means that the organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client only sets general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis, Extra high means a complete a thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions, Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organisation. The computation of this value depends on the CMM Maturity Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Multipliers

- Product attributes
  - concerned with required characteristics of the software product being developed
- Computer attributes
  - constraints imposed on the software by the hardware platform
- Personnel attributes
  - multipliers that take the experience and capabilities of the people working on the project into account.
- Project attributes
  - concerned with the particular characteristics of the software development project

Software Architecture and Design

tutorialspoint

SIMPLYEASYLEARNING

About the Tutorial

Software Architecture typically refers to the bigger structures of a software system and it deals with how multiple software processes cooperate to carry out their tasks. **Software Design** refers to the smaller structures and it deals with the internal design of a single software process.

By the end of this tutorial, the readers will develop a sound understanding of the concepts of software architecture and design concepts and will be in a position to choose and follow the right model for a given software project.

Audience

This tutorial is designed for all software professionals, architects, and senior system design engineers. Managers of architecture teams will also benefit from this tutorial.

Prerequisites

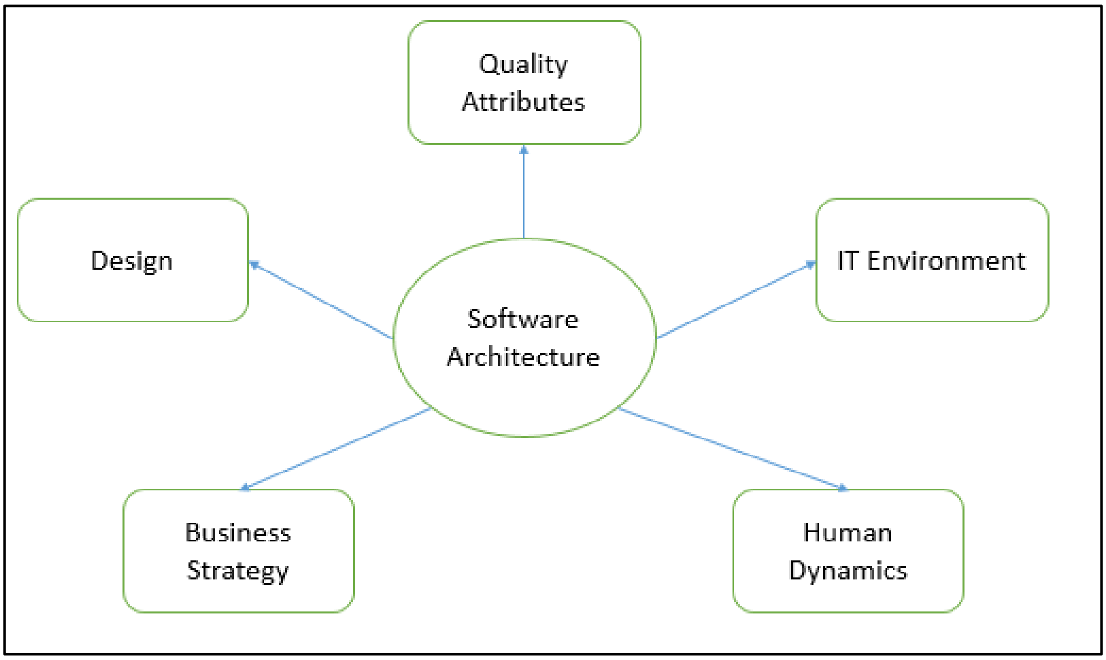
There are no exact prerequisites for this tutorial. Any software professional can go through this tutorial to get a bigger picture of how high quality software applications and products are designed.

Copyright & Disclaimer

Table of Contents

1. Software Architecture and Design – Introduction

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design is a process that includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements. In **Design**, functional requirements are accomplished.

Software Architecture

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

- It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

- It involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of:

    - Selection of structural elements and their interfaces by which the system is composed.

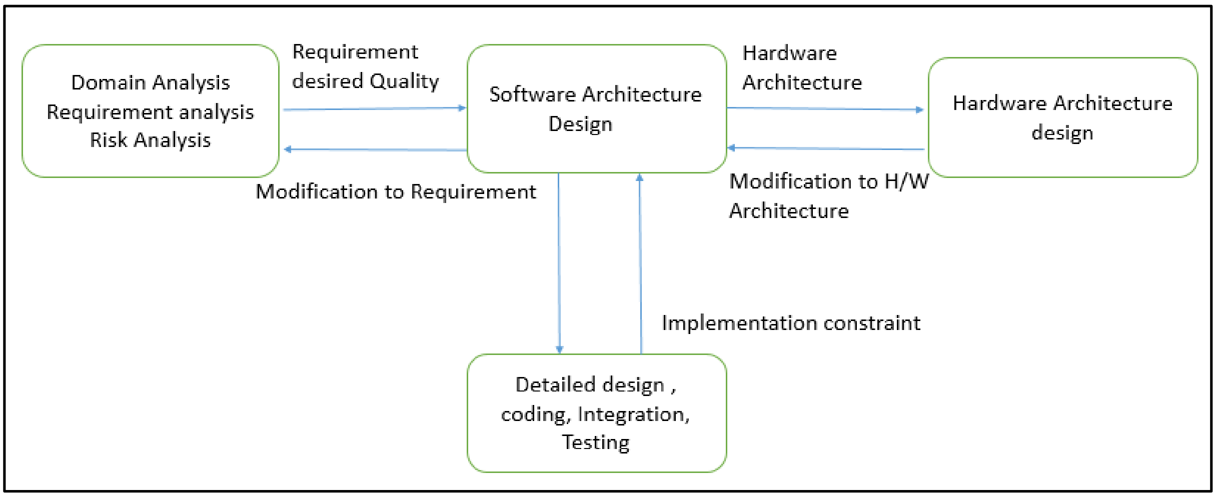
- Behavior as specified in collaborations among those elements.
- Composition of these structural and behavioral elements into large subsystem.
- Architectural decisions align with business objectives.
- Architectural styles that guide the organization.

Software Design

Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows:

- To negotiate system requirements, and to set expectations with customers, marketing and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.

It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.



Goals of Architecture

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements. Some of the other goals are as follows:

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.

- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization's market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

Limitations

Software architecture is still an emerging discipline within software engineering. It has the following limitations:

- Lack of tools and standardized ways to represent architecture
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development
- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design

Role of Software Architect

A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas:

Design Expertise

- Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.
- Lead the development team and coordinate the development efforts for the integrity of the design.
- Should be able to review design proposals and tradeoffs among them.

Domain Expertise

- Expert on the system being developed and plan for software evolution.
- Assist in the requirement investigation process assuring completeness and consistency.
- Coordinate the definition of domain model for the system being developed.

Technology Expertise

- Expert on available technologies that helps in the implementation of the system.
- Coordinate the selection of programming language, framework, platforms, databases, etc.

Methodological Expertise

- Expert on software development methodologies that may be adopted during SDLC (Software Development Life Cycle).
- Choose the appropriate approaches for development that helps the entire team.

Hidden Role of Software Architect

- Facilitates the technical work among team members and reinforcing the trust relationship in the team.
- Information specialist who shares knowledge and has vast experience.
- Protect the team members from external forces that would distract them and bring less value to the project.

Deliverables of the Architect

- A clear, complete, consistent, and achievable set of functional goals
- A functional description of the system, with at least two layers of decomposition
- A concept for the system
- A design in the form of the system, with at least two layers of decomposition
- A notion of the timing, operator attributes, and the implementation and operation plans
- A document or process which ensures functional decomposition is followed, and the form of interfaces is controlled

Quality Attributes

Quality is a measure of excellence or the state of being free from deficiencies or defects. Quality attributes are system properties that are separate from the functionality of the system.

Implementing quality attributes makes it is easier to differentiate a good system from a bad one. Attributes are overall factors that affect runtime behavior, system design, and user experience. They can be classified as follows:

Static Quality Attributes

Reflect the structure of system and organization, directly related to architecture, design, source code. They are invisible to end-user, but affect the development and maintenance cost, e.g.: modularity, testability, maintainability, etc.

Dynamic Quality Attributes

Reflect the behavior of the system during its execution. They are directly related to system's architecture, design, source code and also the configuration, deployment parameters, environment, and platform.

They are visible to the end-user and exist at runtime, e.g.: throughput, robustness, scalability, etc.

Quality Scenarios

Quality scenarios specify how to prevent a fault from becoming a failure. They can be divided into six parts based on their attribute specifications:

- **Source** An internal or external entity such as people, hardware, software, or physical infrastructure that generates the stimulus.
- **Stimulus** A condition that needs to be considered when it arrives on a system.
- **Environment** The stimulus occurs within certain conditions.
- **Artifact** A whole system or some part of it such as processors, communication channel, persistent storage, processes etc.
- **Response** An activity undertaken after the arrival of stimulus such as detect faults, recover from fault, disable event source etc.
- **Response measure** Should measure the occurred responses so that the requirements can be tested.

Common Quality Attributes

The following table lists the common quality attributes a software architecture must have.

Category	Quality Attribute	Description
Design Qualities	Conceptual Integrity	Defines the consistency and coherence of the overall design. This includes the way components or modules are designed
	Maintainability	Ability of the system to undergo changes with a degree of ease.
	Reusability	Defines the capability for components and subsystems to be suitable for use in other applications

Run-time Qualities	Interoperability	Ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties.
	Manageability	Defines how easy it is for system administrators to manage the application
	Reliability	Ability of a system to remain operational over time
	Scalability	Ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged.
	Security	Capability of a system to prevent malicious or accidental actions outside of the designed usages
	Performance	Indication of the responsiveness of a system to execute any action within a given time interval.
	Availability	Defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period.
System Qualities	Supportability	Ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
	Testability	Measure of how easy it is to create test criteria for the system and its components
User Qualities	Usability	Defines how well the application meets the requirements of the user and consumer by being intuitive
Architecture Quality	Correctness	Accountability for satisfying all the requirements of the system.
Non-runtime Quality	Portability	Ability of the system to run under different computing environment.
	Integrality	Ability to make separately developed components of the system work correctly together.
	Modifiability	Ease with which each software system can accommodate changes to its software.
Business quality attributes	Cost and schedule	Cost of the system with respect to time to market, expected project lifetime & utilization of legacy
	Marketability	Use of system with respect to market competition.

2. Key Principles

Software architecture is described as the organization of a system, where the system represents a collection of components that accomplish a specific set of functions.

Architectural Style

The **architectural style**, also called as **architectural pattern,** is a set of principles which shapes an application. It defines an abstract framework for a family of system in terms of the pattern of structural organization. The architectural style

- Provides a lexicon of components and connectors with rules on how they can be combined.
- Improves partitioning and allows the reuse of design by giving solutions to frequently occurring problems.
- Describes a particular way to configure a collection of components (a module with well-defined interfaces, reusable, replaceable) and connectors (communication link between modules).

The software that is built for computer-based systems exhibit one of many architectural styles. Each style describes a system category that encompasses:

- A set of component types which perform a required function by the system
- A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
- Semantic constraints which define how components can be integrated to form the system
- A topological layout of the components indicating their runtime interrelationships

Common Architectural Design

The following table lists architectural styles that can be organized by their key focus area.

Category	Architectural Design	Description
	Message bus	Prescribes use of a software system that can receive and send messages using one or more communication channels

Communication	Service–Oriented Architecture (SOA)	Defines the applications that expose and consume functionality as a service using contracts and messages.
Deployment	Client/server	Separate the system into two applications, where the client makes requests to the server
	3-tier or N-tier	Separates the functionality into separate segments with each segment being a tier located on a physically separate computer.
Domain	Domain Driven Design	Focused on modeling a business domain and defining business objects based on entities within the business domain.
Structure	Component Based	Breakdown the application design into reusable functional or logical components that expose well-defined communication interfaces.
	Layered	Divide the concerns of the application into stacked groups (layers).
	Object oriented	Based on the division of responsibilities of an application or system into objects, each containing the data and the behavior relevant to the object.

Types of Architecture

There are four types of architecture from the viewpoint of an enterprise and collectively, these architectures are referred to as **enterprise architecture**.

- **Business architecture**: Defines the strategy of business, governance, organization, and key business processes within an enterprise and focuses on the analysis and design of business processes.

- **Application (software) architecture**: Serves as the blueprint for individual application systems, their interactions, and their relationships to the business processes of the organization

- **Information architecture**: Defines the logical and physical data assets and data management resources.

- **Information technology (IT) architecture**: Defines the hardware and software building blocks that make up the overall information system of the organization.

Architecture Design Process

The architecture design process focuses on the decomposition of a system into components and their interactions to satisfy functional and nonfunctional requirements. The key inputs to software architecture design are:

- The requirements produced by the analysis tasks
- The hardware architecture (the software architect in turn provides requirements to the system architect, who configures the hardware architecture)

The result or output of the architecture design process is an **architectural description**. The basic architecture design process is composed of the following steps:

Understand the Problem

- This is the most crucial step because it affects the quality of the design that follows.
- Without a clear understanding of the problem, it is not possible to create an effective solution.
- Many software projects and products are considered failures because they did not actually solve a valid business problem or have a recognizable return on investment (ROI).

Identify Design Elements and their Relationships

- In this phase, build a baseline for defining the boundaries and context of the system.
- Decomposition of the system into its main components based on functional requirements. The decomposition can be modeled using a design structure matrix (DSM), which shows the dependencies between design elements without specifying the granularity of the elements.
- In this step, the first validation of the architecture is done by describing a number of system instances and this step is referred as functionality based architectural design.

Evaluate the Architecture Design

- Each quality attribute is given an estimate so in order to gather qualitative measures or quantitative data, the design is evaluated.
- It involves evaluating the architecture for conformance to architectural quality attributes requirements.
- If all estimated quality attributes are as per the required standard, the architectural design process is finished.
- If not, the third phase of software architecture design is entered: architecture transformation. If the observed quality attribute does not meet its requirements, then a new design must be created.

Transform the Architecture Design

- This step is performed after an evaluation of the architectural design. The architectural design must be changed until it completely satisfies the quality attribute requirements.

- It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality.

- A design is transformed by applying design operators, styles, or patterns. For transformation, take the existing design and apply design operator such as decomposition, replication, compression, abstraction, and resource sharing.

- The design is again evaluated and the same process is repeated multiple times if necessary and even performed recursively.

- The transformations (i.e. quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively

Key Architecture Principles

Following are the key principles to be considered while designing an architecture:

Build to Change Instead of Building to Last.

Consider how the application may need to change over time to address new requirements and challenges, and build in the flexibility to support this.

Reduce Risk and Model to Analyze

Use design tools, visualizations, modeling systems such as UML to capture requirements and design decisions. The impacts can also be analyzed. Do not formalize the model to the extent that it suppresses the capability to iterate and adapt the design easily.

Use Models and Visualizations as a Communication and Collaboration Tool

Efficient communication of the design, the decisions, and ongoing changes to the design is critical to good architecture. Use models, views, and other visualizations of the architecture to communicate and share the design efficiently with all the stakeholders. This enables rapid communication of changes to the design.

Identify and understand key engineering decisions and areas where mistakes are most often made. Invest in getting key decisions right the first time to make the design more flexible and less likely to be broken by changes.

Use an Incremental and Iterative Approach

Start with baseline architecture and then evolve candidate architectures by iterative testing to improve the architecture. Iteratively add details to the design over multiple passes to get the big or right picture and then focus on the details.

Key Design Principles

Following are the design principles to be considered for minimizing cost, maintenance requirements, and maximizing extendibility, usability of architecture

Separation of Concerns

Divide the components of system into specific features so that there is no overlapping among the components functionality. This will provide high cohesion and low coupling. This approach avoids the interdependency among components of system which helps in maintaining the system easy.

Single Responsibility Principle

Each and every module of system should have one specific responsibility which helps the user to clearly understand the system. It should also help with integration of the component with other components.

Principle of Least Knowledge

Any component or object should not have knowledge about internal details of other components. This approach avoids interdependency and helps maintainability.

Minimize Large Design Upfront

Minimize large design upfront if the requirements of an application are unclear. If there is a possibility of modifying requirements, then avoid making a large design for whole system.

Do not Repeat the Functionality

Do not repeat functionality specifies that functionality of components should not to be repeated and hence a piece of code should be implemented in one component only. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.

Prefer Composition over Inheritance while Reusing the Functionality

Inheritance creates dependency between children and parent classes and hence it blocks the free use of the child classes. In contrast, the composition provides a great level of freedom and reduces the inheritance hierarchies.

Identify Components and Group them in Logical Layers

Identity components and the area of concern that are needed in system to satisfy the requirements. Then group these related components in logical layer which will help the user to understand the structure of the system at a high level. Avoid mixing components of different type of concerns in same layer.

Define the Communication Protocol between Layers

Understand how components will communicate with each other which requires a complete knowledge of deployment scenarios and the production environment.

Define Data Format for a Layer

Various components will interact with each other through data format. Do not mix the data formats so that applications are easy to implement, extend and maintain.
Try to keep data format same for a layer, so that various components need not code/decode the data while communicating with each other. It reduces a processing overhead.

System Service Components should be Abstract

Code related to security, communications, or system services like logging, profiling, and configuration etc. should be abstracted in separate components. Do not mix this code with business logic, as it is easy to extend design and maintain it.

Design Exceptions and Exception Handling Mechanism

Defining exceptions in advance helps the components to manage errors or unwanted situation in an elegant manner. The exception management will be same throughout the system.

Naming Conventions

Naming conventions should be defined in advance. They provide a consistent model that help the users to understand the system easily. It is easier for team members to validate code written by others, and hence will increase the maintainability.

3. Architecture Models

Software architecture specifies the high level structure of software system abstraction, by using decomposition and composition, with architectural style and quality attributes. A software architecture design must conform to the major functionality and performance requirements of the system, as well as satisfy the non-functional requirements such as reliability, scalability, portability, and availability.

A software architecture must describe its group of components, their connections, interactions among them and deployment configuration of all components.

A software architecture can be defined in many ways:

- **UML (Unified Modeling Language):** UML is one of object-oriented solutions used in software modeling and design.

- **Architecture View Model (4+1 view model):** Architecture view model represents the functional and non-functional requirements of software application.

- **ADL (Architecture Description Language):** ADL defines the software architecture formally and semantically.

UML

UML stands for Unified Modeling Language. It is a pictorial language used to make software blueprints. UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. It serves as a standard for software requirement analysis and design documents which are the basis for developing a software.

UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document a software system. Although UML is generally used to model software system, it is not limited within this boundary. It is also used to model non software systems like process flows in a manufacturing unit.

The elements are like components which can be associated with different ways to make a complete UML picture which is known as a **diagram**. So it is very important to understand the different diagrams to implement the knowledge in real-life systems. We have two broad categories of diagrams and they are again divided into sub-categories: **Structural Diagrams** and **Behavioral Diagrams**.

Structural Diagrams

Structural diagrams represent the static aspect of a system. These static aspects represent those parts of a diagram which forms the main structure and is therefore stable.

These static parts are represented by classes, interfaces, objects, components and nodes. Structural diagrams can be sub-divided as follows:

- Class diagram

- Object diagram
- Component diagram
- Deployment diagram
- Package diagram
- Composite structure

The following table provides a brief description of these diagrams:

Diagram	Description
Class	Represents the object orientation of a system. Shows how classes are statically related
Object	Represents a set of objects and their relationships at runtime and also represent the static view of the system
Component	Describes all the components, their interrelationship, interactions and interface of the system
Composite structure	Describes inner structure of component including all classes, interfaces of the component etc
Package	Describes the package structure and organization. Covers classes in the package and packages within another package
Deployment	Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed

Behavioral Diagrams

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspects are basically the changing/moving parts of a system. UML has the following types of behavioral diagrams:

- Use case diagram
- Sequence diagram
- Communication diagram
- State chart diagram
- Activity diagram
- Interaction overview
- Time sequence diagram

The following table provides a brief description of these diagrams:

Diagram	Description
Use case	Describe the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.
Activity	Describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent or branched
State Machine/state chart	Represent the event driven state change of a system. It basically describes the state change of a class, interface, etc. Used to visualize the reaction of a system by internal/external factors.
Sequence	Visualize the sequence of calls in a system to perform a specific functionality.
Interaction Overview	Combines activity and sequence diagrams to provide a control flow overview of system and business process.
Communication	Same as sequence diagram except that it focuses on the object's role. Each communication is associated with a sequence order number plus the past messages.
Time Sequenced	Describes the changes by messages in state, condition and events.

Architecture View Model

A model is a complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint.

A view is a representation of an entire system from the perspective of a related set of concerns. A view is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers.

4+1 View Model

The 4+1 View Model was designed by Philippe Kruchten for describing the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a **multiple view** model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.

It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides 4 essential views:

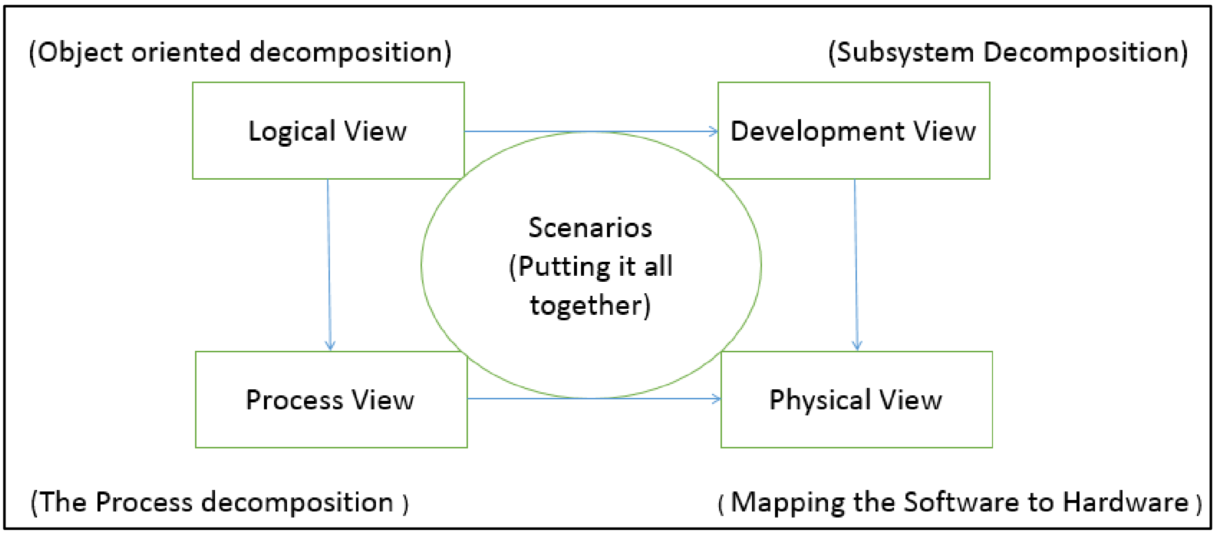
- **The logical view or conceptual view**: It describes the object model of the design.

- **The process view**: It describes the activities of the system, captures the concurrency and synchronization aspects of the design.

- **The physical view**: It describes the mapping of software onto hardware and reflects its distributed aspect.

- **The development view**: It describes the static organization or structure of the software in its development environment.

This view model can be extended by adding one more view called **scenario view** or **use case view** for end-users or customers of software systems. It is coherent with other four views and are utilized to illustrate the architecture serving as "plus one" view, (4+1) view model. The following figure describes the software architecture using five concurrent views (4+1) model.



Why is it called 4+1 instead of 5?

The **use case view** has a special significance as it details the high level requirement of system while other views details how those requirements are realized. When all other 4 views are completed, it's effectively redundant. However, all other views would not be possible without it. The following table shows the 4+1 view in detail:

	Logical	Process	Development	Physical	Scenario
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration
Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional requirements	Non Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Nonfunctional requirement regarding to underlying hardware	System Consistency and validity
UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

Architecture Description Languages (ADLs)

An ADL is a language that provides syntax and semantics for defining a software architecture. It is a notation specification which provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation.

ADLs must support the architecture components, their connections, interfaces, and configurations which are the building block of architecture description. It is a form of expression for use in architecture descriptions and provides the ability to decompose components, combine the components, and define the interfaces of components.

An architecture description language is a formal specification language which describes the software features such as processes, threads, data, and subprograms as well as hardware component such as processors, devices, buses, and memory.

It is hard to classify or differentiate an ADL and a programming language or a modeling language. However, there are some requirements for a language to be classified as an ADL:

- It should be appropriate for communicating the architecture to all concerned parties.
- It should be suitable for tasks of architecture creation, refinement, and validation.
- It should provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL.
- It should have the ability to represent most of the common architectural styles.
- It should support analytical capabilities or provide quick generating prototype implementations.

4. Object-Oriented Paradigm

The object-oriented (OO) paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later. OO analysis and design paradigm is the logical result of the wide adoption of OO programming languages.

- The first object–oriented language was **Simula** (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, **Alan Kay** and his research group at Xerox PARC created a personal computer named **Dynabook** and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.
- In the 1980s, **Grady Booch** published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object–oriented design method.
- In the 1990s, **Coad** incorporated behavioral ideas to object-oriented methods.

The other significant innovations were Object Modeling Techniques (OMT) by **James Rum Baugh** and Object-Oriented Software Engineering (OOSE) by **Ivar Jacobson**.

Introduction to Object-Oriented Paradigm

OO paradigm is a significant methodology for the development of any software. Most of the architecture styles or patterns such as pipe and filter, data repository and component-based etc. can be implemented using this paradigm.

Basic concepts and terminologies of object–oriented systems:

Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has:

- Identity that distinguishes it from other objects in the system.
- State that determines characteristic properties of an object as well as values of properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modeled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or the description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, an object is an instance of a class.

The constituents of a class are:

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.

- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows:

- x–coord, to denote x–coordinate of the center

- y–coord, to denote y–coordinate of the center

- a, to denote the radius of the circle

Some of its operations can be defined as follows:

- findArea(), a method to calculate area

- findCircumference(), a method to calculate circumference

- scale(), a method to increase or decrease the radius

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Example

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.Relationships

In order to describe a system, both dynamic (behavioral) and static (logical) specification of system must be provided. The dynamic specification describes the relationships among objects. e.g. message passing and static specification describe the relationships among classes, e.g. Aggregation, association and inheritance.

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

Composition or Aggregation

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a "part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Association

Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association. The Degree of an association denotes the number of classes involved in a connection. The degree may be unary, binary, or ternary.

- A unary relationship connects objects of the same class.
- A binary relationship connects objects of two classes.
- A ternary relationship connects objects of three or more classes.

Inheritance

It is a mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses.

The subclass can inherit or derive the attributes and methods of the super-class (es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines a "is – a" relationship.

Example

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow "is – a" mammal.

OO Analysis

In object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified. The aim of OO analysis is to understand the application domain and specific requirements of the system. The result of this phase is requirement specification and initial analysis of logical structure and feasibility of system.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modeling, dynamic modeling, and functional modeling.

Object Modeling

Object modeling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modeling can be visualized in the following steps:

- Identify objects and group into classes
- Identify the relationships among classes
- Create a user object model diagram
- Define a user object attributes
- Define the operations that should be performed on the classes

Dynamic Modeling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modeling.

Dynamic Modeling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world".

The process of dynamic modeling can be visualized in the following steps:

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct a dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modeling

Functional Modeling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modeling and the actions of dynamic modeling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modeling can be visualized in the following steps:

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify the constraints
- Specify optimization criteria

Object-Oriented Design

After the analysis phase, the conceptual model is developed further into an object-oriented model using object-oriented design (OOD). In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. The main aim of OO design is to develop the structural architecture of a system. The stages for object–oriented design can be identified as:

- Defining the context of the system
- Designing the system architecture
- Identification of the objects in the system
- Construction of design models
- Specification of object interfaces

OO Design can be divided into two stages: Conceptual design and Detailed design.

Conceptual design

In this stage, all the classes are identified that are needed for building the system and specific responsibilities are assigned to each class. Class diagram is used to clarify the relationships among classes, and interaction diagram are also used to show the flow of events. It is also known as **high-level design**.

Detailed design

In this stage, attributes and operations are assigned to each class based on their interaction diagram. State machine diagram are developed to describe the further details of design. It is also known as **low-level design**.

Design Principles

Principle of Decoupling It is difficult to maintain a system with a set of highly interdependent classes as modification in one class may result in cascading updates of other classes. In an OO design, tight coupling can be eliminated by introducing new classes or inheritance.

Ensuring Cohesion A cohesive class performs a set of closely related functions. A lack of cohesion means a class performs unrelated functions, although it does not affect the operation of the whole system. It makes the entire structure of software hard to manage, expand, maintain, and change.

Open-closed principle According to this principle, the system should be able to extend to meet new requirements. The existing implementation and the code of the system should not be modified as a result of a system expansion. In addition, the following guidelines have to be followed in open-closed principle:

- For each concrete class, separate interface and implementations have to be maintained.

- In a multithreaded environment, keep the attributes private.

- Minimize the use of global variables and class variables.

5. DATA FLOW Architecture

In data flow architecture, the whole software system is seen as a series of transformations on consecutive pieces or set of input data, where data and operations are independent of each other. In this approach, the data enters the system and then flows through the modules one at a time until they are assigned to some final destination (output or a data store).

The connections between the components or modules may be implemented as I/O stream, I/O buffers, piped or other types of connections. The data can be flown in the graph topology with cycles, in a linear structure without cycles, or in a tree type structure.

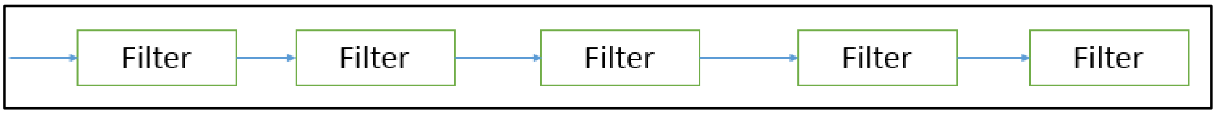
The main objective of this approach is to achieve the qualities of reuse and modifiability. It is suitable for applications that involve a well-defined series of independent data transformations or computations on orderly defined input and output such as compilers and business data processing applications. There are three types of execution sequences between modules:

- Batch sequential
- Pipe and filter or non-sequential pipeline mode
- Process control

Batch Sequential

Batch sequential is a classical data processing model in which a data transformation subsystem can initiate its process only after its previous subsystem is completely through.

- The flow of data carries a batch of data as a whole from one subsystem to another.
- The communications between the modules are conducted through temporary intermediate files which can be removed by successive subsystems.
- It is applicable for those applications where data is batched, and each subsystem reads related input files and writes output files.
- Typical application of this architecture includes business data processing such as banking and utility billing.



Advantages

- Provides simpler divisions on subsystems.

- Each subsystem can be an independent program working on input data and producing output data.

Disadvantages

- Provides high latency and low throughput.
- Does not provide concurrency and interactive interface.
- External control is required for implementation.

Pipe and Filter Architecture

This approach lays emphasis on the incremental transformation of data by successive component. In this approach, the flow of data is driven by data and the whole system is decomposed into components of data source, filters, pipes and data sinks.

The connections between modules are data stream which is first-in/first-out buffer that can be stream of bytes, characters or any other type. The main feature of this architecture is its concurrent and incremented execution.

Filter

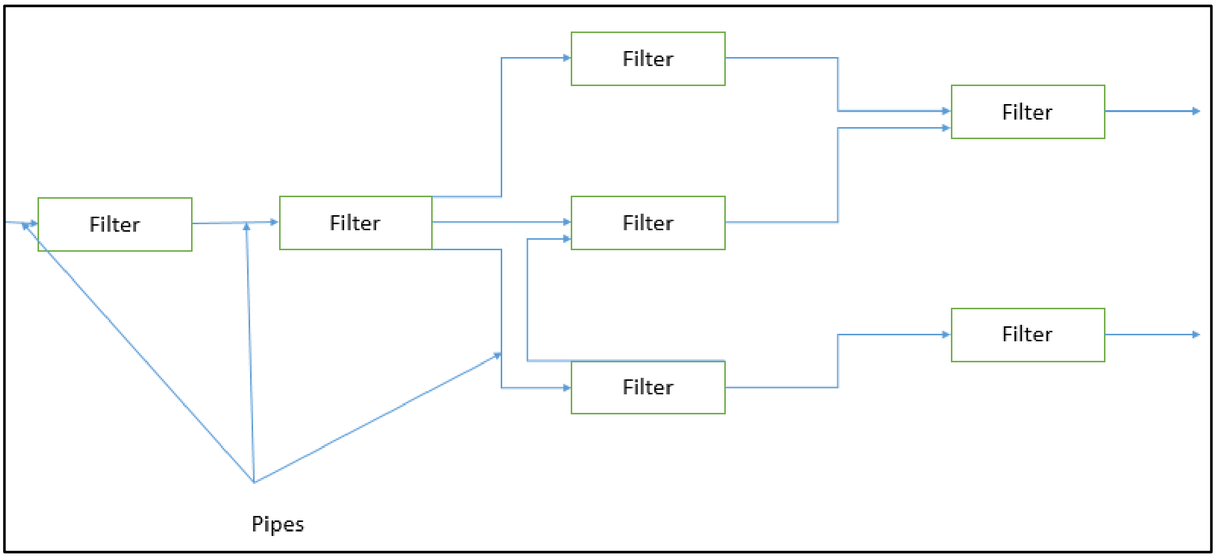
A filter is an independent data stream transformer or stream transducers. It transforms the data of input data stream, processes it, and writes the transformed data stream over a pipe for the next filter to process. It works in an incremental mode in which it start working as soon as data arrives through connected pipe. There are two types of filters: **active filter** and **passive filter**.

Active filter

Active filter lets connected pipes to pull data in and pushes out the transformed data. It operates with passive pipe which provide read/write mechanisms for pulling and pushing. This mode is used in UNIX pipe and filter mechanism.

Passive filter

Passive filter lets connected pipes to push data in and pull data out. It operates with active pipe which pulls data from a filter and pushes data into the next filter. It must provide read/write mechanism.



Advantages

- Provides concurrency and high throughput for excessive data processing.
- Provides reusability and simplifies system maintenance.
- Provides modifiability and low coupling between filters.
- Provides simplicity by offering clear divisions between any two filters connected by pipe.
- Provides flexibility by supporting both sequential and parallel execution.

Disadvantages

- Not suitable for dynamic interactions.
- A low common denominator is needed for transmission of data in ASCII formats.
- Overhead of data transformation between filters.
- Does not provide a way for filters to cooperatively interact to solve a problem.
- Difficult to configure this architecture dynamically.

Pipe

Pipes are stateless and they carry binary or character stream which exist between two filters. It can move a data stream from one filter to another. Pipes use little contextual information and retain no state information between instantiations.

Process Control Architecture

It is a type of data flow architecture where data is neither batched sequential nor pipelined stream. The flow of data comes from a set of variables which controls the execution of process. It decomposes the entire system into subsystems or modules and connects them.

Types of Subsystems

A process control architecture would have a **processing unit** for changing the process control variables and a **controller unit** for calculating the amount of changes.

A controller unit must have the following elements:

- **Controlled Variable** Controlled Variable values the system it intends to control and should be measured by sensors. For example, speed in cruise control system.
- **Input Variable** Measures an input to the process. For example, temperature of return air in temperature control system
- **Manipulated Variable** Manipulated Variable value is adjusted or changed by the controller.
- **Process Definition** It includes mechanisms for manipulating some process variables.
- **Sensor** Obtains values of process variables pertinent to control and can be used as a feedback reference to recalculate manipulated variables.
- **Set Point** It is the desired value for a controlled variable.
- **Control Algorithm** For deciding how to manipulate process variables.

Application Areas

Process control architecture is suitable in the following domains:

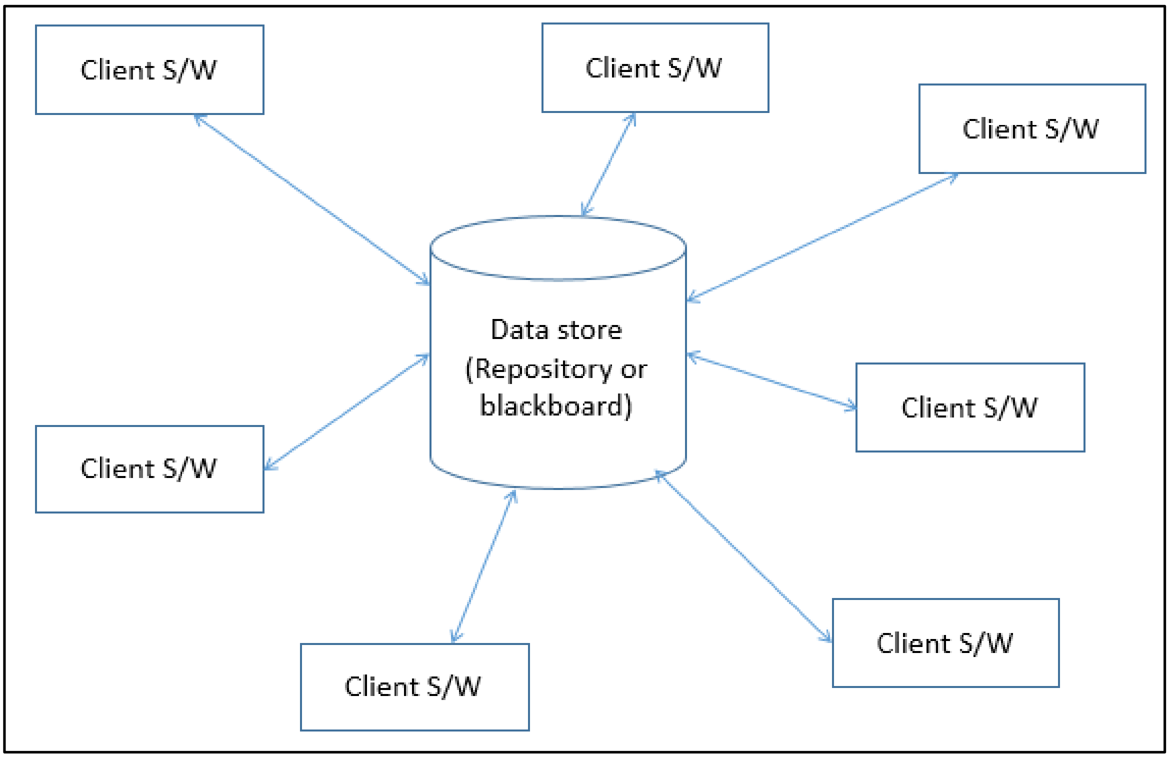
- Embedded system software design where the system is manipulated by process control variable data
- Applications whose aim is to maintain specified properties of the outputs of the process at given reference values.
- Applicable for car-cruise control and building temperature control systems.
- Real-time system software to control automobile anti-lock brakes, nuclear power plants, etc.

6. Data-Centered Architecture

In data-centered architecture, the data is centralized and accessed frequently by other components that modify data. The main purpose of this style is to achieve integrality of data. Data-centered architecture consists of different components that communicate through shared data repositories. The components access a shared data structure and are relatively independent, in that, they interact only through the data store.

The most well-known examples of the data-centered architecture is a database architecture in which the common database schema is created with data definition protocol – for example, a set of related tables with fields and data types in an RDBMS.

Another example of data-centered architectures is the web architecture which has a common data schema (i.e. meta-structure of the Web) and follows hypermedia data model and processes communicate through the use of shared web-based data services.



Types of Components

There are two types of components:

- A **central data** structure or data store or data repository which is responsible for providing permanent data storage. It represents the current state.

- A **data accessor** or a collection of independent components that operate on the central data store, perform computations, and might put back the results.

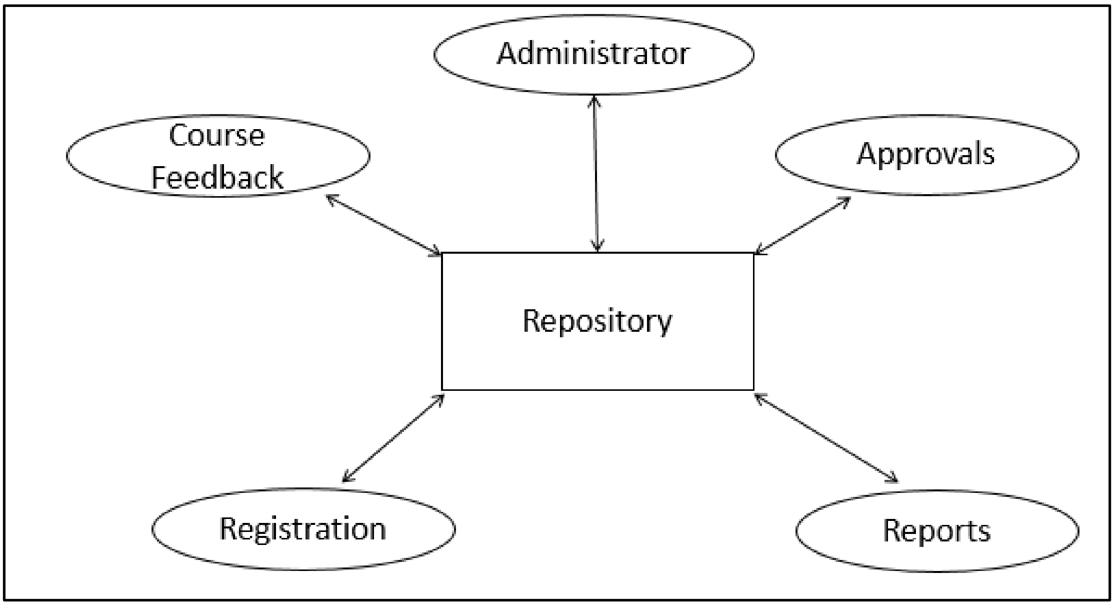
Interactions or communication between the data accessors is only through the data store. The data is the only means of communication among clients. The flow of control differentiates the architecture into two categories:

- Repository Architecture Style
- Blackboard Architecture Style

Repository Architecture Style

In Repository Architecture Style, the data store is passive and the clients (software components or agents) of the data store are active which control the logic flow. The participating components check the data-store for changes.

- The client sends a request to the system to perform actions (e.g**.** insert data).
- The computational processes are independent and triggered by incoming requests.
- If the types of transactions in an input stream of transactions trigger selection of processes to execute, then it is traditional database or repository architecture, or passive repository.
- This approach is widely used in DBMS, library information system, the interface repository in CORBA, compilers and CASE (computer aided software engineering) environments.



Advantages

- Provides data integrity, backup and restore features.
- Provides scalability and reusability of agents as they do not have direct communication with each other.

- Reduces overhead of transient data between software components.

Disadvantages

- It is more vulnerable to failure and data replication or duplication is possible.
- High dependency between data structure of data store and its agents.
- Changes in data structure highly affect the clients.
- Evolution of data is difficult and expensive.
- Cost of moving data on network for distributed data.

Blackboard Architecture Style

In Blackboard Architecture Style, the data store is active and its clients are passive. Therefore the logical flow is determined by the current data status in data store. It has a blackboard component, acting as a central data repository, and an internal representation is built and acted upon by different computational elements.

- A number of components that act independently on the common data structure are stored in the blackboard.
- In this style, the components interact only through the blackboard. The data-store alerts the clients whenever there is a data-store change.
- The current state of the solution is stored in the blackboard and processing is triggered by the state of the blackboard.
- The system sends notifications known as **trigger** and data to the clients when changes occur in the data.
- This approach is found in certain AI applications and complex applications, such as speech recognition, image recognition, security system, and business resource management systems etc.
- If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard and this shared data source is an active agent.
- A major difference with traditional database systems is that the invocation of computational elements in a blackboard architecture is triggered by the current state of the blackboard, and not by external inputs.

Parts of Blackboard Model

The blackboard model is usually presented with three major parts:

Knowledge Sources (KS)

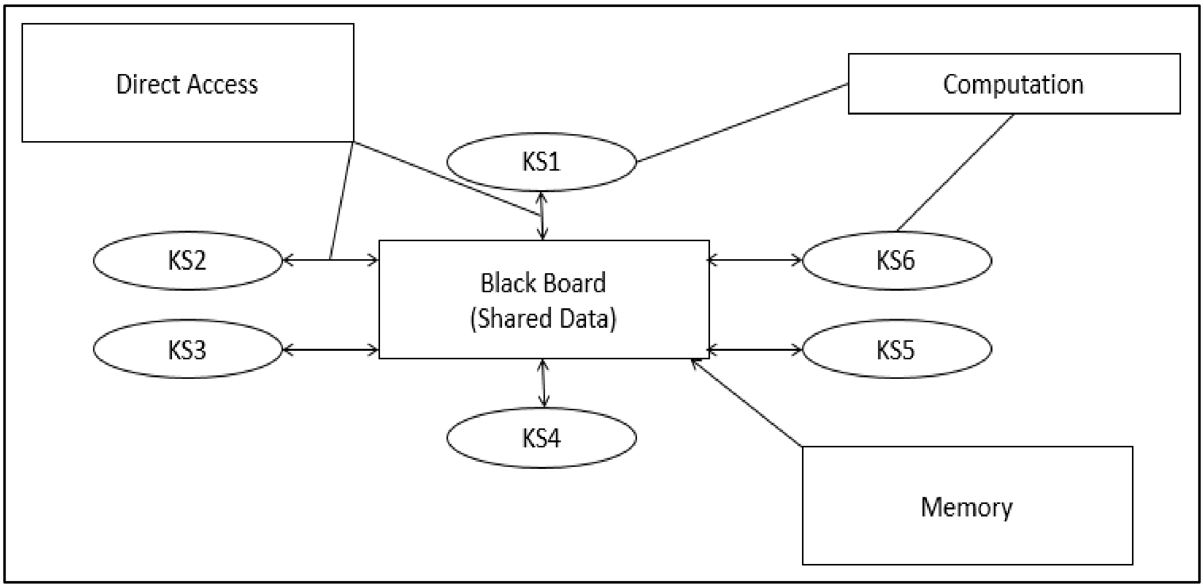
Knowledge Sources, also known as **Listeners** or **Subscribers** are distinct and independent units. They solve parts of a problem and aggregate partial results. Interaction among knowledge sources takes place uniquely through the blackboard.

Blackboard Data Structure

The problem-solving state data is organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

Control

Control manages tasks and checks the work state.



Advantages

- Provides scalability which provides easy to add or update knowledge source.
- Provides concurrency that allows all knowledge sources to work in parallel as they are independent of each other.
- Supports experimentation for hypotheses.
- Supports reusability of knowledge source agents.

Disadvantages

- The structure change of blackboard may have a significant impact on all of its agents as close dependency exists between blackboard and knowledge source.

- It can be difficult to decide when to terminate the reasoning as only approximate solution is expected.
- Problems in synchronization of multiple agents.
- Major challenges in designing and testing of system.

7. Hierarchical architecture

Hierarchical architecture views the whole system as a hierarchy structure in which the software system is decomposed into logical modules or subsystems at different levels in the hierarchy. This approach is typically used in designing system software such as network protocols and operating systems.

In system software hierarchy design, a low-level subsystem gives services to its adjacent upper level subsystems which invoke the methods in the lower level. The lower layer provides more specific functionality such as I/O services, transaction, scheduling, security services etc.; the middle layer provides more domain dependent functions such as business logic and core processing services. The upper layer provides more abstract functionality in the form of user interface such as GUIs, shell programming facilities, etc.

It is also used in organization of class libraries such as .NET class library in namespace hierarchy. All the design types can implement this hierarchical architecture and often combine with other architecture styles. Hierarchical architectural styles include the following:

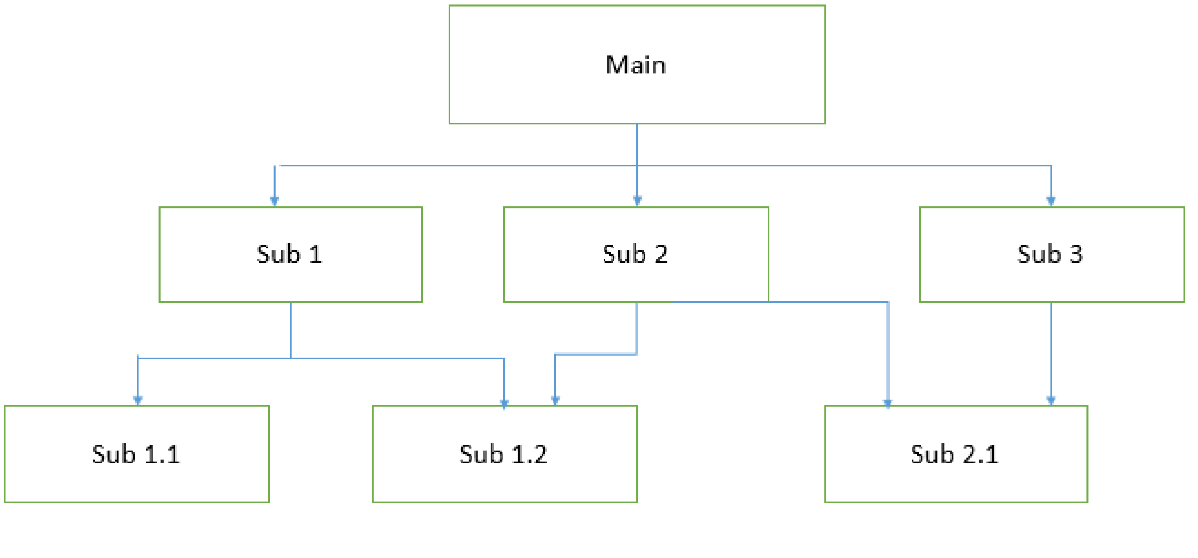
- Main-subroutine
- Master-slave
- Virtual machine

Main-subroutine

The aim of this style is to reuse the modules and freely develop individual modules or subroutine. In this style, a software system is divided into subroutines by using top-down refinement according to desired functionality of the system.

These refinements lead vertically until the decomposed modules is simple enough to have its exclusive independent responsibility. Functionality may be reused and shared by multiple callers in the upper layers. There are two ways by which data is passed as parameters to subroutines from callers:

- **Pass by Value** Subroutines only use the past data but can't modify it.
- **Pass by Reference** Subroutines use as well as changes the value of the data referenced by the parameter.



Main Subroutine

Advantages

- Easy to decompose the system based on hierarchy refinement.
- Can be used in a subsystem of object oriented design.

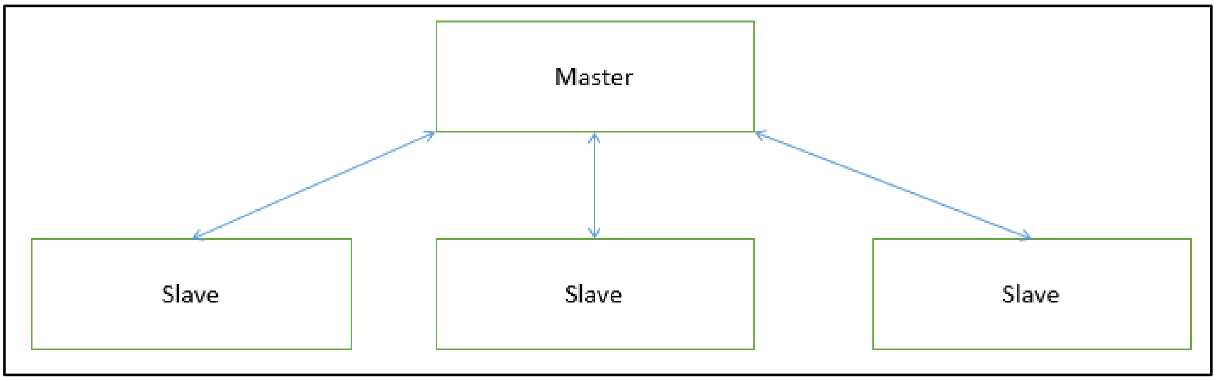
Disadvantages

- Vulnerable as it contains globally shared data.
- Tight coupling may cause more ripple effects of changes.

Master-Slave

This approach applies the 'divide and conquer' principle and supports fault computation and computational accuracy. It is a modification of the main-subroutine architecture that provides reliability of system and fault tolerance.

In this architecture, slaves provide duplicated services to the master, and the master chooses a particular result among slaves by a certain selection strategy. The slaves may perform the same functional task by different algorithms and methods or totally different functionality. It includes parallel computing in which all the slaves can be executed in parallel.



The implementation of the Master-Slave pattern follows five steps:

1. Specify how the computation of the task can be divided into a set of equal sub-tasks and identify the sub-services that are needed to process a sub-task.

2. Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.

3. Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.

4. Implement the slave components according to the specifications developed in the previous step.

5. Implement the master according to the specifications developed in step 1 to 3.

Applications

- Suitable for applications where reliability of software is critical issue.

- Widely applied in the areas of parallel and distributed computing.

Advantages

- Faster computation and easy scalability.

- Provides robustness as slaves can be duplicated.

- Slave can be implemented differently to minimize semantic errors.

Disadvantages

- Communication overhead.

- Not all problems can be divided.

- Hard to implement and portability issue.

Virtual Machine Architecture

Virtual Machine architecture pretends some functionality which is not native to the hardware and/or software on which it is implemented. A virtual machine is built upon an existing system and provides a virtual abstraction, a set of attributes, and operations.

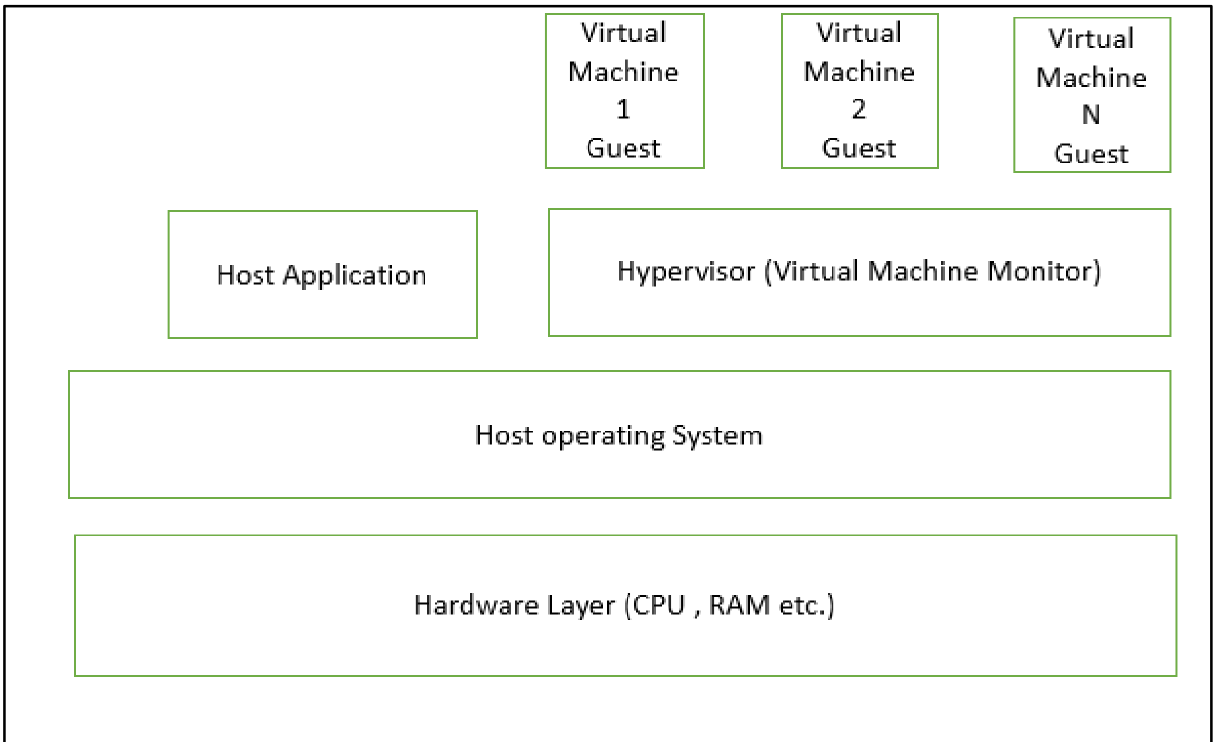
In virtual machine architecture, the master uses the 'same' subservice' from the slave and performs functions such as split work, call slaves, and combine results. It allows developers to simulate and test platforms that have not yet been built, and simulate "disaster'' modes that would be too complex, costly, or dangerous to test with the real system.

In most cases, a virtual machine splits a programming language or application environment from an execution platform. The main objective is to provide **portability**. Interpretation of a particular module via a Virtual Machine may be perceived as follows:

- The interpretation engine chooses an instruction from the module being interpreted.

- Based on the instruction, the engine updates the virtual machine's internal state and the above process is repeated.

The following figure shows the architecture of a standard VM infrastructure on a single physical machine.



Virtual Machine Architecture

The **hypervisor**, also called the **virtual machine monitor**, runs on the host OS and allocates matched resources to each guest OS. When the guest makes a system-call, the hypervisor intercepts and translates it into the corresponding system-call supported by the host OS. The hypervisor controls each virtual machine access to the CPU, memory, persistent storage, I/O devices, and the network.

Applications

Virtual machine architecture is suitable in the following domains:

- Suitable for solving a problem by simulation or translation if there is no direct solution.
- Sample applications include interpreters of microprogramming, XML processing, script command language execution, rule-based system execution, Smalltalk and Java interpreter typed programming language
- Common examples of virtual machines are interpreters, rule-based systems, syntactic shells, and command language processors.

Advantages

- Portability and machine platform independency.
- Simplicity of software development.
- Provides flexibility through the ability to interrupt and query the program.
- Simulation for disaster working model.
- Introduce modifications at runtime.

Disadvantages

- Slow execution of the interpreter due to the interpreter nature.
- There is a performance cost because of the additional computation involved in execution.

Layered Style

In this approach, the system is decomposed into a number of higher and lower layers in a hierarchy and each layer has its own sole responsibility in the system.

- Each layer consists of a group of related classes that are encapsulated in a package, in a deployed component, or as a group of subroutines in the format of method library or header file.
- Each layer provides service to the layer above it and serves as a client to the layer below i.e. request to layer i +1 invokes the services provided by the layer i via the interface of layer i. The response may go back to the layer i +1 if the task is completed; otherwise layer i continually invokes services from layer i -1 below.

Applications

Layered style is suitable in the following areas:

- Applications that involve distinct classes of services that can be organized hierarchically.
- Any application that can be decomposed into application-specific and platform-specific portions.
- Applications that have clear divisions between core services, critical services, and user interface services, etc.

Advantages

- Design based on incremental levels of abstraction.
- Provides enhancement independence as changes to the function of one layer affects at most two other layers.
- Separation of the standard interface and its implementation.
- Implemented by using component-based technology which makes the system much easier to allow for plug-and-play of new components.
- Each layer can be an abstract machine deployed independently which support portability.
- Easy to decompose the system based on the definition of the tasks in a top-down refinement manner
- Different implementations (with identical interfaces) of the same layer can be used interchangeably

Disadvantages

- Many applications or systems are not easily structured in a layered fashion.
- Lower runtime performance since a client's request or a response to client must go through potentially several layers.
- There are also performance concerns on overhead on the data marshaling and buffering by each layer.
- Opening of interlayer communication may cause deadlocks and "bridging" may cause tight coupling.
- Exceptions and error handling is an issue in the layered architecture, since faults in one layer must spread upwards to all calling layers

8. INTERACTION-ORIENTED Architecture

The primary objective of interaction-oriented architecture is to separate the interaction of user from data abstraction and business data processing. The interaction-oriented software architecture decomposes the system into three major partitions:

- **Data module** Data module provides the data abstraction and all business logic.

- **Control module** Control module identifies the flow of control and system configuration actions.

- **View presentation module** View presentation module is responsible for visual or audio presentation of data output and it also provides an interface for user input.

Interaction-oriented architecture has two major styles: **Model-View-Controller** (MVC) and **Presentation-Abstraction-Control** (PAC). Both MVC and PAC propose three components decomposition and are used for interactive applications such as web applications with multiple talks and user interactions. They are different in their flow of control and organization. PAC is an agent-based hierarchical architecture but MVC does not have a clear hierarchical structure.

Model-View-Controller (MVC)

MVC decomposes a given software application into three interconnected parts that help in separating the internal representations of information from the information presented to or accepted from the user.

Module	Function
Model	Encapsulation the underlying data and business logic
Controller	Respond to user action and direct the application flow
View	Formats and present the data from model to user.

Model

Model is a central component of MVC that directly manages the data, logic, and constraints of an application. It consists of data components which maintain the raw application data and application logic for interface.

- It is an independent user interface and captures the behavior of application problem domain.

- It is the domain-specific software simulation or implementation of the application's central structure.

- When there has been change in its state, it gives notification to its associated view to produce updated output and the controller to change the available set of commands.

View

View can be used to represent any output of information in graphical form such as diagram or chart. It consists of presentation components which provide the visual representations of data

- Views request information from their model and generate an output representation to the user.

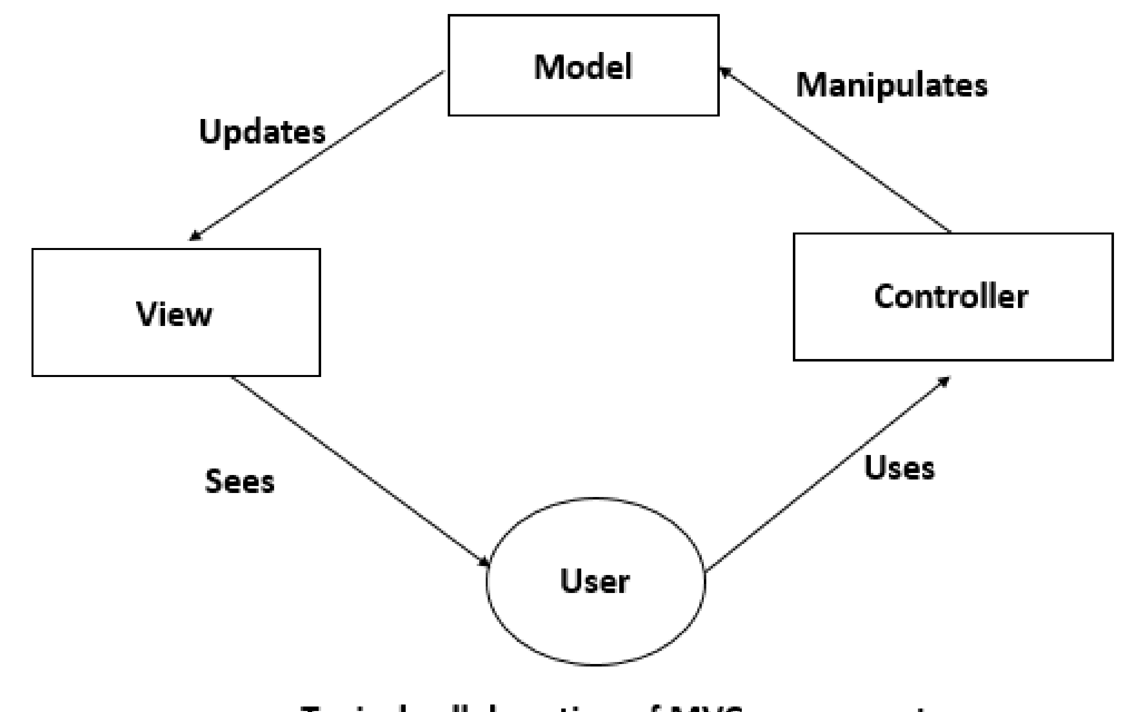
- Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

Controller

A controller accepts an input and converts it to commands for the model or view. It consists of input processing components which handle input from the user by modifying the model.

- It acts as an interface between the associated models and views and the input devices.

- It can send commands to the model to update the model's state and to its associated view to change the view's presentation of the model.



Typical collaboration of MVC component

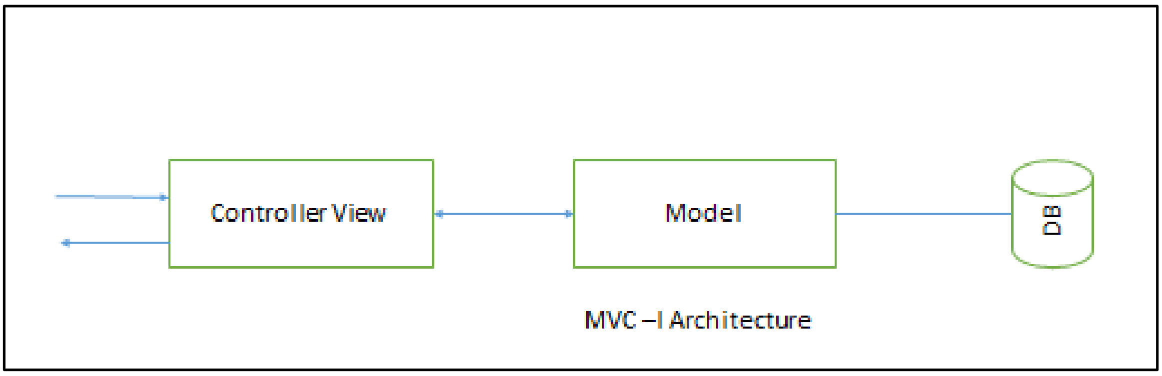
MVC - I

It is a simple version of MVC architecture where the system is divided into two sub-systems:

- **The Controller-View** The controller-view acts as input /output interface and processing is done.
- **The Model** The model provides all the data and domain services.

MVC-I Architecture

The model module notifies controller-view module of any data changes so that any graphics data display will be changed accordingly; the controller also takes appropriate action upon the changes.



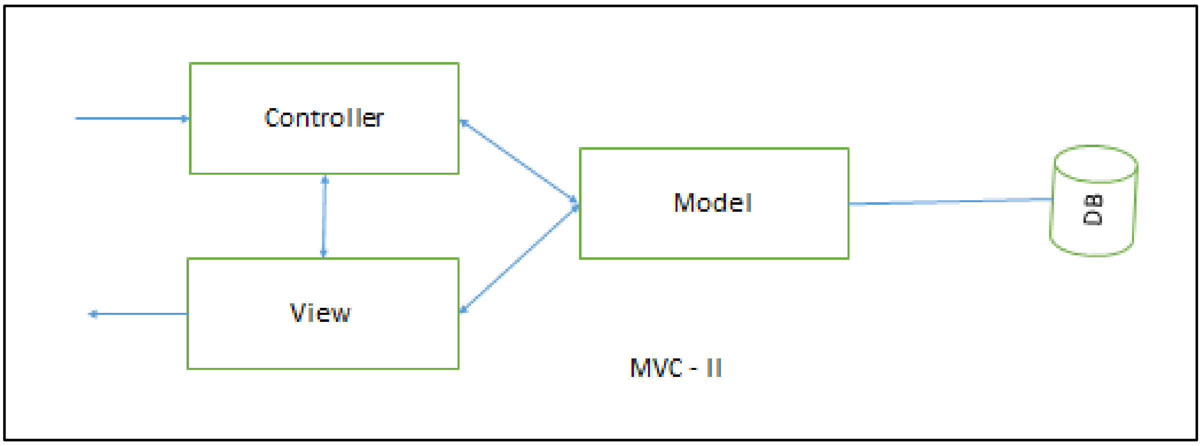
The connection between controller-view and model can be designed in a pattern of subscribe-notify whereby the controller-view subscribes to model and model notifies controller-view of any changes.

MVC - II

MVC–II is an enhancement of MVC-I architecture in which the view module and the controller module are separate. The model module plays an active role as in MVC-I by providing all the core functionality and data supported by database.

The view module presents data while controller module accepts input request, validates input data, initiates the model, the view, their connection, and also dispatches the task.

MVC-II Architecture



MVC Applications

MVC applications are effective for interactive applications where multiple views are needed for a single data model and easy to plug-in new or change interface view.

MVC applications are suitable for applications where there are clear divisions between the modules so that different professionals can be assigned to work on different aspects of such applications concurrently.

Advantages

- There are many MVC vendor framework toolkits available.
- Multiple views synchronized with same data model.
- Easy to plug-in new or replace interface views.
- Used for application development where graphics expertise professionals, programming professionals, and data base development professionals are working in a designed project team.

Disadvantages

- Not suitable for agent-oriented applications such as interactive mobile and robotics applications.
- Multiple pairs of controllers and views based on the same data model make any data model change expensive.
- The division between the View and the Controller is not clear in some cases.

Presentation-Abstraction-Control (PAC)

In PAC, the system is divided into a hierarchy of many cooperating agents (triads). It was developed from MVC to support the application requirement of multiple agents in addition to interactive requirements.

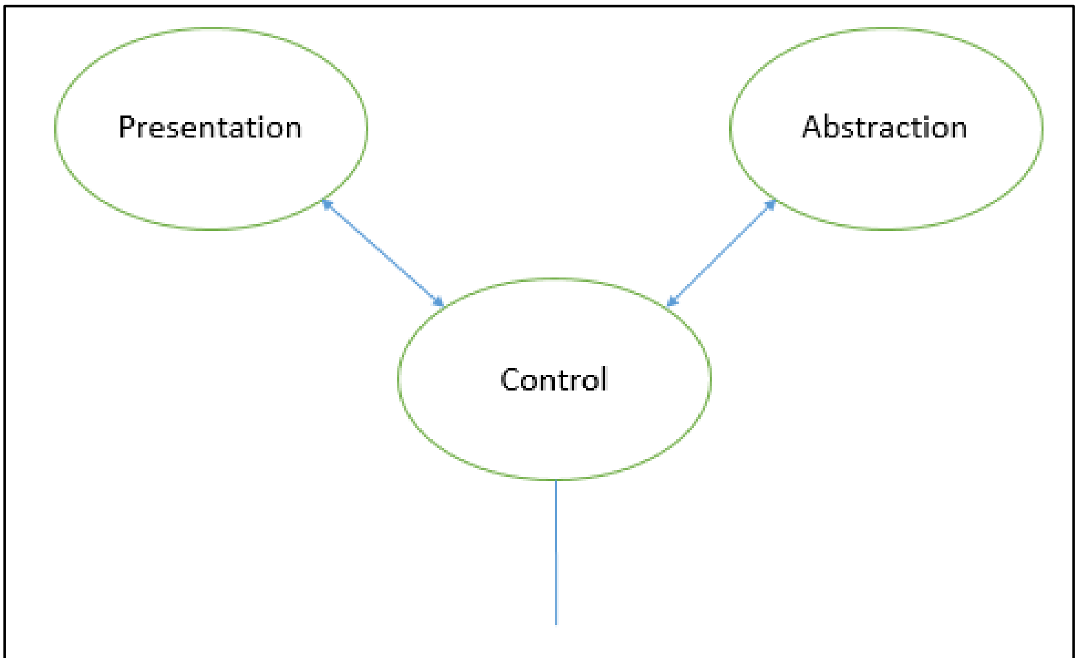
Each agent has three components:

- **The presentation component**: Formats the visual and audio presentation of data.
- **The abstraction component**: Retrieves and processes the data.
- **The control component**: Handles the task such as the flow of control and communication between the other two components.

The PAC architecture is similar to MVC, in the sense that presentation module is like view module of MVC. The abstraction module looks like model module of MVC and the control module is like the controller module of MVC but they differ in their flow of control and organization.

There are no direct connections between abstraction component and presentation component in each agent. The control component in each agent is in charge of communications with other agents.

The following figure shows a block diagram for a single agent in PAC design.



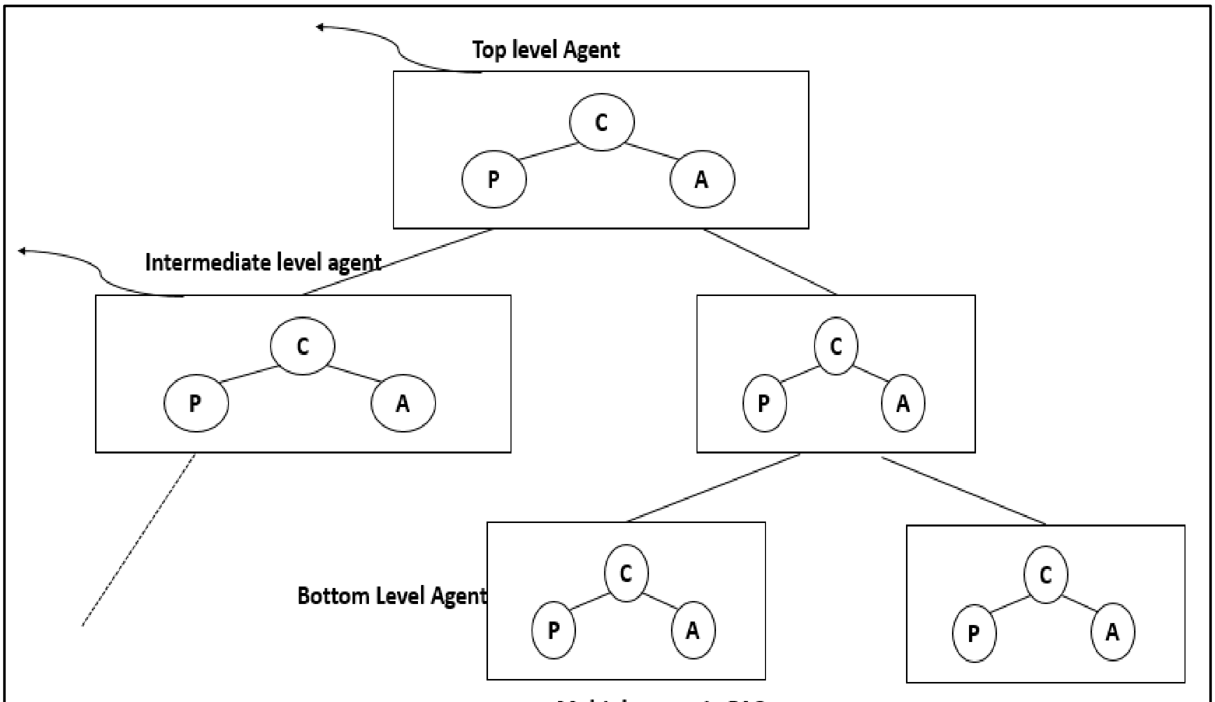
Single Agent in PAC Design

PAC with Multiple Agents

In PACs consisting of multiple agents, the top-level agent provides core data and business logics. The bottom level agents define detailed specific data and presentations. The intermediate level or middle level agent acts as coordinator of low-level agents.

- Each agent has its own specific assigned job.
- For some middle level agents the interactive presentations are not required, so they do not have a presentation component.
- The control component is required for all agents through which all the agents communicate with each other.

The following figure shows the Multiple Agents that take part in PAC.



Multiple agent in PAC

Applications

- Effective for an interactive system where the system can be decomposed into many cooperating agents in a hierarchical manner.

- Effective when the coupling among the agents is expected to be loose so that changes on an agent does not affect others.
- Effective for distributed system where all the agents are distantly distributed and each of them has its own functionalities with data and interactive interface.
- Suitable for applications with rich GUI components where each of them keeps its own current data and interactive interface and needs to communicate with other components.

Advantages

- Support for multi-tasking and multi-viewing
- Support for agent reusability and extensibility
- Easy to plug-in new agent or change an existing one
- Support for concurrency where multiple agents are running in parallel in different threads or different devices or computers

Disadvantages

- Overhead due to the control bridge between presentation and abstraction and the communication of controls among agents
- Difficult to determine the right number of the agents due to the loose coupling and high independence between agents
- Complete separation of presentation and abstraction by control in each agent generate development complexity since communications between agents only take place between the controls of agents

9. DISTRIBUTED ARCHITECTURE

In distributed architecture, components are presented on different platforms and several components can cooperate with one another over a communication network in order to achieve a specific objective or goal.

- In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.

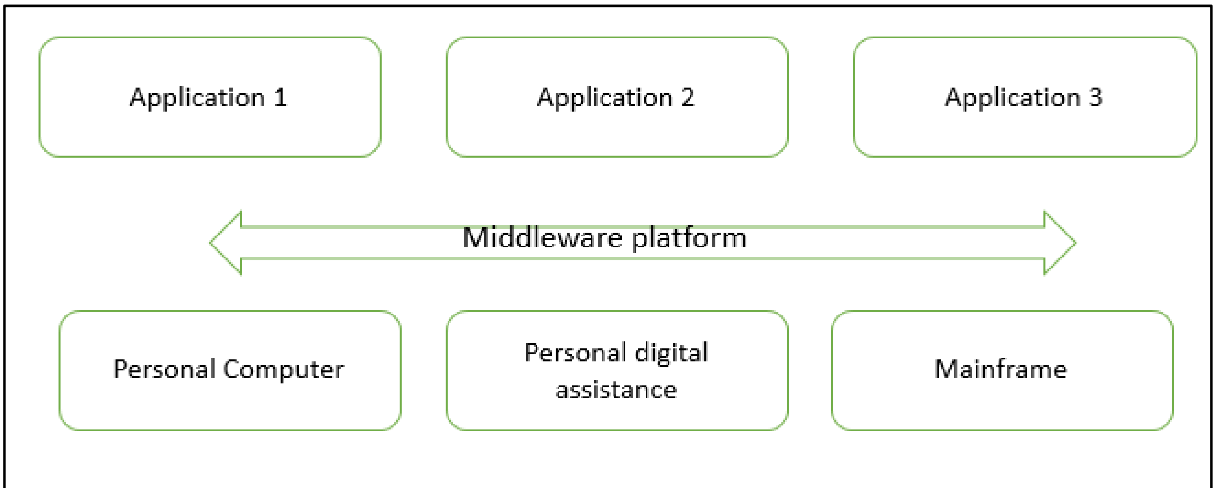
- A distributed system can be demonstrated by the client-server architecture which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA).

- There are several technology frameworks to support distributed architectures, including .NET, J2EE, CORBA, .NET Web services, AXIS Java Web services, and Globus Grid services.

- Middleware is an infrastructure that appropriately supports the development and execution of distributed applications. It provides a buffer between the applications and the network.

- It sits in the middle of system and manages or supports the different components of a distributed system. Examples are transaction processing monitors, data convertors and communication controllers etc.

Middleware as an infrastructure for distributed system



Middleware as an infrastructure for distributed system

The basis of a distributed architecture is its transparency, reliability, and availability.

The following table lists the different forms of transparency in a distributed system:

Transparency	Description
Access	Hides the way in which resources are accessed and the differences in data platform
Location	Hides where resources are located
Technology	Hides different technologies such as programming language and OS from user
Migration / Relocation	Hide resources that may be moved to another location which are in use
Replication	Hide resources that may be copied at several location
Concurrency	Hide resources that may be shared with other users
Failure	Hides failure and recovery of resources from user
Persistence	Hides whether a resource (software) is in memory or disk

Advantages

- **Resource sharing**: Sharing of hardware and software resources.
- **Openness**: Use of equipment and software different vendors
- **Concurrency**: Concurrent processing to enhance performance.
- **Scalability**: Increased throughput by adding new resources
- **Fault tolerance**: The ability to continue in operation after a fault has occurred.

Disadvantages

- **Complexity**: They are more complex than centralized systems.
- **Security**: More susceptible to external attack.
- **Manageability**: More effort required for system management.
- **Unpredictability**: Unpredictable responses depending on the system organization and network load.

Centralized System vs. Distributed System

Criteria	Centralized system	Distributed System
Economics	Low	High
Availability	Low	High
Complexity	Low	High
Consistency	Simple	High
Scalability	Poor	Good
Technology	Homogeneous	Heterogeneous
Security	High	Low

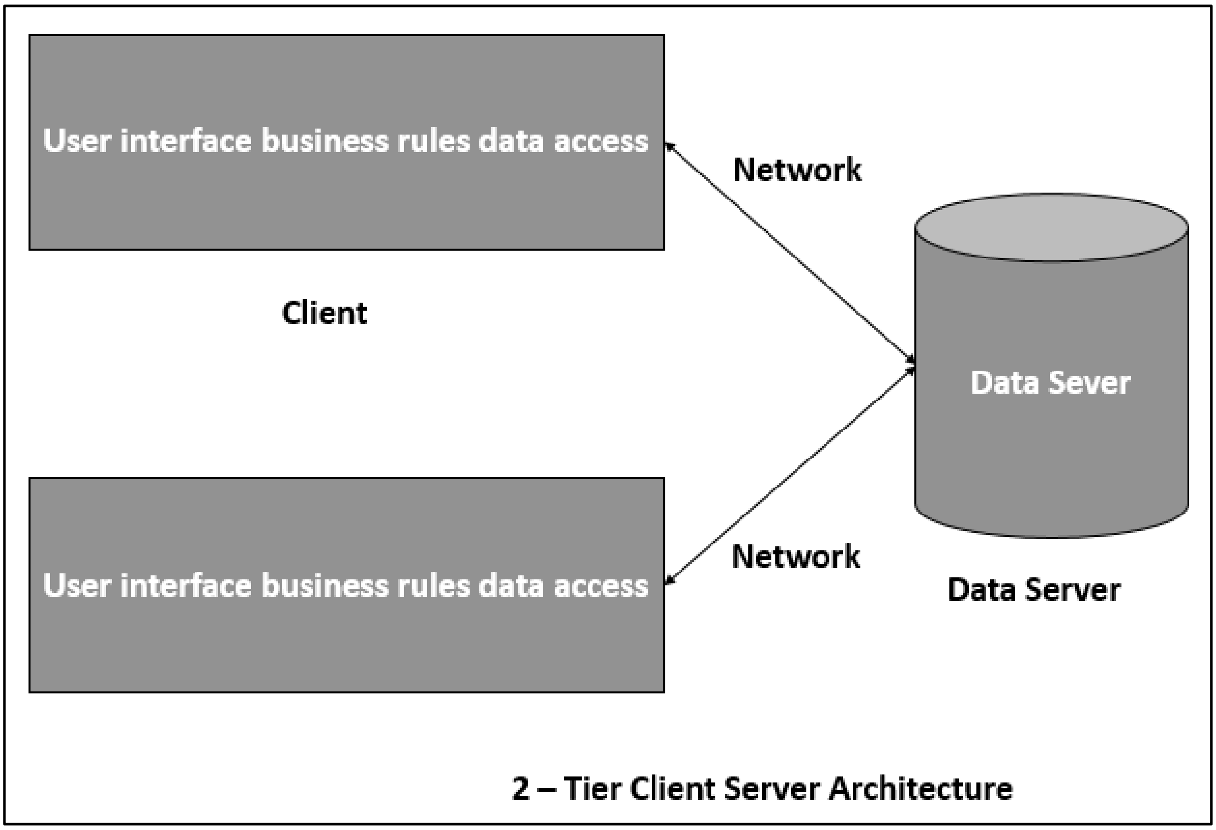
Client-Server Architecture

The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes:

- **Client**: This is the first process that issues a request to the second process i.e., the server.

- **Server**: This is the second process that receives the request, carries it out, and sends a reply to the client.

In this architecture, the application is modelled as a set of services that are provided by servers and a set of clients that use these services. The servers need not know about clients, but the clients must know the identity of servers, and the mapping of processors to processes is not necessarily 1 : 1



Client-server Architecture can be classified into two models based on the functionality of the client:

Thin-client model

In thin-client model, all the application processing and data management is carried by the server. The client is simply responsible for running the presentation software.

- Used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client

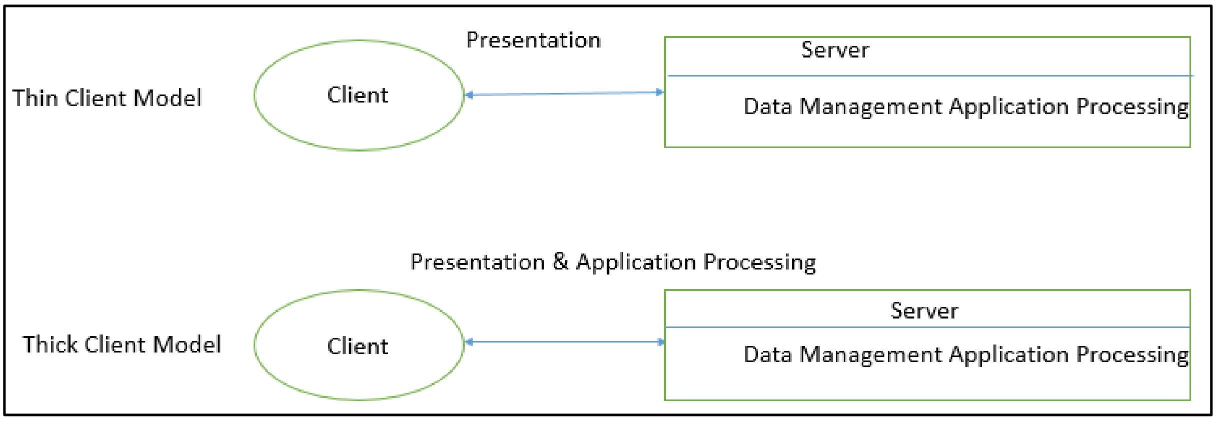
- A major disadvantage is that it places a heavy processing load on both the server and the network.

Thick/Fat-client model

In thick-client model, the server is only in charge for data management. The software on the client implements the application logic and the interactions with the system user.

- Most appropriate for new C/S systems where the capabilities of the client system are known in advance

- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.



Advantages

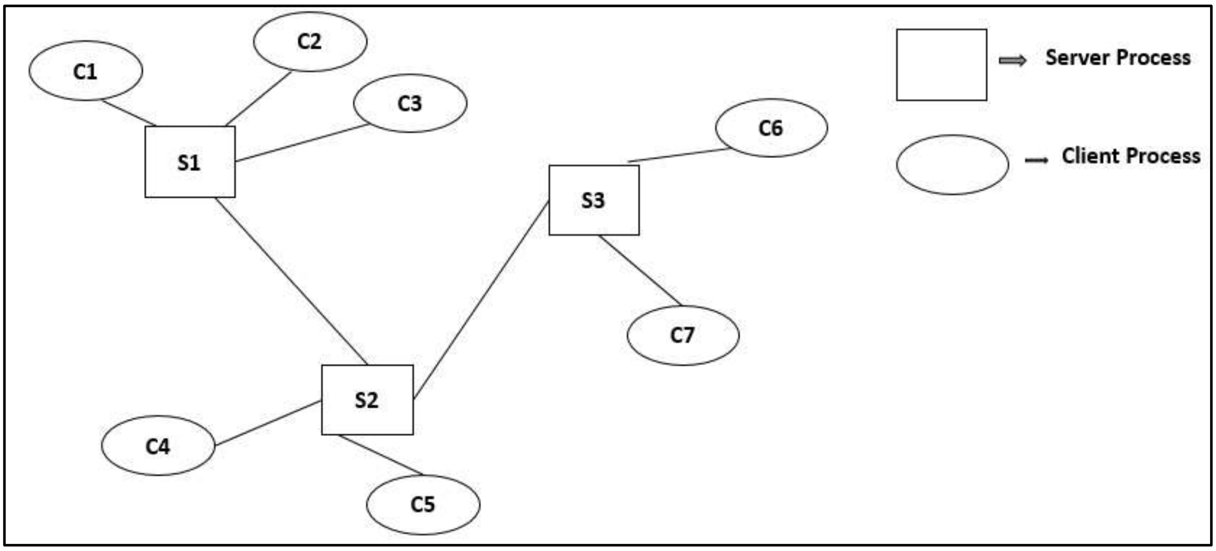
- Separation of responsibilities such as user interface presentation and business logic processing.
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications.
- It makes it easy to migrate or integrate existing applications into a distributed environment.
- It also makes effective use of resources when a large number of clients are accessing a high-performance server.

Disadvantages

- Lack of heterogeneous infrastructure to deal with the requirement changes.
- Security complications.
- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.

Multi-Tier Architecture (n-tier Architecture)

Multi-tier architecture is a client–server architecture in which the functions such as presentation, application processing, and data management are physically separated. By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application. It provides a model by which developers can create flexible and reusable applications.



The most general use of multi-tier architecture is the three-tier architecture. A three-tier architecture is typically composed of a presentation tier, an application tier, and a data storage tier and may execute on a separate processor.

Presentation Tier

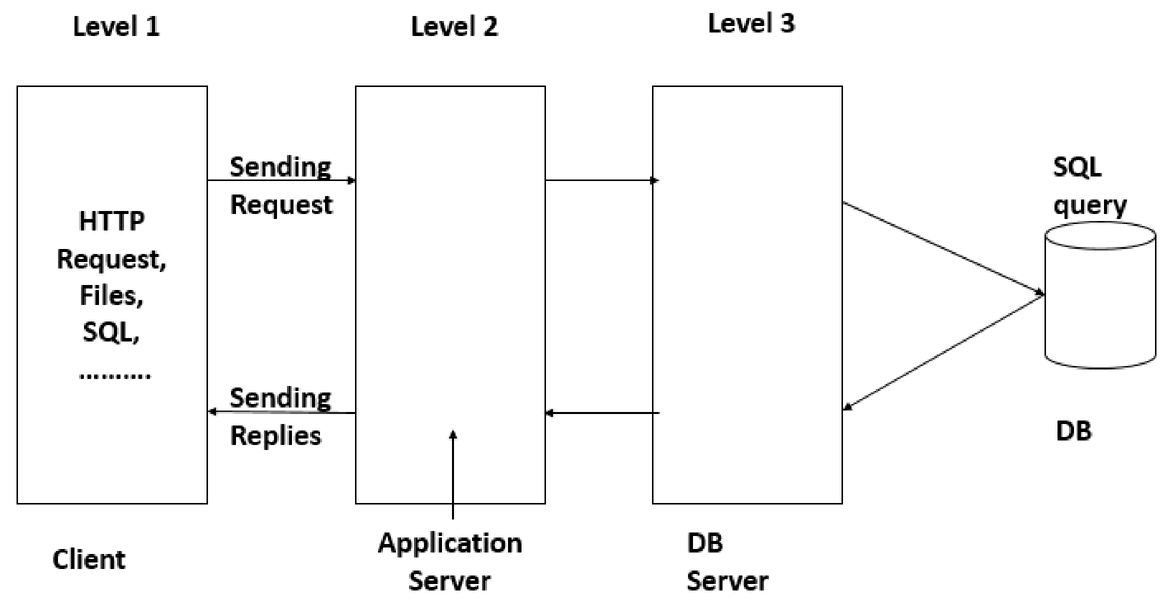
Presentation layer is the topmost level of the application by which users can access directly such as webpage or Operating System GUI (Graphical User interface). The primary function of this layer is to translate the tasks and results to something that user can understand. It communicates with other tiers so that it places the results to the browser/client tier and all other tiers in the network.

Application Tier (Business Logic, Logic Tier, or Middle Tier)

Application tier coordinates the application, processes the commands, makes logical decisions, evaluation, and performs calculations. It controls an application's functionality by performing detailed processing. It also moves and processes data between the two surrounding layers.

Data Tier

In this layer, information is stored and retrieved from the database or file system. Then this information is then passed back for processing and then back to the user. It includes the data persistence mechanisms (database servers, file shares, etc.) and provides API (application programming Interface) to the application tier which provides methods of managing the stored data.



3 Tier Architecture

Advantages

- Better performance than a thin-client approach and is simpler to manage than a thick-client approach.
- Enhances the reusability and scalability - as demands increase, extra servers can be added.
- Provides multi-threading support and also reduces network traffic.
- Provides maintainability and flexibility

Disadvantages

- Unsatisfactory Testability due to lack of testing tools.
- More critical server reliability and availability.

Broker Architectural Style

Broker Architectural Style is a middleware architecture used in distributed computing to coordinate and enable the communication between registered servers and clients. Here, object communication takes place through a middleware system called an object request broker (software bus).

- Client and the server do not interact with each other directly. Client and server have a direct connection to its proxy which communicates with the mediator-broker.
- A server provides services by registering and publishing their interfaces with the broker and clients can request the services from the broker statically or dynamically by look-up.

- CORBA (Common Object Request Broker Architecture) is a good implementation example of the broker architecture.

Components of Broker Architectural Style

The components of broker architectural style are discussed below.

Broker

Broker is responsible for coordinating communication, such as forwarding and dispatching the results and exceptions. It can be either an invocation-oriented service, a document or message - oriented broker to which clients send a message.

- It is responsible for brokering the service requests, locating a proper server, transmitting requests, and sending responses back to clients.

- It retains the servers' registration information including their functionality and services as well as location information.

- It provides APIs for clients to request, servers to respond, registering or unregistering server components, transferring messages, and locating servers.

Stub

Stubs are generated at the static compilation time and then deployed to the client side which is used as a proxy for the client. Client-side proxy acts as a mediator between the client and the broker and provides additional transparency between them and the client; a remote object appears like a local one.

The proxy hides the IPC (inter-process communication) at protocol level and performs marshaling of parameter values and un-marshaling of results from the server.

Skeleton

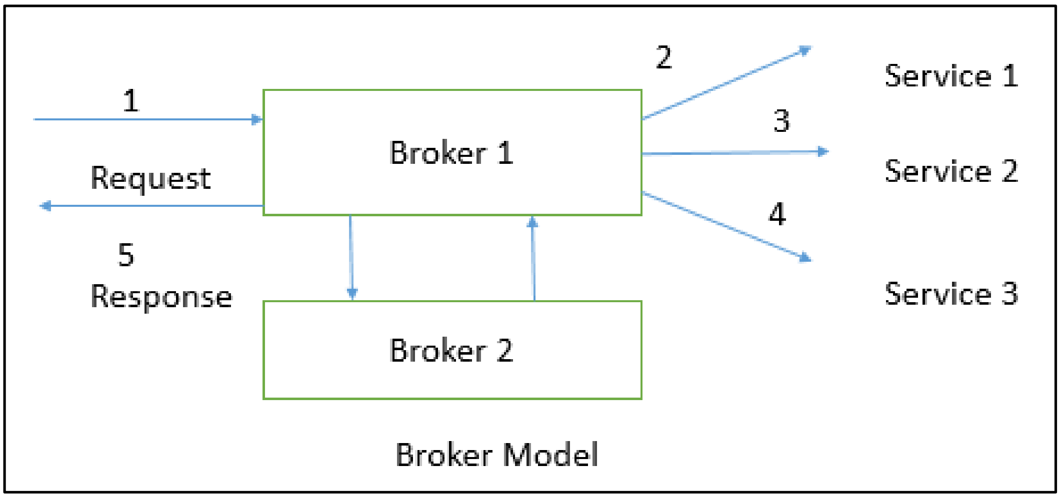
Skeleton is generated by the service interface compilation and then deployed to the server side which is used as a proxy for the server. Server-side proxy encapsulates low-level system-specific networking functions and provides high-level APIs to mediate between the server and the broker.

It receives the requests, unpacks the requests, unmarshals the method arguments, calls the suitable service, and also marshals the result before sending it back to the client.

Bridge

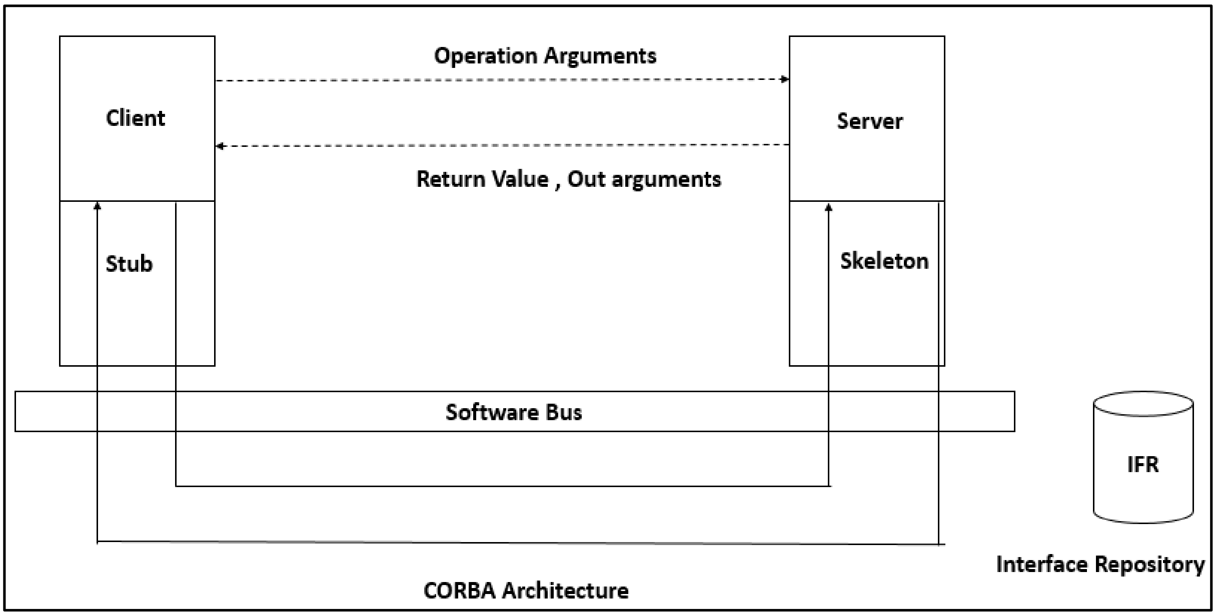
A bridge can connect two different networks based on different communication protocols. It mediates different brokers such as DCOM, .NET remote and Java CORBA brokers.

Bridges are optional component which hides the implementation details when two brokers interoperate and take requests and parameters in one format and translate them to another format.



Broker implementation in CORBA

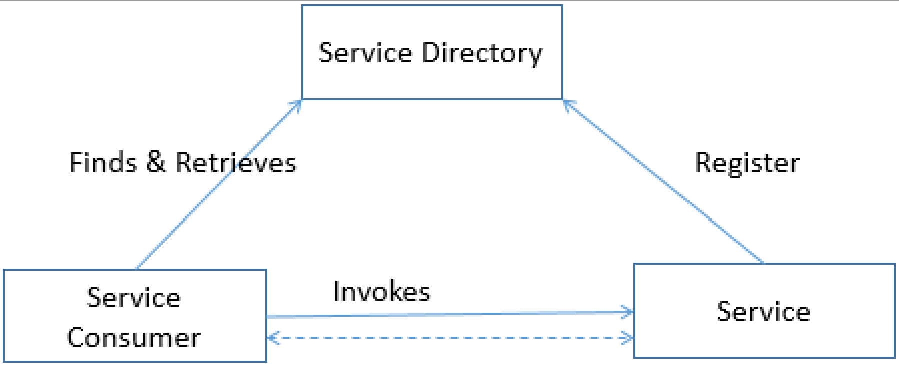
CORBA is an international standard for an Object Request Broker – a middleware to manage communications between distributed objects defined by OMG (object management group)



Service-Oriented Architecture (SOA)

A service is a component of business functionality that is well-defined, self-contained, independent, published, and available to be used via a standard programming interface. The connections between services are conducted by common and universal message-oriented protocols such as the SOAP Web service protocol, which can deliver requests and responses between services loosely.

Service-oriented architecture is a client/server design which support business-driven IT approach in which an application consists of software services and software service consumers (also known as clients or service requesters).



SOA

Features of SOA

A service-oriented architecture provides the following features:

- **Distributed Deployment** Expose enterprise data and business logic as loosely, coupled, discoverable, structured, standard-based, coarse-grained, stateless units of functionality called services.

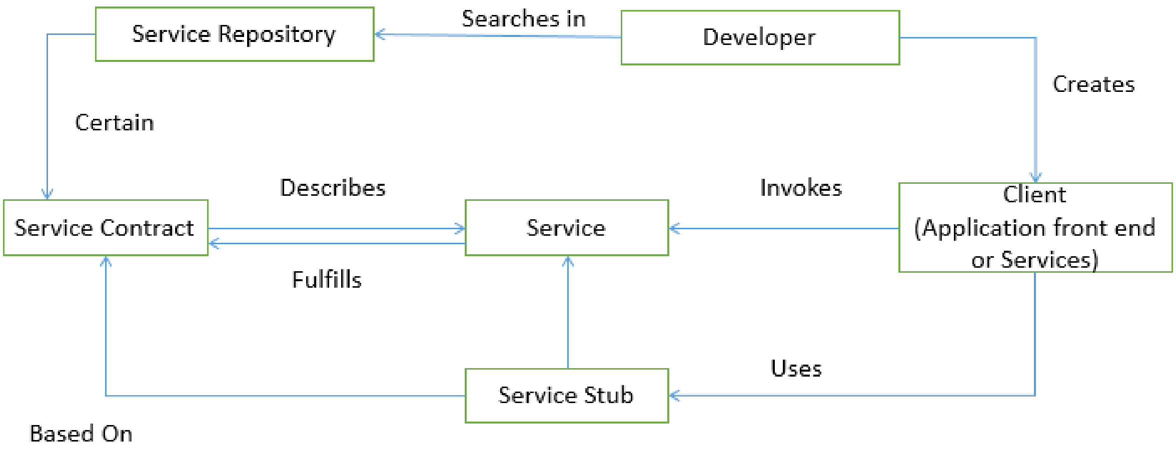
- **Composability** Assemble new processes from existing services that are exposed at a desired granularity through well defined, published, and standard complaint interfaces.

- **Interoperability** Share capabilities and reuse shared services across a network irrespective of underlying protocols or implementation technology.

- **Reusability** Choose a service provider and access to existing resources exposed as services.

SOA Operation

The following figure illustrates how SOA operates:



SOA Operations

Advantages

- Loose coupling of service–orientation provides great flexibility for enterprises to make use of all available service recourses irrespective of platform and technology restrictions.
- Each service component is independent from other services due to the stateless service feature.
- The implementation of a service will not affect the application of the service as long as the exposed interface is not changed.
- A client or any service can access other services regardless of their platform, technology, vendors, or language implementations.
- Reusability of assets and services since clients of a service only need to know its public interfaces, service composition.
- SOA based business application development are much more efficient in terms of time and cost.
- Enhances the scalability and provide standard connection between systems.
- Efficient and effective usage of 'Business Services'.
- Integration becomes much easier and improved intrinsic interoperability.
- Abstract complexity for developers and energize business processes closer to end users

10. Component-Based architecture

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure **component reusability**. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit. There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services. These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.

Component-oriented software design has many advantages over the traditional object-oriented approaches such as:

- Reduced time in market and the development cost by reusing existing components.

- Increased reliability with the reuse of the existing components.

What is a Component?

A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.

A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

Views of a Component

A component can have three different views: object-oriented view, conventional view, and process-related view.

Object-oriented view

A component is viewed as a set of one or more cooperating classes. Each problem domain class (analysis) and infrastructure class (design) is explained to identify all attributes and operations that apply to its implementation. It also involves defining the interfaces that enable classes to communicate and cooperate.

Conventional view

It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.

Process-related view

In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.

- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.

- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach

- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

Characteristics of Components

- **Reusability** Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.

- **Replaceable** Components may be freely substituted with other similar components.

- **Not context specific** Components are designed to operate in different environments and contexts.

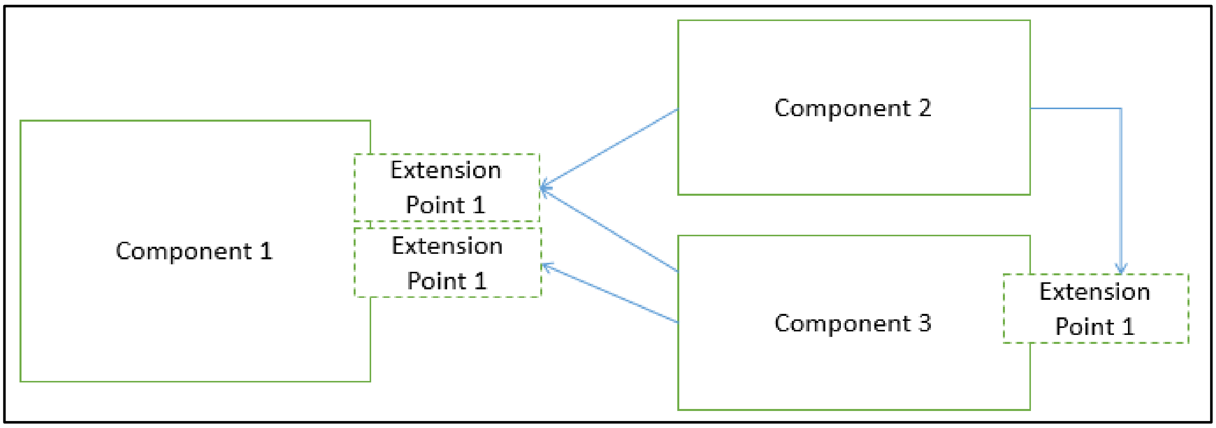
- **Extensible** A component can be extended from existing components to provide new behavior.
- **Encapsulated** A component depicts the interfaces that allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.

- **Independent** Components are designed to have minimal dependencies on other components.

Principles of Component−Based Design

A component-level design can be represented using some intermediary representation (e.g. graphical, tabular, or text-based) that can be translated into source code. The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.
- Depend on abstractions component do not depend on other concrete components which increase difficulty in expendability.
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components
- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.
- For a server class, specialized interfaces should be created to serve major categories of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.
- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plugin to offer to another plugin API.



Component-Level Design Guidelines

Create a naming conventions for components that are specified as part of the architectural model and then refine or elaborate as part of the component-level model

- Attain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model.
- Extract the business process entities that can exist independently without any associated dependency on other entities.
- Recognize and discover these independent entities as new components.

- Use infrastructure component names that reflect their implementation-specific meaning.
- Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

Conducting Component-Level Design

Recognize all design classes that correspond to the problem domain as defined in the analysis model and architectural model

- Recognize all design classes that correspond to the infrastructure domain
- Describe all design classes that are not acquired as reusable components, specify message details
- Identify appropriate interfaces for each component and elaborate attributes and define data types and data structures required to implement them.
- Describe processing flow within each operation in detail by means of pseudo code or UML activity diagrams
- Describe persistent data sources (databases and files) and identify the classes required to manage them
- Develop and elaborate behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
- Elaborate deployment diagrams to provide additional implementation detail.
- Demonstrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environment.
- The final decision can be made by using established design principles and guidelines. Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model.

Advantages

- **Ease of deployment** As new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost** The use of third-party components allows you to spread the cost of development and maintenance.

- **Ease of development** Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- **Reusable** The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- **Modification of technical complexity** A component modifies the complexity through the use of a component container and its services.
- **Reliability** The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse
- **System maintenance and evolution** Easy to change and update the implementation without affecting the rest of the system.
- **Independent** Independency and flexible connectivity of components. Independent development of components by different group in parallel. Productivity for the software development and future software development.

11. User interface

User interface is the first impression of a software system from the user's point of view. Therefore any software system must satisfy the requirement of user. UI mainly performs two functions:

- Accepting the user's input
- Displaying the output

User interface plays a crucial role in any software system. It is possibly the only visible aspect of a software system.

- Users will initially see the architecture of software system's external user interface without considering its internal architecture.

- A good user interface must attract the user to use the software system without mistakes. It should help the user to understand the software system easily without misleading information. A bad UI may cause market failure against the competition of software system.

- UI has its syntax and semantics. The syntax comprises component types such as textual, icon, button etc. and usability summarizes the semantics of UI. The quality of UI is characterized by its look and feel (syntax) and its usability (semantics).

- There are basically two major kinds of user interface: a) Textual b) Graphical.

- Software in different domains may require different style of its user interface for e.g. calculator need only a small area for displaying numeric numbers but a big area for commands, A web page needs forms, links, tabs etc.

Graphical User Interface

A graphical user interface is the most common type of user interface seen today. It is a very user friendly because it makes use of pictures, graphics, and icons - hence why it is called 'graphical'.

It is also known as a **WIMP interface** because it makes use of:

- **Windows** A rectangular area on the screen where the commonly used applications run

- **Icons** A picture or symbol which is used to represent a software application or hardware device

- **Menus** A list of options from which the user can choose what they require

- **Pointers** A symbol such as an arrow which moves around the screen as user move the mouse. It helps user to select objects.

Design of User Interface

It starts with task analysis which understands the user's primary tasks and problem domain. It should be designed in terms of User's terminology and outset of user's job rather than programmer's.

- To perform user interface analysis, the practitioner needs to study and understand four elements:

    - The **users** who will interact with the system through the interface

    - The **tasks** that end users must perform to do their work

    - The **content** that is presented as part of the interface

    - The **work environment** in which these tasks will be conducted

- Proper or good UI design works from the user's capabilities and limitations not the machines. While designing the UI, knowledge of the nature of the user's work and environment is also critical.

- The task to be performed can then be divided which are assigned to the user or machine, based on knowledge of the capabilities and limitations of each. The design of a user interface is often divided into four different levels:
    - **The conceptual level** It describes the basic entities considering the user's view of the system and the actions possible upon them.

    - **The semantic level** It describes the functions performed by the system i.e. description of the functional requirements of the system, but does not address how the user will invoke the functions.

    - **The syntactic level** It describes the sequences of inputs and outputs required to invoke the functions described.

    - **The lexical level** It determines how the inputs and outputs are actually formed from primitive hardware operations.

- User interface design is an iterative process, where all the iteration explains and refines the information developed in the preceding step. General steps for user interface design

    - Define user interface objects and actions (operations)

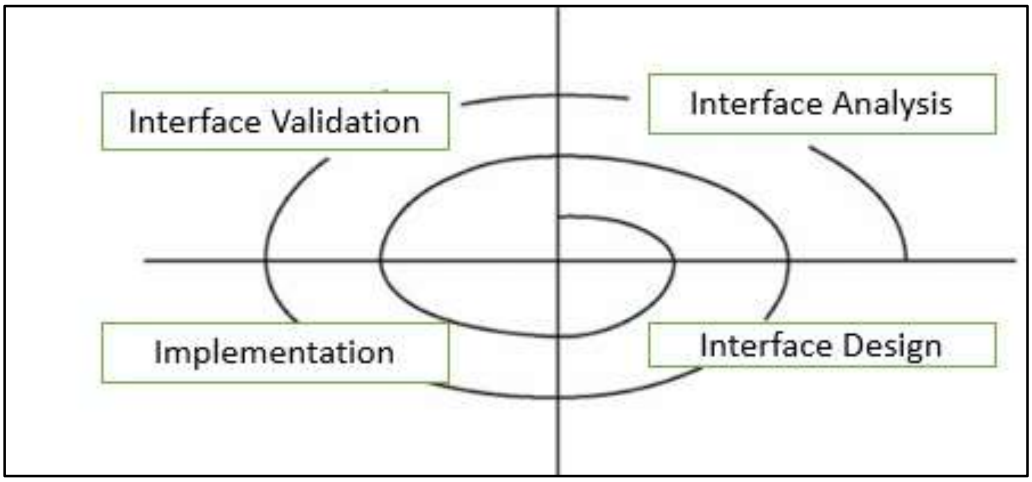
    - Define events (user actions) that will cause the state of the user interface to change

    - Indicate how the user interprets the state of the system from information provided through the interface

    - Describe each interface state as it will actually look to the end user

User Interface Development Process

It follows a spiral process as shown in the following diagram:



Interface analysis

It concentrates or focuses on users, tasks, content and work environment who will interact with the system. Defines the human - and computer-oriented tasks that are required to achieve system function.

Interface design

It defines a set of interface objects, actions and their screen representations that enable a user to perform all defined tasks in a manner that meets every usability objective defined for the system.

Interface construction

It starts with a prototype that enables usage scenarios to be evaluated and continues with development tools to complete the construction.

Interface validation

It focuses on the ability of the interface to implement every user task correctly, accommodate all task variations, to achieve all general user requirements and the degree to which the interface is easy to use and easy to learn.

User Interface Models

When a user interface is analyzed and designed these four models are used:

User profile model

- Created by a user or software engineer which establishes the profile of the end-users of the system based on age, gender, physical abilities, education, motivation, goals, and personality.

- Considers syntactic and semantic knowledge of the user and classifies users as novices, knowledgeable intermittent users and knowledgeable frequent users.

Design model

- Created by a software engineer which incorporates data, architectural, interface, and procedural representations of the software.

- Derived from the analysis model of the requirements and controlled by the information in the requirements specification which helps in defining the user of the system.

Implementation model

- Created by the software implementers who work on look and feel of the interface combined with all supporting information (books, videos, help files) that describes system syntax and semantics.

- Serves as a translation of the design model and attempts to agree with the user's mental model so that users then feel comfortable with the software and use it effectively

User's mental model

- Created by the user when interacting with the application which contains the image of the system that users carry in their heads.

- Often called the user's system perception and correctness of the description depends upon the user's profile and overall familiarity with the software in the application domain.

Design Considerations of User Interface

User centered

A user interface must be a user-centered product which involves users throughout a product's development lifecycle. The prototype of a user interface should be available to users and feedback from users should be incorporated into the final product.

Simple and Intuitive

UI provides simplicity and intuitiveness so that it can be used quickly and effectively without instructions. GUI are better than textual UI as GUI consists of menus, windows, and buttons and is operated by simply using mouse.

Place Users in Control

Do not force users to complete predefined sequences. Give them options—to cancel or to save and return to where they left off. Use terms throughout the interface that users can understand, rather than system or developer terms.

Provide users with some indication that an action has been performed, either by showing them the results of the action, or acknowledging that the action has taken place successfully.

Transparency

UI must be transparent that helps users to feel like they are reaching right through computer and directly manipulating the objects they are working with. The interface can be made transparent by giving users work objects rather than system objects. For example, users should understand that their system password must be at least 6 characters, not how many bytes of storage a password must be.

Use progressive disclosure

Always provide easy access to common features and frequently used actions. Hide less common features and actions and allow users to navigate them. Do not try to put every piece of information in one main window. Use secondary windows for information that is not key information.

Consistency

UI maintains the consistency within and across product, keep interaction results the same, UI commands and menus should have the same format, command punctuations should be similar and parameters should be passed to all commands in the same way. UI should not have behavior's that can surprise the users and should include the mechanisms that allows users to recover from their mistakes.

Integration

The software system should integrate smoothly with other applications such as MS notepad and MS-Office. It can use Clipboard commands directly to perform data interchange.

Component Oriented

UI design must be modular and incorporate component oriented architecture so that the design of UI will have the same requirements as the design of the main body of the software system. The modules can easily be modified and replaced without affecting of other parts of the system.

Customizable

The architecture of whole software system incorporate plug-in modules which allows many different people to independently extend the software. It allows individual users to select from various available forms in order to suit personal preferences and needs.

Reduce Users' Memory Load

Do not force users to have to remember and repeat what the computer should be doing for them. For example, when filling in online forms, customer names, addresses, and telephone numbers should be remembered by the system once a user has entered them, or once a customer record has been opened.

User interfaces support long-term memory retrieval by providing users with items for them to recognize rather than having to recall information.

Separation

UI must be separated from the logic of the system through its implementation for increasing reusability and maintainability.

12. Architecture Techniques

Iterative and Incremental Approach

It is an iterative and incremental approach consisting of five main steps that helps to generate candidate solutions. This candidate solution can further be refined by repeating these steps and finally create an architecture design that best fits our application. At the end of the process, we can review and communicate our architecture to all interested parties.

It is just one possible approach. There are many other more formal approaches to defining, reviewing, and communicating your architecture.

Identify Architecture Goal

Identify the architecture goal that forms the architecture and design process. Flawless and defined objectives emphasize on the architecture, solve the right problems in the design and helps to determine when the current phase has completed, and ready to move to the next phase.

This step includes the following activities:

- Identify your architecture goals at the start.
- Identify the consumer of our architecture.
- Identify the constraints.

Examples of architecture activities include building a prototype to get feedback on the order-processing UI for a Web application, building a customer order-tracking application, and designing the authentication and authorization architecture for an application in order to perform a security review.

Key Scenarios

This step puts emphasis on the design that matters the most. A scenario is an extensive and covering description of a user's interaction with the system.

Key scenarios are those that are considered the most important scenarios for the success of your application. It helps to make decisions about the architecture. The goal is to achieve a balance between the user, business, and system objectives. For example, user authentication is a key scenario because they are an intersection of a quality attribute (security) with important functionality (how a user logs into your system).

Application Overview

Build an overview of application which makes the architecture more touchable, connecting it to real-world constraints and decisions. It consists of the following activities:

Identify Application Type

Identify application type whether it is a mobile application, a rich client, a rich internet application, a service, a web application, or some combination of these types.

Identify Deployment Constraints

Choose an appropriate deployment topology and resolve conflicts between the application and the target infrastructure.

Identify Important Architecture Design Styles

Identify important architecture design styles such as client/server, layered, message-bus, and domain-driven design etc. to improve partitioning and promotes design reuse by providing solutions to frequently recurring problems. Applications will often use a combination of styles.

Identify the Relevant Technologies

Identify the relevant technologies by considering the type of application we are developing, our preferred options for application deployment topology and architectural styles. The choice of technologies will also be directed by organization policies, infrastructure limitations, resource skills, and so on.

Key Issues or Key Hotspots

While designing an application, hot spots are the zones where mistakes are most often made. Identify key issues based on quality attributes and crosscutting concerns. Potential issues include the appearance of new technologies, and critical business requirements.

Quality attributes are the overall features of your architecture that affect run-time behavior, system design, and user experience. Crosscutting concerns are the features of our design that may apply across all layers, components, and tiers.

These are also the areas in which high-impact design mistakes are most often made. Examples of crosscutting concerns are authentication and authorization, communication, configuration management, exception management and validation etc.

Candidate Solutions

After defining the key hotspots, build the initial baseline architecture or first high level design and then start to fill in the details to generate candidate architecture.

Candidate architecture includes the application type, the deployment architecture, the architectural style, technology choices, quality attributes, and crosscutting concerns. If the candidate architecture is an improvement, it can become the baseline from which new candidate architectures can be created and tested.

Validate the candidate solution design against the key scenarios and requirements that have already defined, before iteratively following the cycle and improving the design.

We may use architectural spikes to discover the specific areas of the design or to validate new concepts. Architectural spikes are a design prototype which determine the feasibility of a specific design path, reduce the risk, and quickly determine the viability of different approaches. Test architectural spikes against key scenarios and hotspots.

Architecture Review

Architecture review is the most important task in order to reduce the cost of mistakes and to find and fix architectural problems as early as possible. It is a well-established, cost-effective way of reducing project costs and the chances of project failure.

- Review the architecture frequently at major project milestones, and in response to other significant architectural changes.

- The main objective of an architecture review is to determine the feasibility of baseline and candidate architectures, verify the architecture correctly.

- Links the functional requirements and the quality attributes with the proposed technical solution. It also helps to identify issues and recognize areas for improvement

Scenario-based evaluations are a dominant method for reviewing an architecture design which focuses on the scenarios that are most important from the business perspective, and which have the greatest impact on the architecture. Following are common review methodologies:

Software Architecture Analysis Method (SAAM)

It is originally designed for assessing modifiability, but later was extended for reviewing architecture with respect to quality attributes.

Architecture Tradeoff Analysis Method (ATAM)

It is a polished and improved version of SAAM which review architectural decisions with respect to the quality attributes requirements, and how well they satisfy particular quality goals.

Active Design Review (ADR)

It is best suited for incomplete or in-progress architectures which more focus on a set of issues or individual sections of the architecture at a time, rather than performing a general review.

Active Reviews of Intermediate Designs (ARID)

It combines the ADR aspect of reviewing in-progress architecture with a focus on a set of issues, and the ATAM and SAAM approach of scenario-based review focused on quality attributes.

Cost Benefit Analysis Method (CBAM)

It focuses on analyzing the costs, benefits, and schedule implications of architectural decisions.

Architecture Level Modifiability Analysis (ALMA)

It estimates the modifiability of architecture for business information systems (BIS).

Family Architecture Assessment Method (FAAM)

It estimates information system family architectures for interoperability and extensibility.

Communicating the Architecture Design

After completing the architecture design, we must communicate the design to the other stakeholders which include development team, system administrators, operators, business owners, and other interested parties.

There are several well-known methods for describing architecture to others, including the following:

4 + 1 Model

This approach uses five views of the complete architecture. Four of the views describe the architecture from different approaches: the **logical view,** the **process view,** the **physical view,** and the **development view**. A fifth view shows the scenarios and use cases for the software. It allows stakeholders to see the features of the architecture that specifically interest them.

Architecture Description Language (ADL)

This approach is used to describe software architecture prior to system implementation. It addresses the following concerns: behavior, protocol, and connector.

The main advantage of ADL is that we can analyze the architecture for completeness, consistency, ambiguity, and performance before formally beginning use of the design.

Agile Modeling

This approach follows the concept that "content is more important than representation." It ensures that the models created are simple and easy to understand, sufficiently accurate, detailed, and consistent.

Agile model documents target specific customer(s) and fulfill the work efforts of that customer. The simplicity of the document ensures that there is active participation of stakeholders in the modeling of the artifact.

IEEE 1471

IEEE 1471 is the short name for a standard formally known as ANSI/IEEE 1471-2000, "Recommended Practice for Architecture Description of Software-Intensive Systems." IEEE 1471 enhances the content of an architectural description, in particular giving specific meaning to context, views, and viewpoints.

Unified Modeling Language (UML)

This approach represents three views of a system model. The **functional requirements view** (functional requirements of the system from the point of view of the user, including use cases); **the static structural view** (objects, attributes, relationships, and operations including class diagrams); and the **dynamic behavior view** (collaboration among objects and changes to the internal state of objects, including sequence, activity, and state diagrams).

UML Tutorials

Using UML Part Two – Behavioral Modeling Diagrams

by Sparx Systems

http://www.sparxsystems.com

Trademarks

Object Management Group, OMG, Unified Modeling Language, UML, are registered trademarks or trademarks of the Object Management Group, Inc.

All other product and / or company names mentioned within this document are used for identification purposes only, and may be trademarks or registered trademarks of their respective owners.

Table of Contents

Introduction

The Unified Modeling Language (UML) has become the de-facto standard for building Object-Oriented software. UML 2.1 builds on the already highly successful UML 2.0 standard, which has become an industry standard for modeling, design and construction of software systems as well as more generalized business and scientific processes. UML 2.1 defines thirteen basic diagram types, divided into two general sets: structural modeling diagrams and behavioral modeling diagrams. Part two will deal with behavioral modeling diagrams.

The Object Management Group (OMG) specification states:

> *"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."*

The important point to note here is UML is a "language" for specifying and not a method or procedure. The UML is used to define a software system – to detail the artifacts in the systems, to document and construct; it is the language the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process), but in itself does not specify that methodology or process.

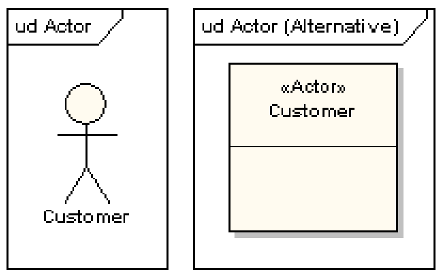
Behavior diagrams capture the varieties of interaction and instantaneous state within a model as it "executes" over time.

Use Case Model

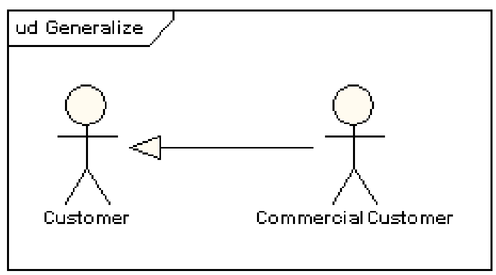
The use case model captures the requirements of a system. Use cases are a means of communicating with users and other stakeholders what the system is intended to do.

Actors

A use case diagram shows the interaction between the system and entities external to the system. These external entities are referred to as actors. Actors represent roles which may include human users, external hardware or other systems. An actor is usually drawn as a named stick figure, or alternatively as a class rectangle with the «actor» keyword.

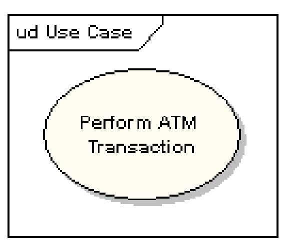


Actors can generalize other actors as detailed in the following diagram:

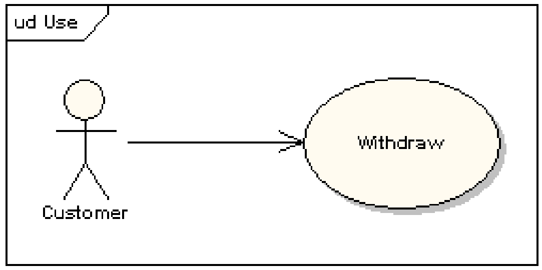


Use Cases

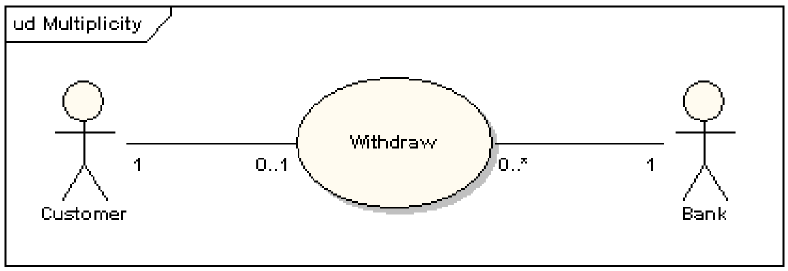
A use case is a single unit of meaningful work. It provides a high-level view of behavior observable to someone or something outside the system. The notation for a use case is an ellipse.



The notation for using a use case is a connecting line with an optional arrowhead showing the direction of control. The following diagram indicates that the actor "Customer" uses the "Withdraw" use case.



The uses connector can optionally have multiplicity values at each end, as in the following diagram, which shows that a customer may only have one withdrawal session at a time, but a bank may have any number of customers making withdrawals concurrently.



Use Case Definition

A use case typically includes:

- Name and description
- Requirements
- Constraints
- Scenarios
- Scenario diagrams
- Additional information.

Name and Description

A use case is normally named as a verb-phrase and given a brief informal textual description.

Requirements

The requirements define the formal functional requirements that a use case must supply to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract or promise that the use case will perform an action or provide some value to the system.

Constraints

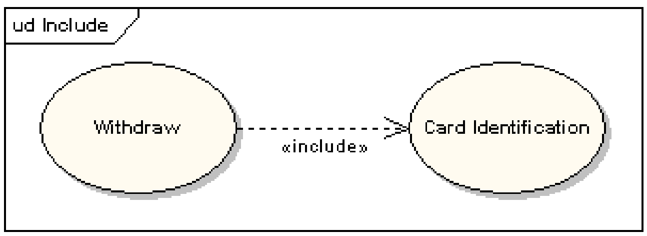
A constraint is a condition or restriction that a use case operates under, and includes pre-, post- and invariant conditions. A precondition specifies the conditions that need to be met before the use case can proceed. A post-condition is used to document the change in conditions that must be true after the execution of the use case. An invariant condition specifies the conditions that are true throughout the execution of the use case.

Scenarios

A scenario is a formal description of the flow of events that occur during the execution of a use case instance. It defines the specific sequence of events between the system and the external actors. It is normally described in text and corresponds to the textual representation of the sequence diagram.

Including Use Cases

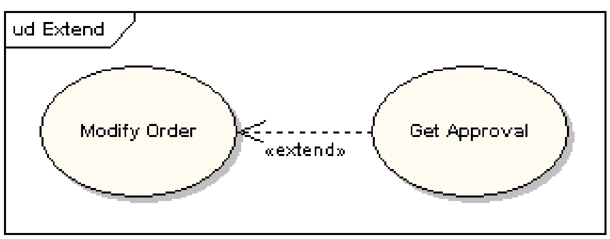
Use cases may contain the functionality of another use case as part of their normal processing. In general it is assumed that any included use case will be called every time the basic path is run. An example of this is to have the execution of the use case <Card Identification> to be run as part of a use case <Withdraw>.



Use Cases may be included by one or more Use Case, helping to reduce the level of duplication of functionality by factoring out common behavior into Use Cases that are re-used many times.

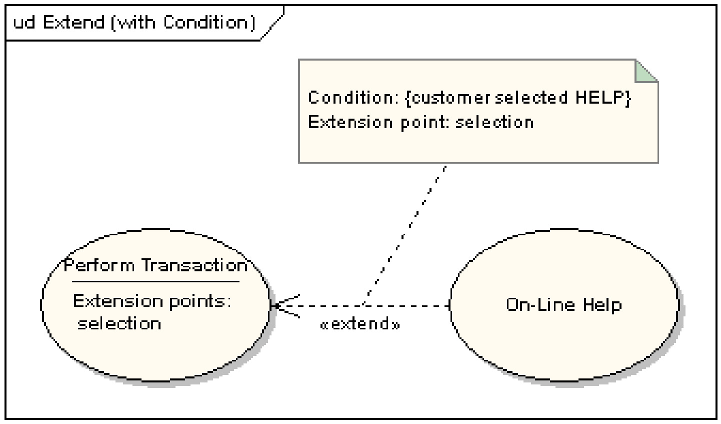
Extending Use Cases

One use case may be used to extend the behavior of another; this is typically used in exceptional circumstances. For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <Get Approval> use case may optionally extend the regular <Modify Order> use case.



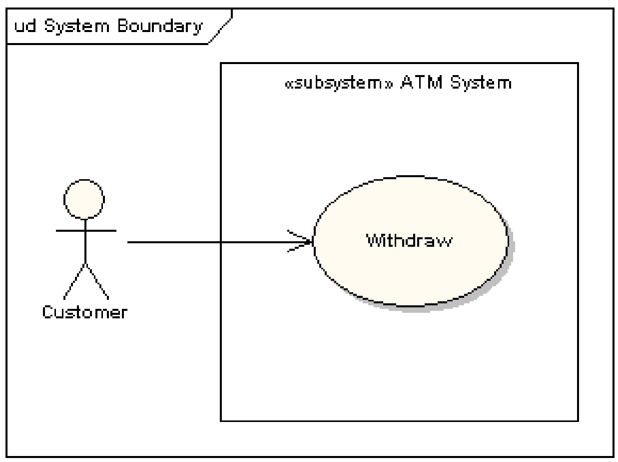
Extension Points

The point at which an extending use case is added can be defined by means of an extension point.



System Boundary

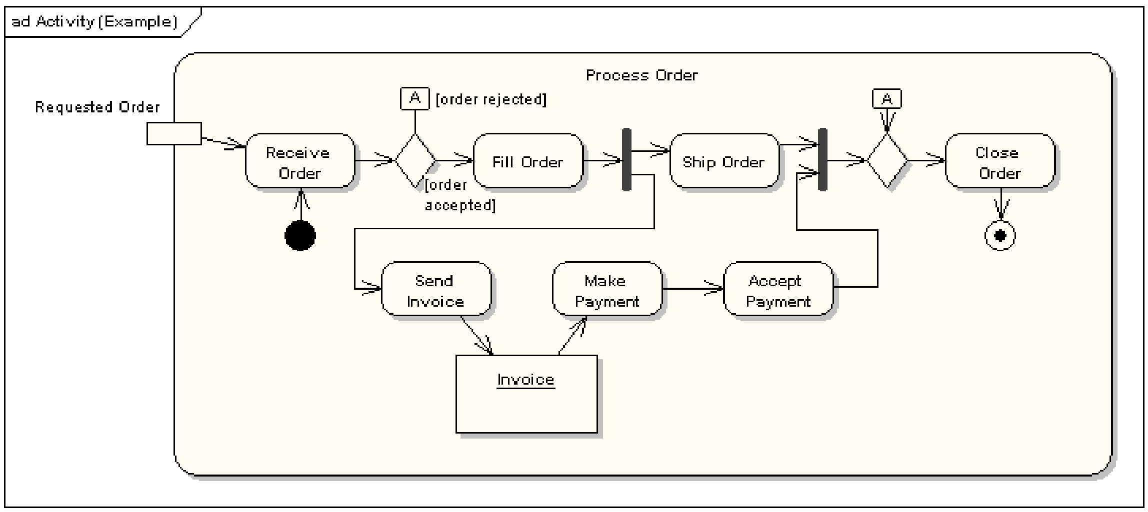
It is usual to display use cases as being inside the system and actors as being outside the system.



Activity Diagrams

In UML an activity diagram is used to display the sequence of activities. Activity diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. They may be used to detail situations where parallel processing may occur in the execution of some activities. Activity diagrams are useful for business modeling where they are used for detailing the processes involved in business activities.

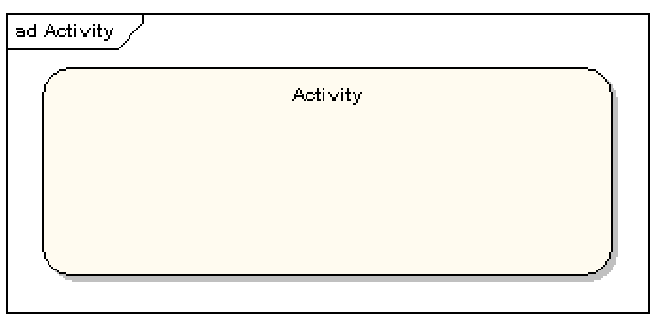
An example of an activity diagram is shown below.



The following sections describe the elements that constitute an Activity diagram.

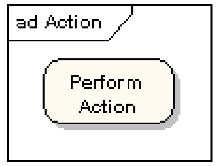
Activities

An activity is the specification of a parameterized sequence of behavior. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity.



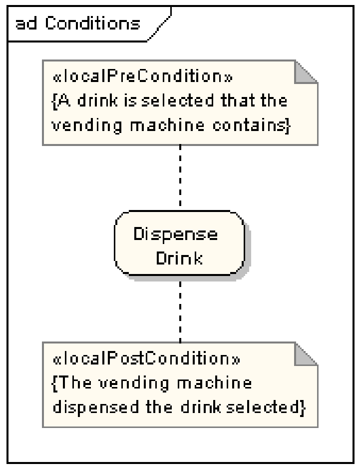
Actions

An action represents a single step within an activity. Actions are denoted by round-cornered rectangles.



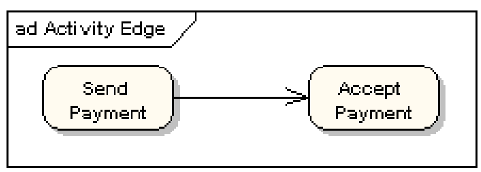
Action Constraints

Constraints can be attached to an action. The following diagram shows an action with local pre- and post-conditions.



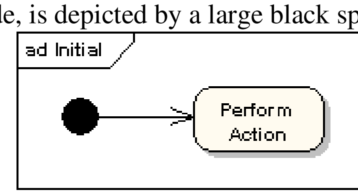
Control Flow

A control flow shows the flow of control from one action to the next. Its notation is a line with an arrowhead.



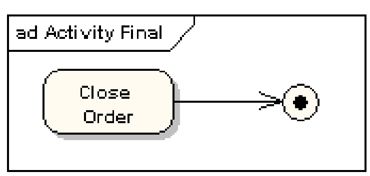
Initial Node

An initial, or start node, is depicted by a large black spot, as shown below.

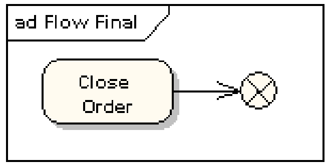


Final Node

There are two types of final node: activity and flow final nodes. The activity final node is depicted as a circle with a dot inside.



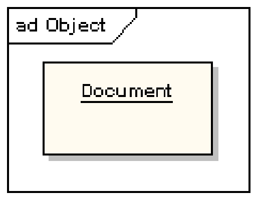
The flow final node is depicted as a circle with a cross inside.



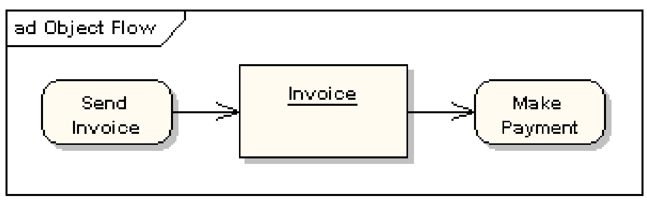
The difference between the two node types is that the flow final node denotes the end of a single control flow; the activity final node denotes the end of all control flows within the activity.

Objects and Object Flows

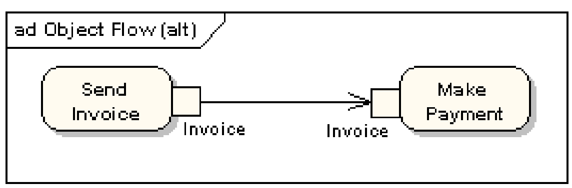
An object flow is a path along which objects or data can pass. An object is shown as a rectangle.



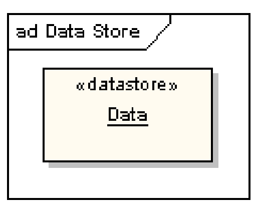
An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed.



An object flow must have an object on at least one of its ends. A shorthand notation for the above diagram would be to use input and output pins.

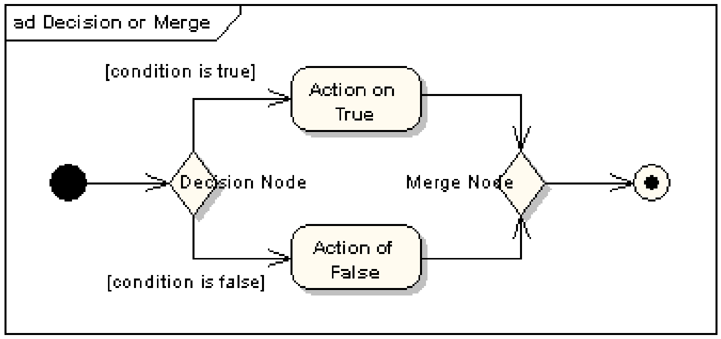


A data store is shown as an object with the «datastore» keyword.



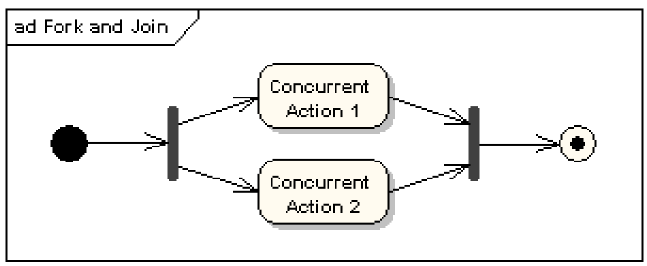
Decision and Merge Nodes

Decision nodes and merge nodes have the same notation: a diamond shape. They can both be named. The control flows coming away from a decision node will have guard conditions which will allow control to flow if the guard condition is met. The following diagram shows use of a decision node and a merge node.



Fork and Join Nodes

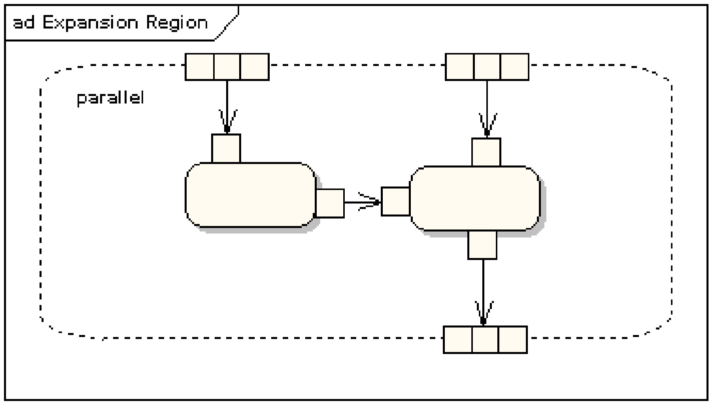
Forks and joins have the same notation: either a horizontal or vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of concurrent threads of control. The following diagram shows an example of their use.



A join is different from a merge, in that the join synchronizes two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received. A merge passes any control flows straight through it. If two or more inflows are received by a merge symbol, the action pointed to by its outflow is executed two or more times.

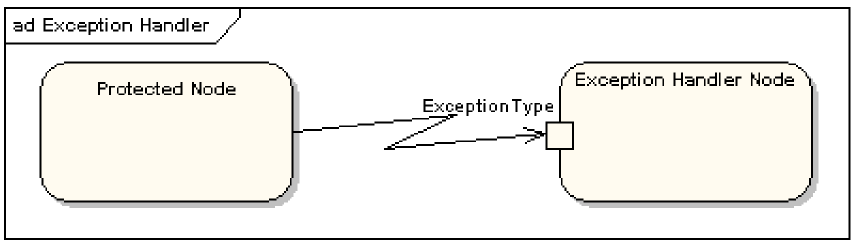
Expansion Region

An expansion region is a structured activity region that executes multiple times. Input and output expansion nodes are drawn as a group of three boxes representing a multiple selection of items. The keyword iterative, parallel or stream is shown in the top left corner of the region.



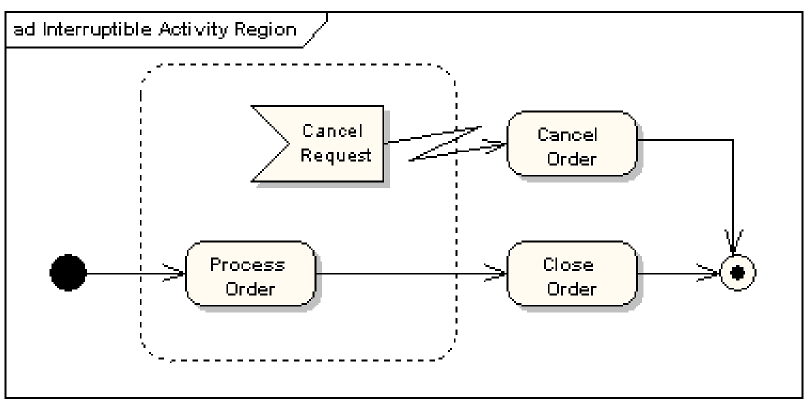
Exception Handlers

Exception handlers can be modeled on activity diagrams, as in the example below.



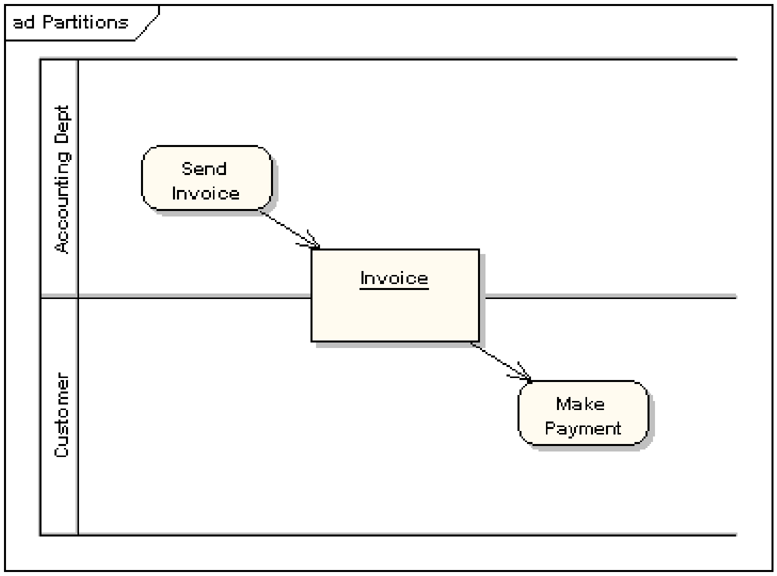
Interruptible Activity Region

An interruptible activity region surrounds a group of actions that can be interrupted. In the very simple example below, the "Process Order" action will execute until completion, when it will pass control to the "Close Order" action, unless a "Cancel Request" interrupt is received, which will pass control to the "Cancel Order" action.



Partition

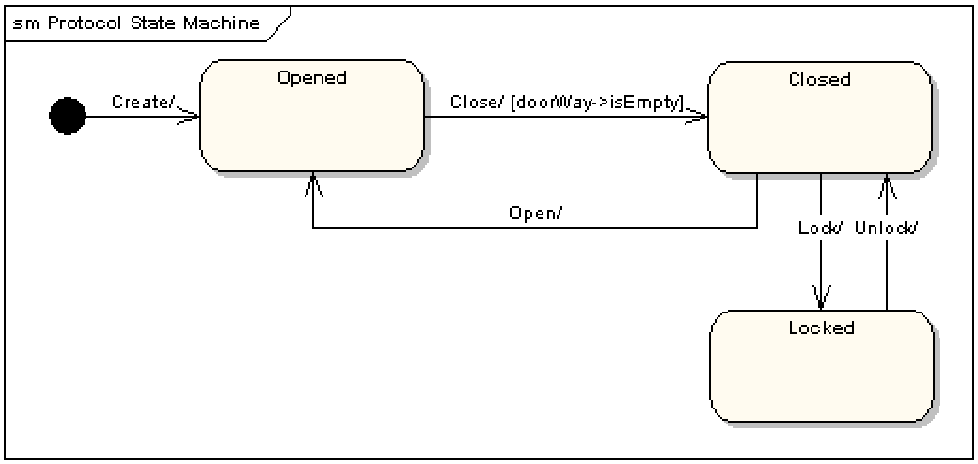
An activity partition is shown as either a horizontal or vertical swimlane
. In the following diagram, the partitions are used to separate actions within an activity into those performed by the accounting department and those performed by the customer.


ad Partitions
Accounting Dept
Customer
Send Invoice
Invoice
Make Payment

State Machine Diagrams

A state machine diagram models the behavior of a single object, specifying the sequence of events an object goes through during its lifetime in response to events.

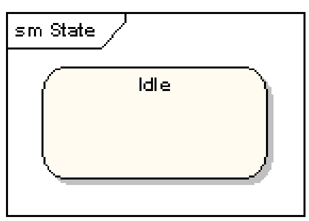
As an example, the following state machine diagram shows the states that a door goes through during its lifetime.



The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. Not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled. The syntax and conventions used in state machine diagrams will be discussed in full in the following sections.

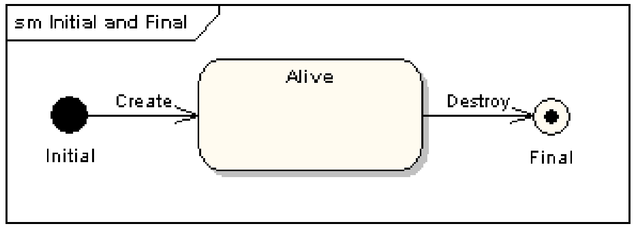
States

A State is denoted by a round-cornered rectangle with the name of the state written inside it.



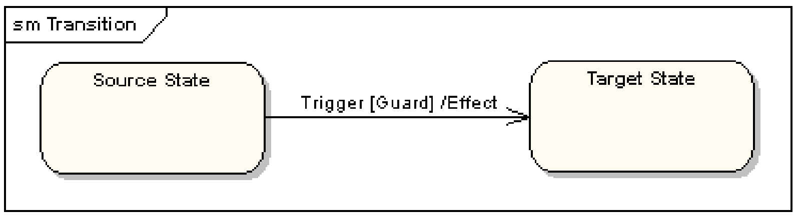
Initial and Final States

The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside, and may also be labeled with a name.



Transitions

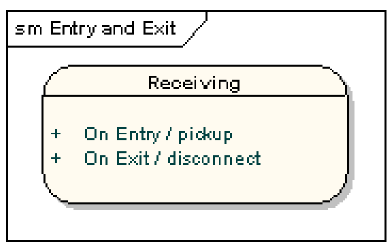
Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Actions

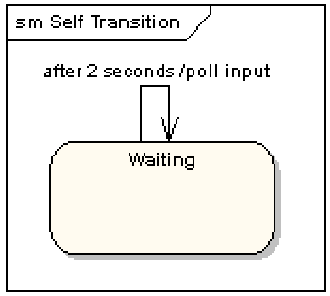
In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

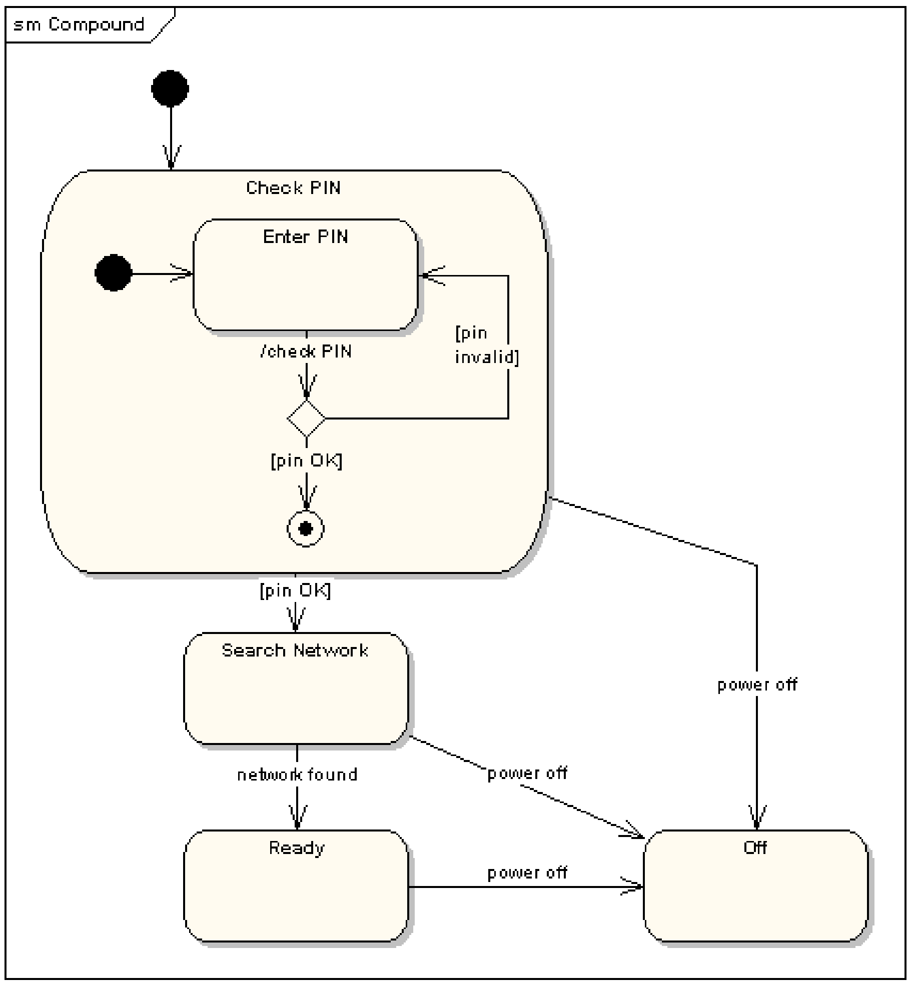
Self-Transitions

A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated with the transition.

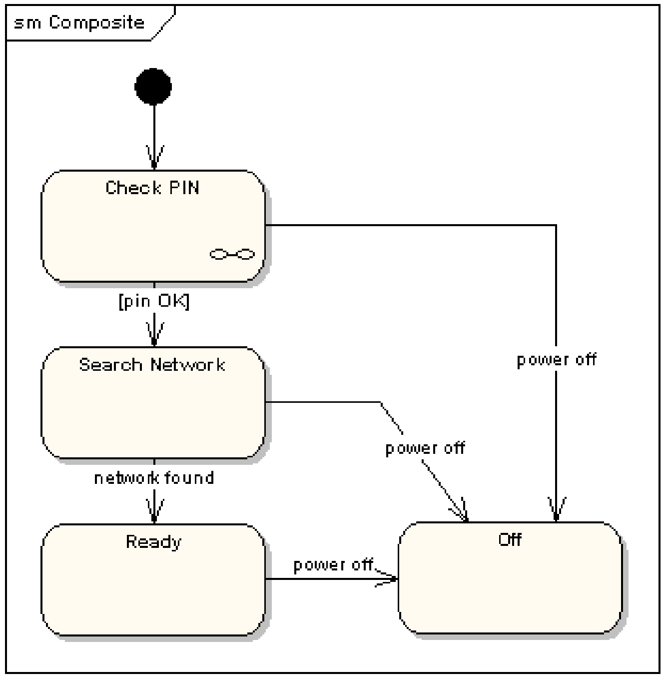


Compound States

A state machine diagram may include sub-machine diagrams, as in the example below.



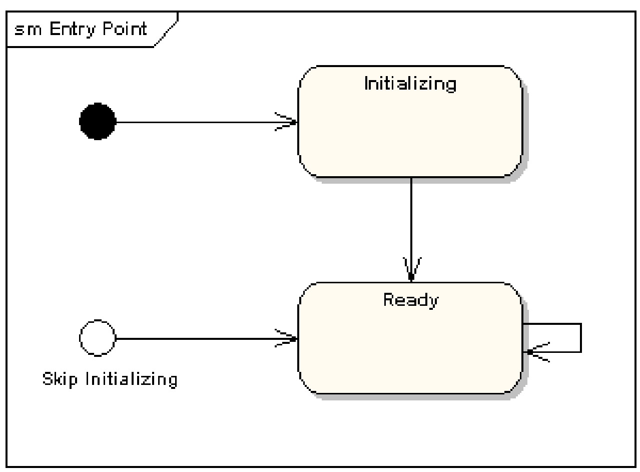
The alternative way to show the same information is as follows.



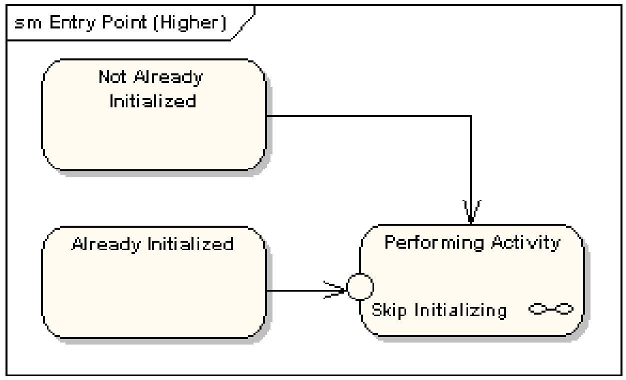
The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

Entry Point

Sometimes you won't want to enter a sub-machine at the normal Initial State. For example, in the following sub-machine it would be normal to begin in the Initializing state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the Ready state by transitioning to the named Entry Point.

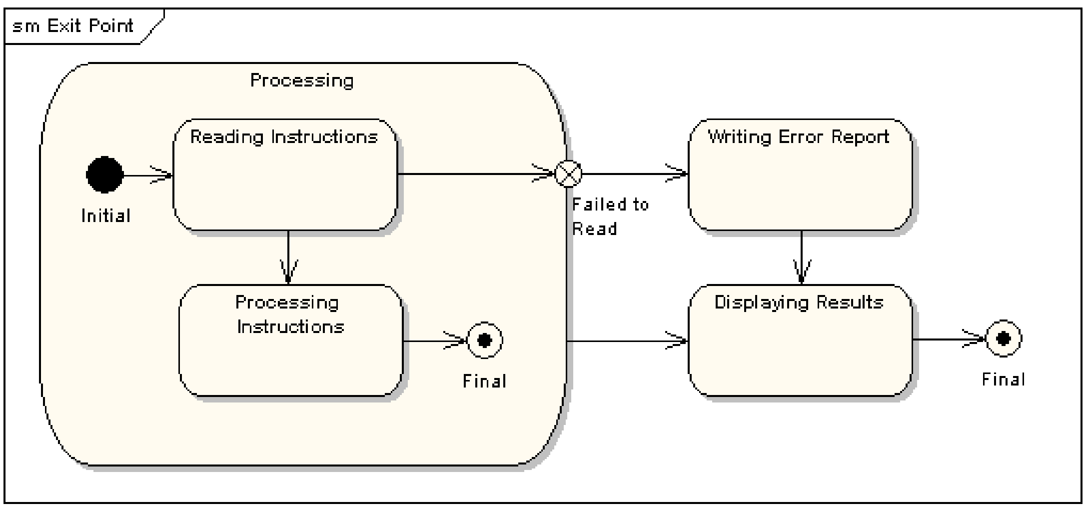


The following diagram shows the state machine one level up.



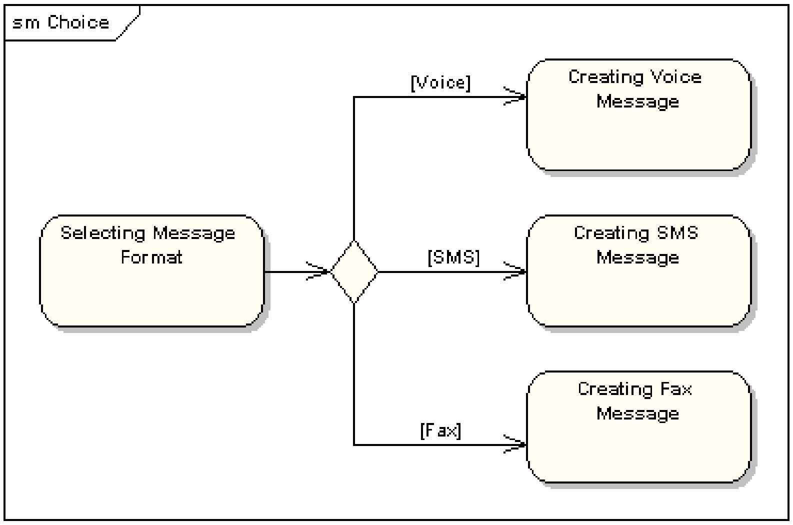
Exit Point

In a similar manner to entry points, it is possible to have named alternative exit points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.



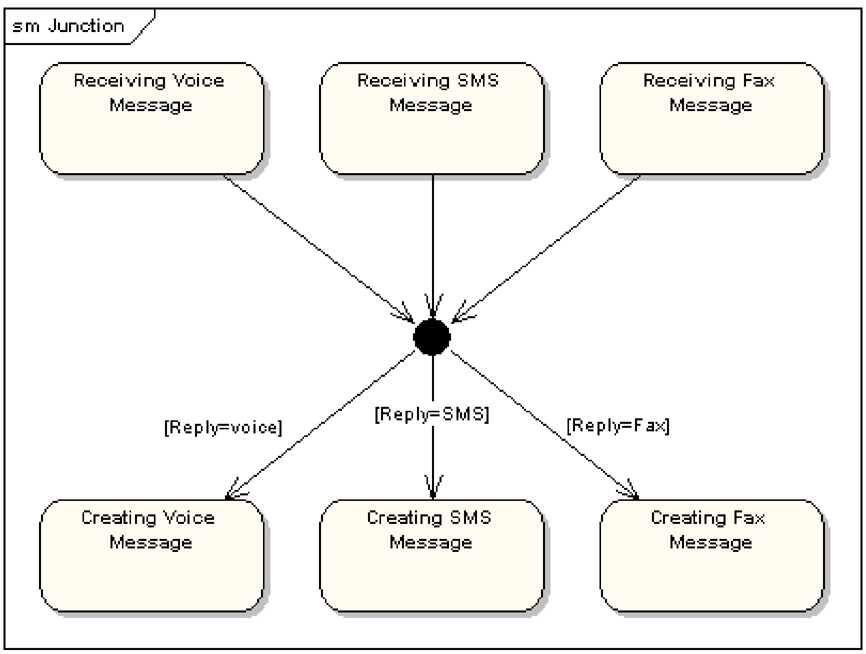
Choice Pseudo-State

A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.



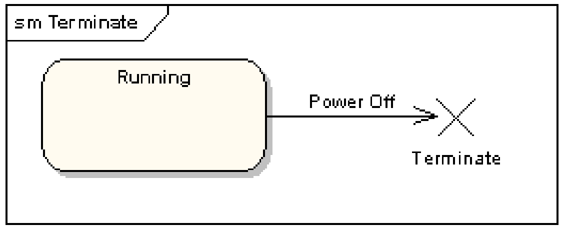
Junction Pseudo-State

Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free; a junction that splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.



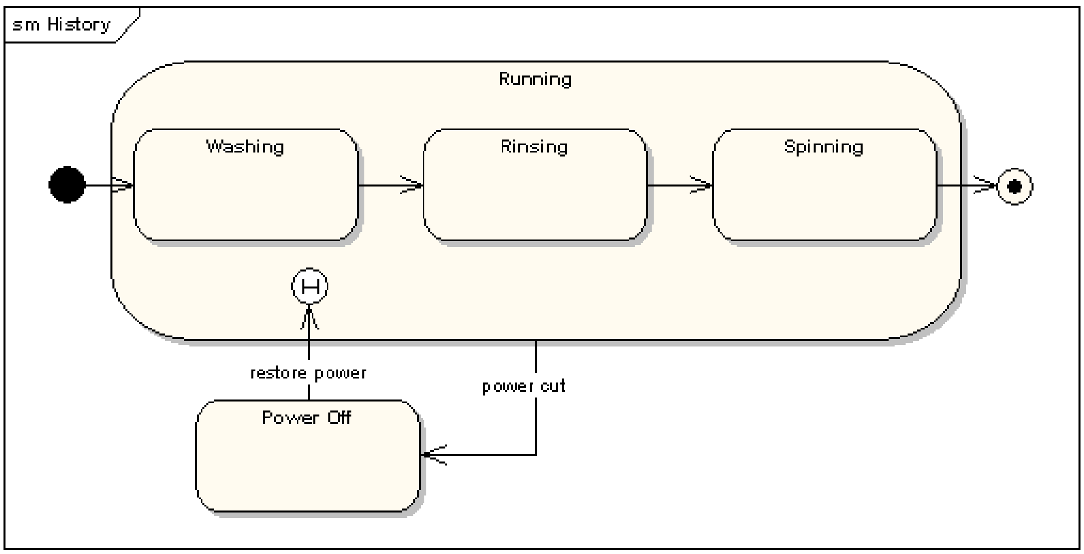
Terminate Pseudo-State

Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.



History States

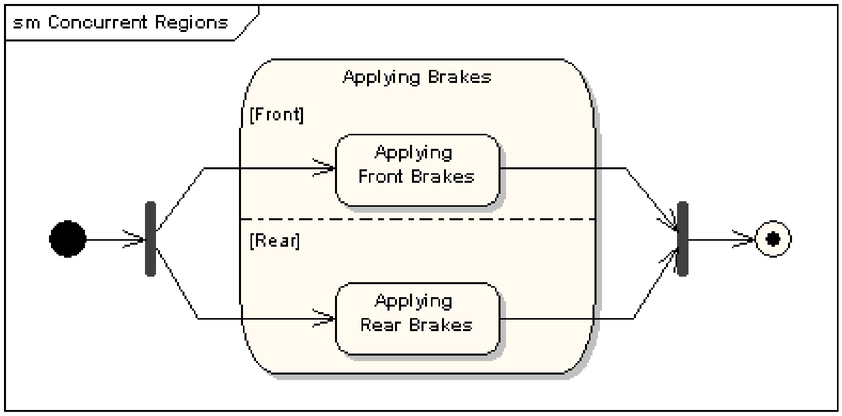
A History State is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning". If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. When the power is restored, the running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

Concurrent Regions

A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

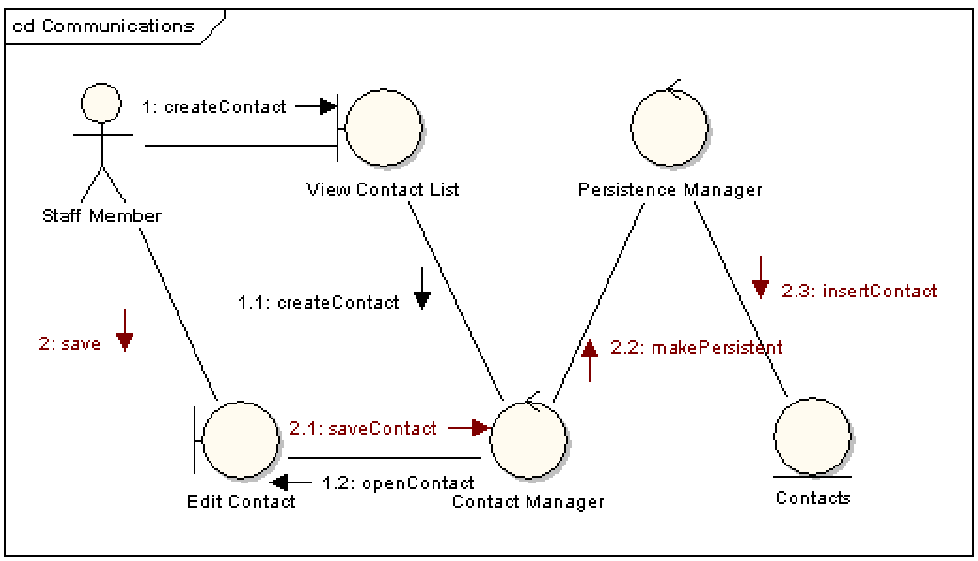

sm Concurrent Regions
Applying Brakes
[Front]
Applying
Front Brakes
[Rear]
Applying
Rear Brakes

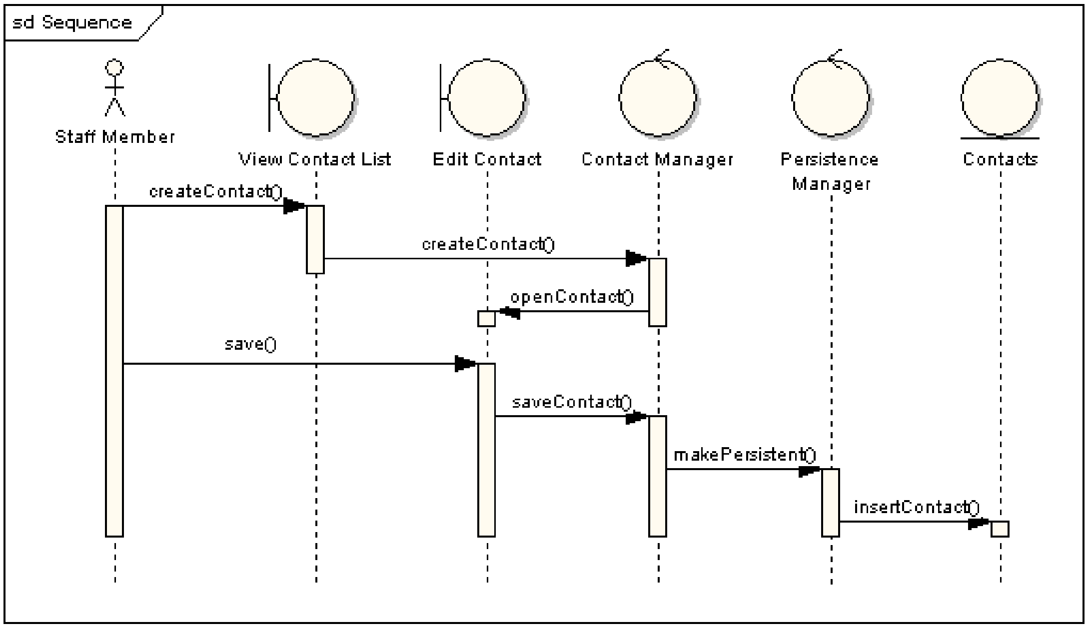
Communication Diagrams

A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but its primary focus is on object relationships.

On communication diagrams, objects are shown with association connectors between them. Messages are added to the associations and show as short arrows pointing in the direction of the message flow. The sequence of messages is shown through a numbering scheme.

The following two diagrams show a communication diagram and the sequence diagram that shows the same information. Although it is possible to derive the sequencing of messages in the communication diagram from the numbering scheme, it isn't immediately visible. What the communication diagram does show quite clearly though, is the full set of messages passed between adjacent objects.



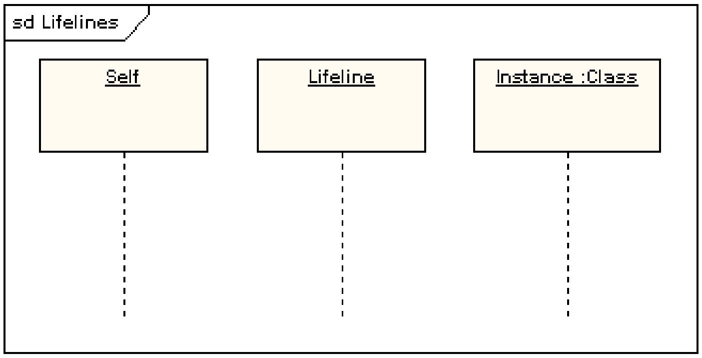

sd Sequence
Staff Member
View Contact List
Edit Contact
Contact Manager
Persistence Manager
Contacts
createContact()
createContact()
openContact()
save()
saveContact()
makePersistent()
insertContact()

Sequence Diagrams

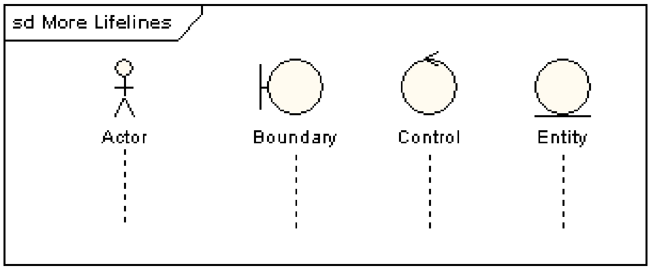
A sequence diagram is a form of interaction diagram that shows objects as lifelines running down the page, with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good for showing which objects communicate with which other objects, and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic.

Lifelines

A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. If its name is "self", that indicates the lifeline represents the classifier which owns the sequence diagram.

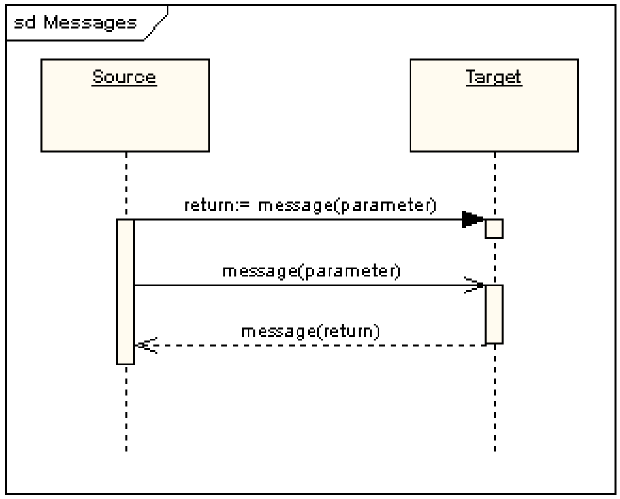


Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines.



Messages

Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. In the following diagram, the first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message; the second message is asynchronous (denoted by line arrowhead), and the third is the asynchronous return message (denoted by the dashed line).

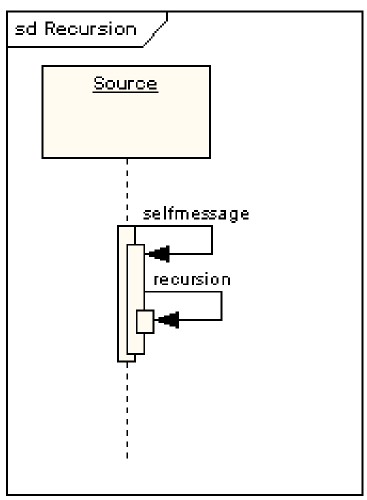


Execution Occurrence

A thin rectangle running down the lifeline denotes the execution occurrence, or activation of a focus of control. In the previous diagram, there are three execution occurrences. The first is the source object sending two messages and receiving two replies; the second is the target object receiving a synchronous message and returning a reply; and the third is the target object receiving an asynchronous message and returning a reply.

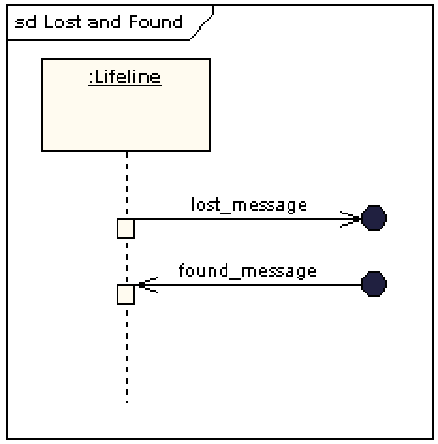
Self Message

A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline's execution occurrence.



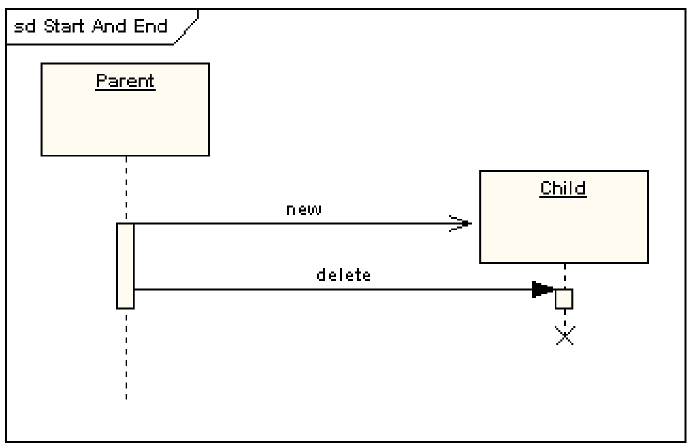
Lost and Found Messages

Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.



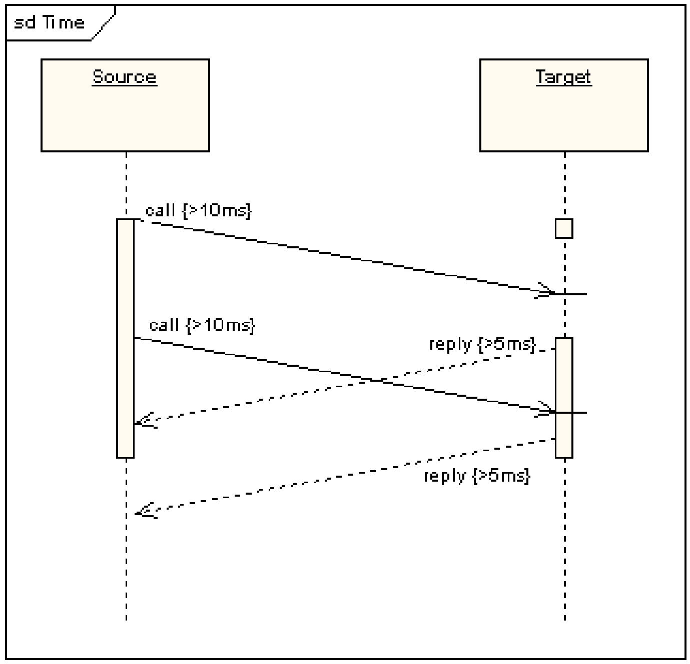
Lifeline Start and End

A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation. The following diagram shows an object being created and destroyed.



Duration and Time Constraints

By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modeling a real-time system, or even a time-bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.



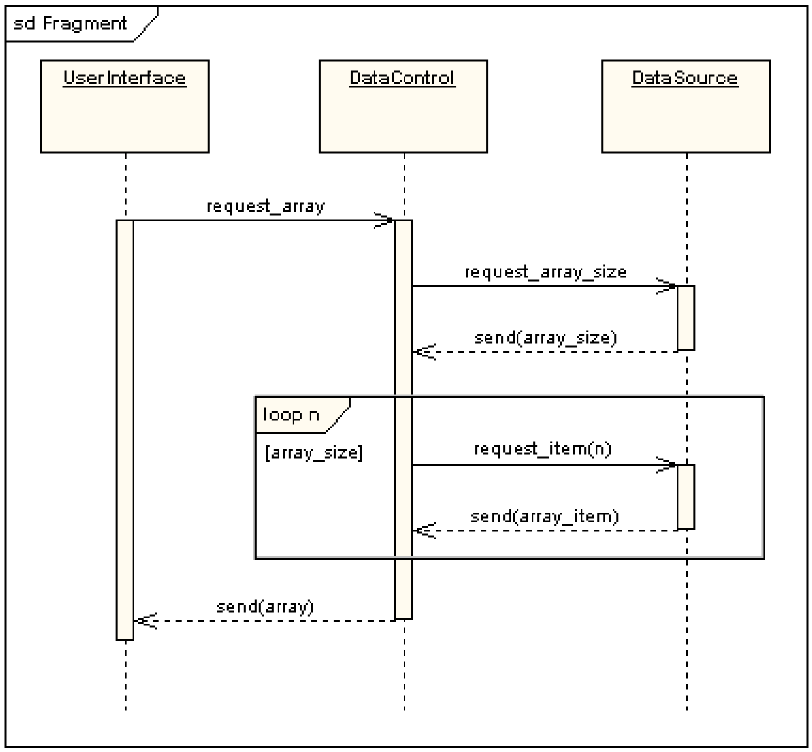
Combined Fragments

It was stated earlier that Sequence diagrams are not intended for showing complex procedural logic. While this is the case, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:

- Alternative fragment (denoted "alt") models if…then…else constructs.
- Option fragment (denoted "opt") models switch constructs.
- Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
- Parallel fragment (denoted "par") models concurrent processing.
- Weak sequencing fragment (denoted "seq") encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don't share a lifeline.
- Strict sequencing fragment (denoted "strict") encloses a series of messages which must be processed in the given order.
- Negative fragment (denoted "neg") encloses an invalid series of messages.
- Critical fragment encloses a critical section.
- Ignore fragment declares a message or message to be of no interest if it appears in the current context.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.

- Assertion fragment (denoted "assert") designates that any sequence not shown as an operand of the assertion is invalid.
- Loop fragment encloses a series of messages which are repeated.

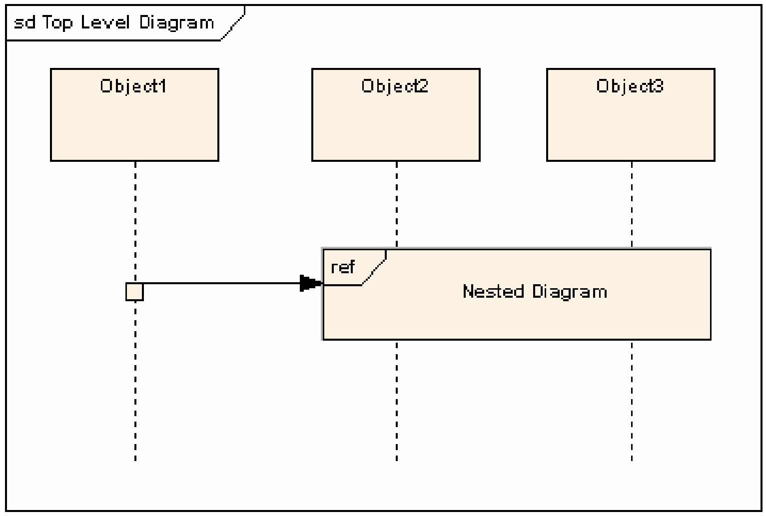
The following diagram shows a loop fragment.

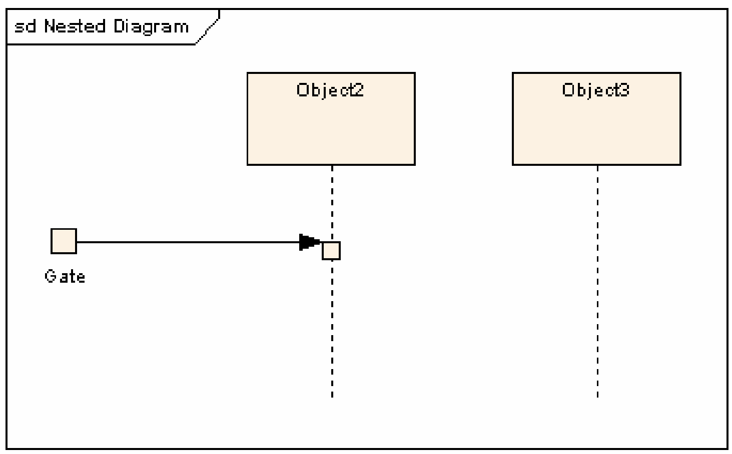


There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to another diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

Gate

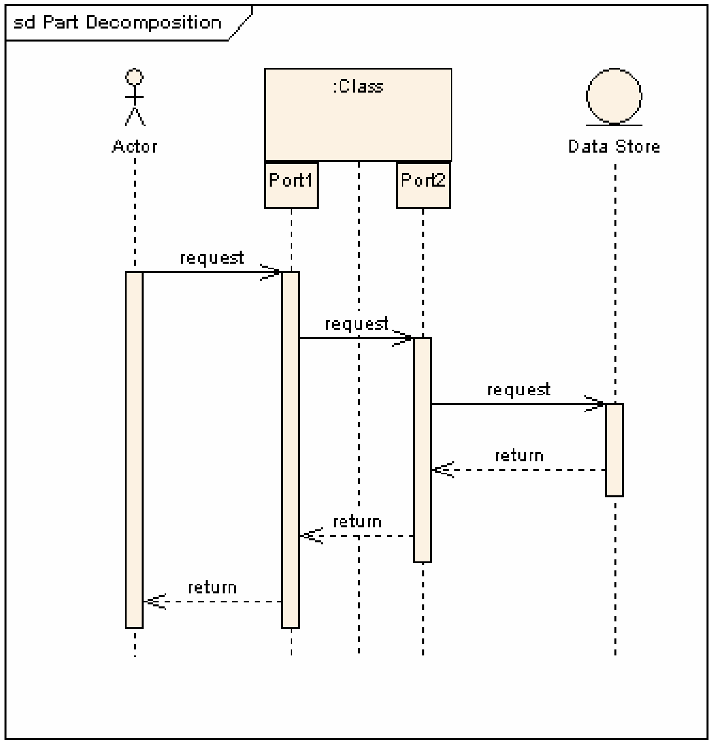
A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame. Diagram gates act as off-page connectors for sequence diagrams, representing the source of incoming messages or the target of outgoing messages. The following two diagrams show how they might be used in practice. Note that the gate on the top level diagram is the point at which the message arrowhead touches the reference fragment - there is no need to render it as a box shape.





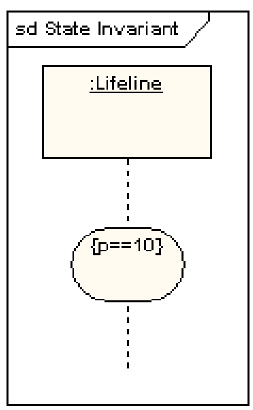
Part Decomposition

An object can have more than one lifeline coming from it. This allows for inter- and intra-object messages to be displayed on the same diagram.



State Invariant / Continuations

A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.



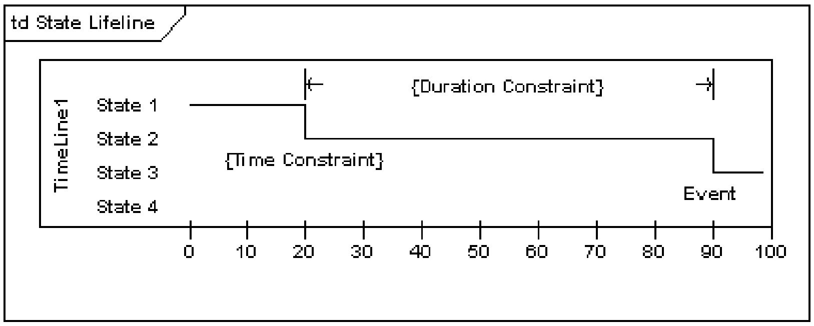
A continuation has the same notation as a state invariant, but is used in combined fragments and can stretch across more than one lifeline.

Timing Diagrams

UML timing diagrams are used to display the change in state or value of one or more elements over time. It can also show the interaction between timed events and the time and duration constraints that govern them.

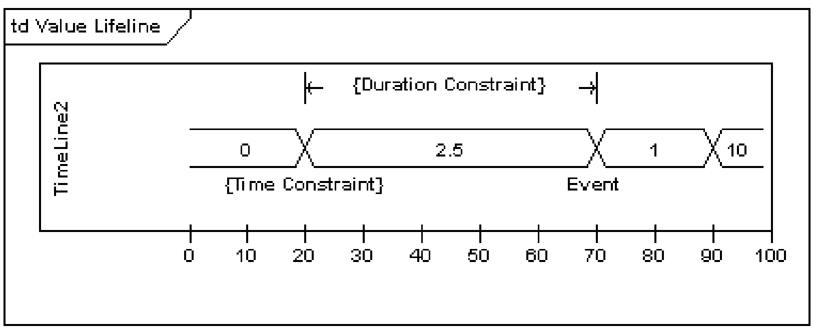
State Lifeline

A state lifeline shows the change of state of an item over time. The X-axis displays elapsed time in whatever units are chosen, while the Y-axis is labeled with a given list of states. A state lifeline is shown below.



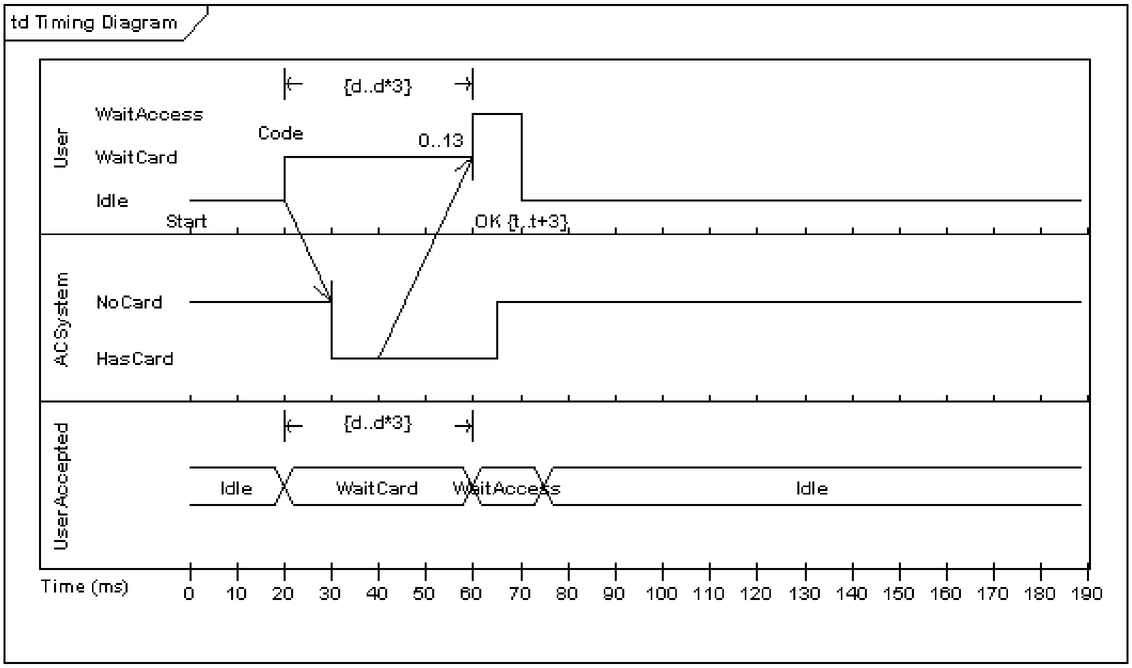
Value Lifeline

A value lifeline shows the change of value of an item over time. The X-axis displays elapsed time in whatever units are chosen, the same as for the state lifeline. The value is shown between the pair of horizontal lines which crosses over at each change in value. A value lifeline is shown below.



Putting it all together

State and value lifelines can be stacked one on top of another in any combination. They must have the same X-axis. Messages can be passed from one lifeline to another. Each state or value transition can have a defined event, a time constraint which indicates when an event must occur, and a duration constraint which indicates how long a state or value must be in effect for. Once these have all been applied, a timing diagram may look like the following.

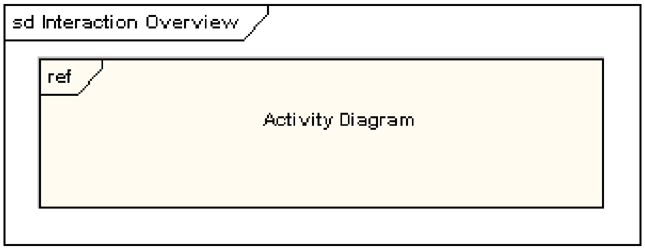

td Timing Diagram
User
WaitAccess
WaitCard
Idle
{d..d*3}
Code
0..13
Start
OK {t..t+3}
ACSystem
NoCard
HasCard
UserAccepted
{d..d*3}
Idle
WaitCard
WaitAccess
Idle
Time (ms)
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190

Interaction Overview Diagrams

An interaction overview diagram is a form of activity diagram in which the nodes represent interaction diagrams. Interaction diagrams can include sequence, communication, interaction overview, and timing diagrams. Most of the notation for interaction overview diagrams is the same for activity diagrams. For example initial, final, decision, merge, fork and join nodes are all the same. However, interaction overview diagrams introduce two new elements: interaction occurrences and interaction elements.

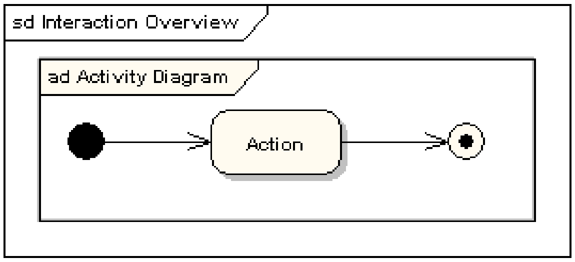
Interaction Occurrence

Interaction occurrences are references to existing interaction diagrams. An interaction occurrence is shown as a reference frame, that is, a frame with "ref" in the top-left corner. The name of the diagram being referenced is shown in the center of the frame.



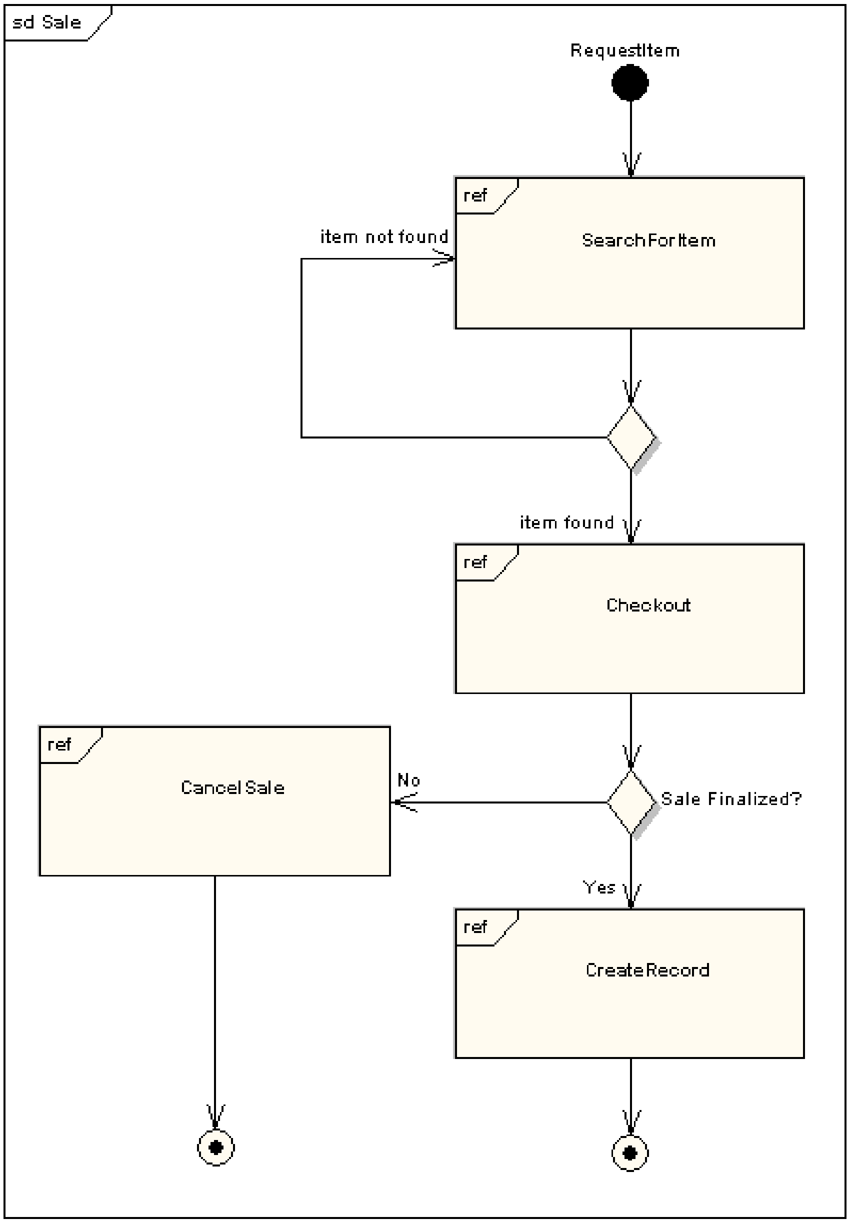
Interaction Element

Interaction elements are similar to interaction occurrences, in that they display a representation of existing interaction diagrams within a rectangular frame. They differ in that they display the contents of the references diagram inline.



Putting it all together

All the same controls from activity diagrams (fork, join, merge, etc.) can be used on interaction overview diagrams to put the control logic around the lower level diagrams. The following example depicts a sample sale process, with sub-processes abstracted within interaction occurrences.


sd Sale
RequestItem
ref
SearchForItem
item not found
item found
ref
Checkout
ref
Cancel Sale
No
Sale Finalized?
Yes
ref
CreateRecord

Recommended Reading

For more information, please refer to:

Sparx Systems' Web Site:	www.sparxsystems.com
Object Management Group's Web Site:	www.omg.com
Object Management Group's UML pages:	www.uml.org