

*Software Engineering: A Practitioner's Approach, 8/e*

Chapter 1  
Software and Software Engineering

copyright © 1996, 2001, 2005  
R.S. Pressman & Associates, Inc.

For University Use Only

May be reproduced ONLY for student use at the university level  
when used in conjunction with *Software Engineering: A Practitioner's Approach*.  
Any other reproduction or use is expressly prohibited.

# Proposed Evaluation Pattern

- Internal– 70 marks
  - Continuous assessment : 50 marks
    - 2 Quizzes : 20 marks
    - Project : 30 marks
      - Sprint 1 : 10 marks
      - Sprint 2 : 10 marks
      - Quiz : 10 marks
  - Periodical Test 1 : 30 marks reduced to 10 marks.
  - Periodical Test 2 : 30 marks reduced to 10 marks.
- External-30 marks
  - End Semester : 50 marks reduced to 30 marks.

# Software's Dual Role

- Software is a product
  - Delivers computing potential
  - Produces, manages, acquires, modifies, displays, or transmits information
- Software is a vehicle for delivering a product
  - Supports or directly provides system functionality
  - Controls other programs (e.g., an operating system)
  - Effects communications (e.g., networking software)
  - Helps build other software (e.g., software tools)

# What is Software?

---

*Software is: (1) **instructions** (computer programs) that when executed provide desired features, function, and performance; (2) **data structures** that enable the programs to adequately manipulate information and (3) **documentation** that describes the operation and use of the programs.*

# What is Software?

---

Software is a set of items or objects  
that form a “configuration” that  
includes

- programs
- documents
- data ...

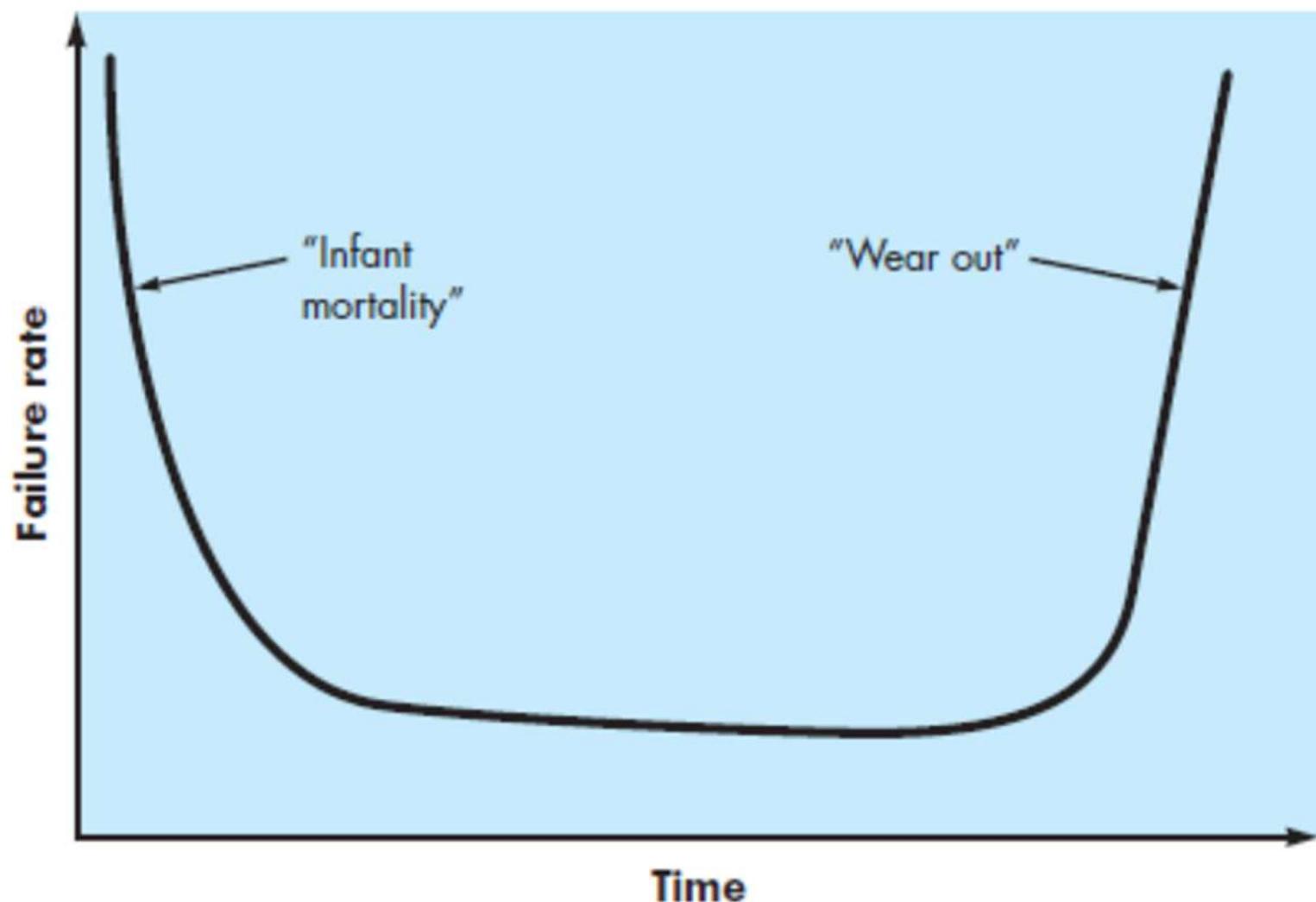
# What is Software?

software is  
engineered

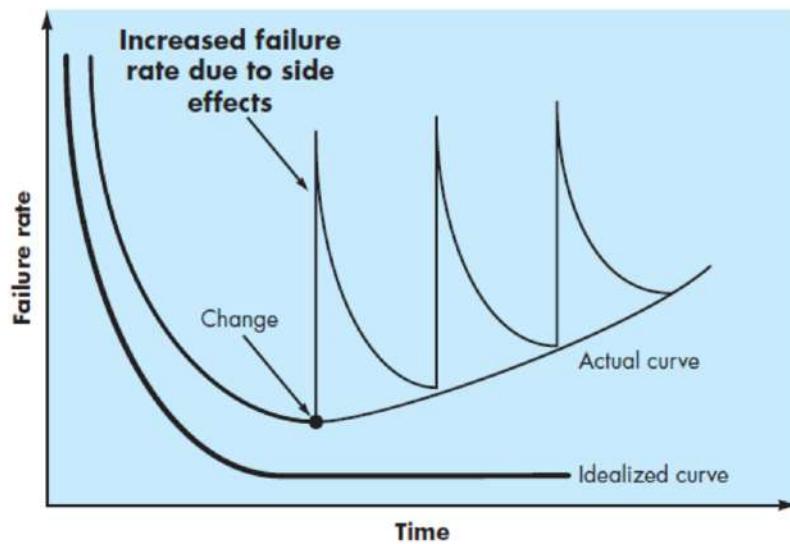
software  
doesn't wear  
out

software is  
complex

- To accomplish that, it's important to examine the characteristics of software that make it different from other things that human beings build.
- Software is a logical rather than a physical system element.
- Therefore, software has one fundamental characteristic that makes it considerably different from hardware: *Software doesn't "wear out."*



- Fig. depicts failure rate as a function of time for hardware. The relationship, often called the “bathtub curve,” indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.
- As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out*.



- Software is not susceptible to the environmental maladies that cause hardware to wear out.
- In theory, therefore, the failure rate curve for software should take the form of the “idealized curve”

- Undiscovered defects will cause high failure rates early in the life of a program.
- However, these are corrected and the curve flattens as shown.
- The idealized curve is a gross over simplification of actual failure models for software.
- However, the implication is clear—software doesn't wear out. But it does *deteriorate!*

- This contradiction can be explained by considering the actual curve in Figure . During its life, software will undergo change. **As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the “actual curve”.**
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

- Another aspect of wear illustrates the difference between hardware and software.
- When a hardware component wears out, it is replaced by a spare part.
- There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code.
- Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

# Software Applications

- system software
- application software
- engineering/scientific software
- embedded software
- product-line software
- WebApps (Web applications)
- AI software



- Millions of software engineers worldwide are hard at work on software projects in one or more of these categories.
- In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced.
- It is not uncommon for a young software engineer to work on a program that is older than she is! Past generations of software people have left a legacy in each of the categories we have discussed.
- Hopefully, the legacy to be left behind by this generation will ease the burden on future software engineers.



## Legacy Software:

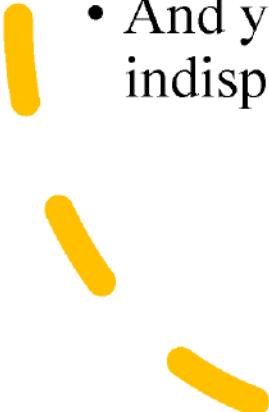
- Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection.
- Some of these are state-of-the-art software—just released to individuals, industry, and government.
- But other programs are older, in some cases *much* older.
- These older programs—often referred to as *legacy software* —have been the focus of continuous attention and concern since the 1960s.



- Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:
- Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.
- Liu and his colleagues [Liu98] extend this description by noting that “many legacy systems remain supportive to core business functions and are ‘indispensable’ to the business.”
- Hence, legacy software is characterized by longevity and business criticality.



- Unfortunately, there is sometimes one additional characteristic that is present in legacy software—***poor quality***.
- Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.
- And yet, these systems support “core business functions and are indispensable to the business.” What to do?





- The only reasonable answer may be: *Do nothing*, at least until the legacy system must undergo some significant change.
- If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed.





- However, as time passes, legacy systems often evolve for one or more of the following reasons:
  - The software must be adapted to meet the needs of new computing environments or technology.
  - The software must be enhanced to implement new business requirements.
  - The software must be extended to make it interoperable with other more modern systems or databases.
  - The software must be re-architected to make it viable within a evolving computing environment.
- Hence legacy system should be reengineered
  - Devise methodologies that are founded on the notion of evolution



# Software—New Categories

- Ubiquitous computing—wireless networks
- Netsourcing—the Web as a computing engine
- Open source—”free” source code open to the computing community (a blessing, but also a potential curse!)
- Also ...
  - Data mining
  - Grid computing
  - Cognitive machines
  - Software for nanotechnologies



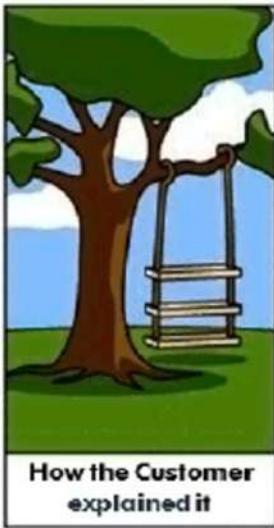


- 4 broad categories of software are evolving to dominate the industry:
  - Web apps
  - Mobile applications
  - Cloud computing
  - Product Line software



# Key Concepts

- Realities behind the software development
  - It follows that a concerted effort should be made to understand the problem before a software solution is developed.*
  - It follows that design becomes a pivotal activity.*
  - It follows that software should exhibit high quality.*
  - It follows that software should be maintainable*
- Software in all of its forms and across all of its application domains should be engineered.



**How the Customer  
explained it**



**What the Project  
Manager understood**



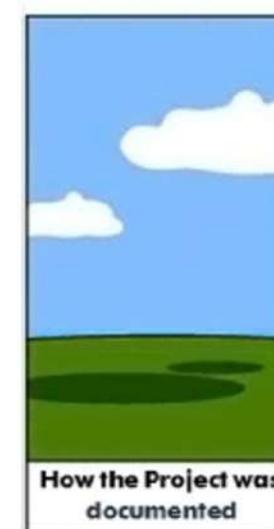
**How the Analyst  
designed it**



**What the  
Programmer wrote**



**What the Business  
Consultant presented**



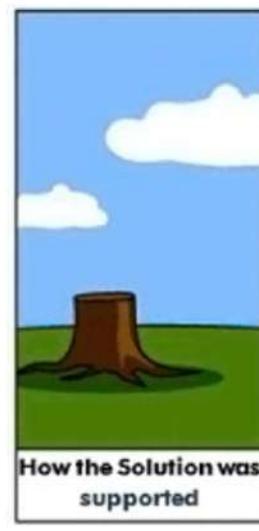
**How the Project was  
documented**



What Operations installed



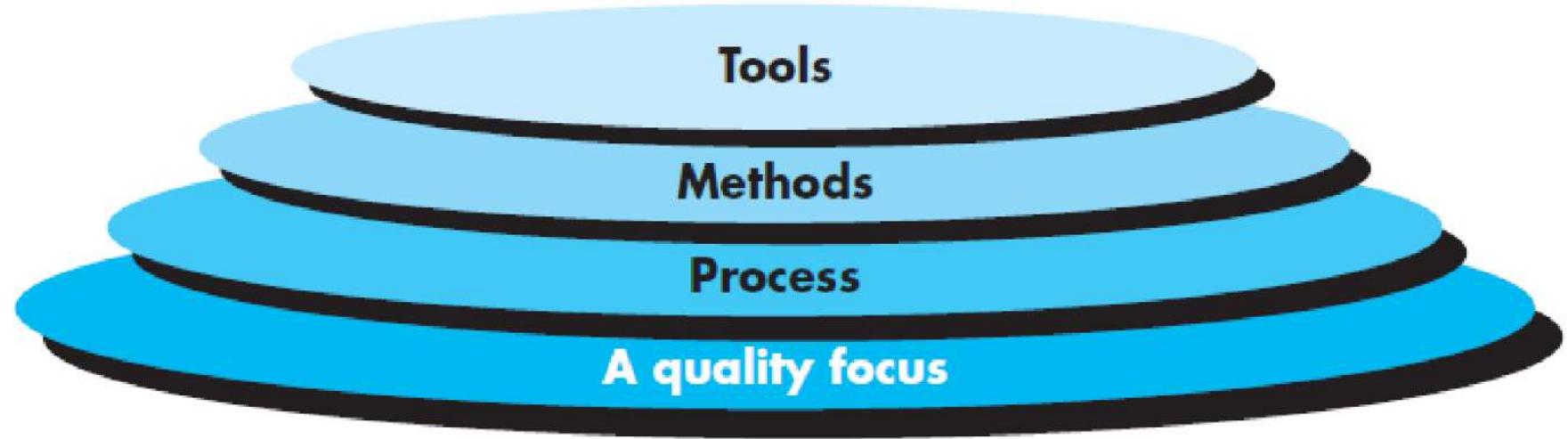
How the Customer was billed



How the Solution was supported



What the Customer really needed



- Defining Software Engineering as proposed by IEEE
- *–(1)The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
- *(2)The study of approaches as in(1).*
- –What about other approaches?
- Adaptability and Agility
- –Layered technology

- Any engineering approach must rest on an organizational commitment to quality.
- Process
  - glue that holds the technology layers together & enables
    - rational and timely development of computer software
    - Forms the basis for management control of software projects and establishes the context in which
      - technical methods are applied
      - Work products are produced
      - Milestones are established
      - Quality is ensured &
      - Change is properly managed



- **Methods** encompass a broad array of tasks that include
  - Communication
  - Requirements analysis
  - Design modeling
  - Program construction
  - Testing &
  - Support
- Rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.
- **Tools** provide automated or semi-automated support for the process and methods

## The Software Process

- A *process is a collection of activities, actions, and tasks that are performed when some work product is to be created*
  - Activity: strives to achieve broad objective
  - Action: encompasses a set of tasks that produce a major work product
  - Task: focuses on a small, but well-defined objective that produces a tangible outcome
- It is an adaptable approach that enables the people doing the work to pick & choose the appropriate set of work actions and tasks

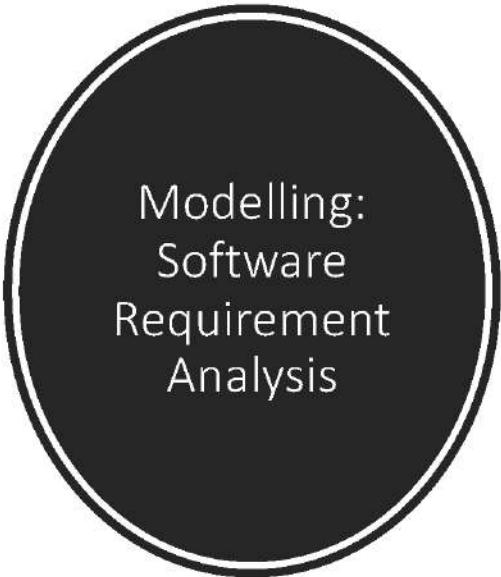
- The Process Framework
  - Foundation for a software engineering process through a small number of framework activities
    - Communication
    - Planning
    - Modeling
    - Construction
    - Deployment
  - Encompasses a set of umbrella activities that are applicable across the entire software process.



## Generic Process Framework

- **Communication**
  - Involves communication among the customer and other stake holders; encompasses requirements gathering
- **Planning**
  - Establishes a plan for software engineering work; addresses technical tasks, resources, work products, and work schedule
- **Modelling (Analyse, Design)**
  - Encompasses the creation of models to better understand the requirements and the design
- **Construction (Code, Test)**
  - Combines code generation and testing to uncover errors
- **Deployment**
  - Involves delivery of software to the customer for evaluation and feedback

3



- Helps software engineers to better understand the problem they will work to solve
- Encompasses the set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end-users will interact with the software
- Uses a combination of text and diagrams to depict requirements for data, function, and behavior
  - Provides a relatively easy way to understand and review requirements for correctness, completeness and consistency



## Modelling: Software Design

- Brings together customer requirements, business needs, and technical considerations to form the “blueprint” for a product
- Creates a model that provides detail about software data structures, software architecture, interfaces, and components that are necessary to implement the system
- Architectural design
  - Represents the structure of data and program components that are required to build the software
  - Considers the architectural style, the structure and properties of components that constitute the system, and interrelationships that occur among all architectural components
- User Interface Design
  - Creates an effective communication medium between a human and a computer
  - Identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype
- Component-level Design
  - Defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component



–Typical activities include,

- •Software Project Tracking and Control
- •Risk Management
- •Software Quality Assurance
- •Formal Technical Reviews
- • Measurement
- •Software Configuration Management
- •Reusability Management
- •Work Product preparation and production

## Software process

### Process framework

#### Umbrella activities

framework activity # 1

software engineering action #1..1

Task sets

work tasks  
work products  
quality assurance points  
project milestones

⋮

software engineering action #1..k

Task sets

work tasks  
work products  
quality assurance points  
project milestones

⋮

framework activity # n

software engineering action #n..1

Task sets

work tasks  
work products  
quality assurance points  
project milestones

⋮

software engineering action #n..m

Task sets

work tasks  
work products  
quality assurance points  
project milestones



## Process Adaptation

It should be agile and adaptable

–Therefore, a process adopted for one project might be significantly different than a process adopted for another project

How do process models differ from one another?

- Interdependencies among activities and tasks
- Degree to which work tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which Quality assurance, Project Tracking and Control activities are applied
- Degree of detail and severity of the process applied
- Degree to which customers are involved
  - Level of autonomy to the team
- Degree to which team organization and roles are prescribed

---

Prescriptive Process Models exists for the past 30 years

- objectives not achieved
- Due to rigidity and without adaptation
- Increase the level of bureaucracy associated and unintentionally create difficulty for developers and customers.

Agile Process Models in recent years

- Project agility
- More informal but no less effective approach
- Emphasize maneuverability and adaptability

# Process Models

---

- We examine a number of “prescriptive” software process models
- Why prescriptive?(giving exact rules)
- –Prescribe a set of process elements for each project
  - Framework activities
  - Software engineering actions
  - Tasks
  - Work products
  - Quality assurance &
  - Change control mechanisms
- –Each process model also prescribes a workflow that invoke each framework activity in a different manner

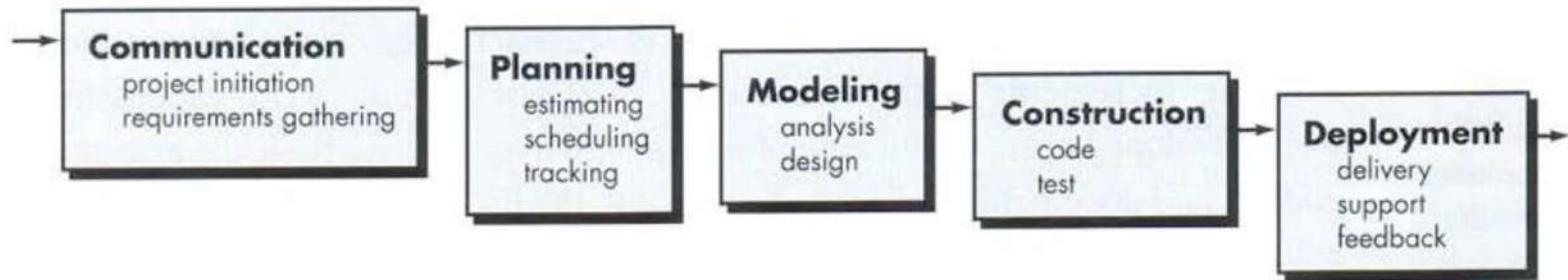
---

- WATERFALL MODEL

- –Classic life cycle; systematic; sequential approach to software development that begins with customer specifications of requirements & progresses through planning, modeling, construction and deployment, concluding in on-going support of the completed s/w.
- Oldest paradigms for s/w engg and sometimes fail; problems encountered are:

**FIGURE 3.1**

The waterfall model



- Real projects rarely follow the sequential flow that the model proposes
- Often difficult for the customer to state all the requirements explicitly
- Customer must have patience
- Some project teams must wait for other team members to complete dependent tasks – “blocking states”

## Incremental Process Models

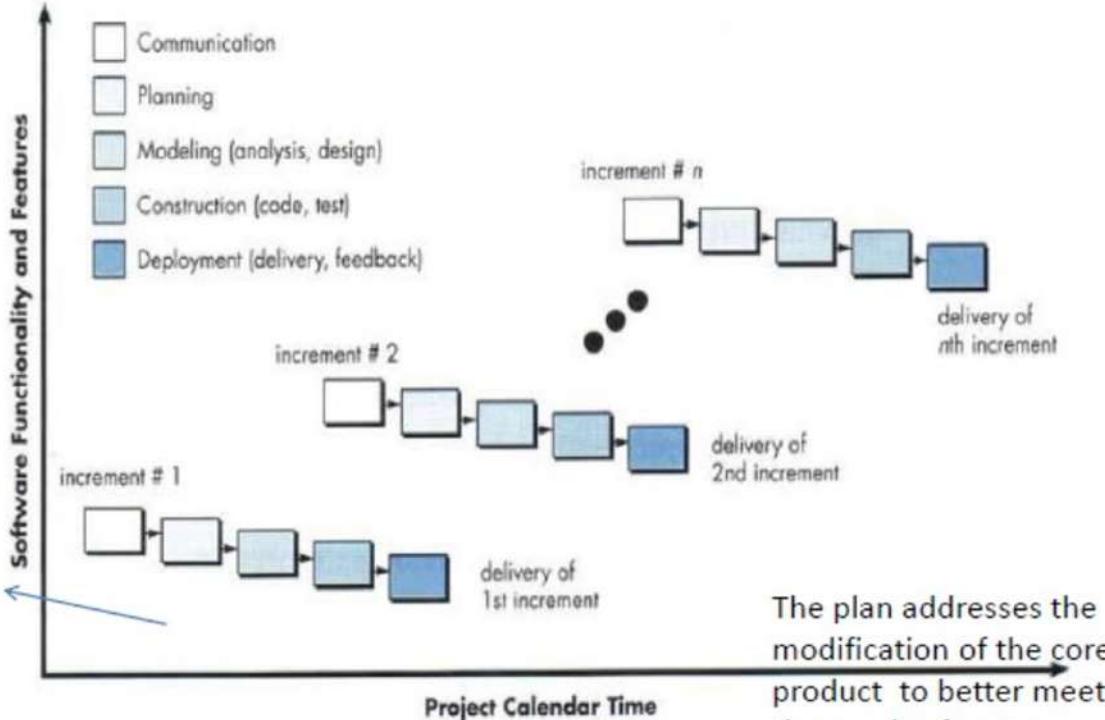
- –A compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases
- –A process model that is designed to produce the software in increments is chosen
  - –Incremental Model
    - Combines elements of the waterfall model applied in an iterative fashion
      - Applies **linear sequences** in a staggered fashion as calendar time progresses –each linear sequences produces deliverable “increments” of the software
      - Process flow for any increment may incorporate the prototype model

**FIGURE 3.2**

The  
incremental  
model

Often the  
“core  
product”

Customer uses the product; as a result of  
use and evaluation, a plan is developed for  
the next increment





## Rapid Application Development (RAD) model

- –An incremental software process model that emphasizes a short development cycle
- –“high-speed” adaptation of the waterfall model
- –If requirements are well understood and project scope is constrained , the RAD process enables to create a “fully functional system” within a very short time period

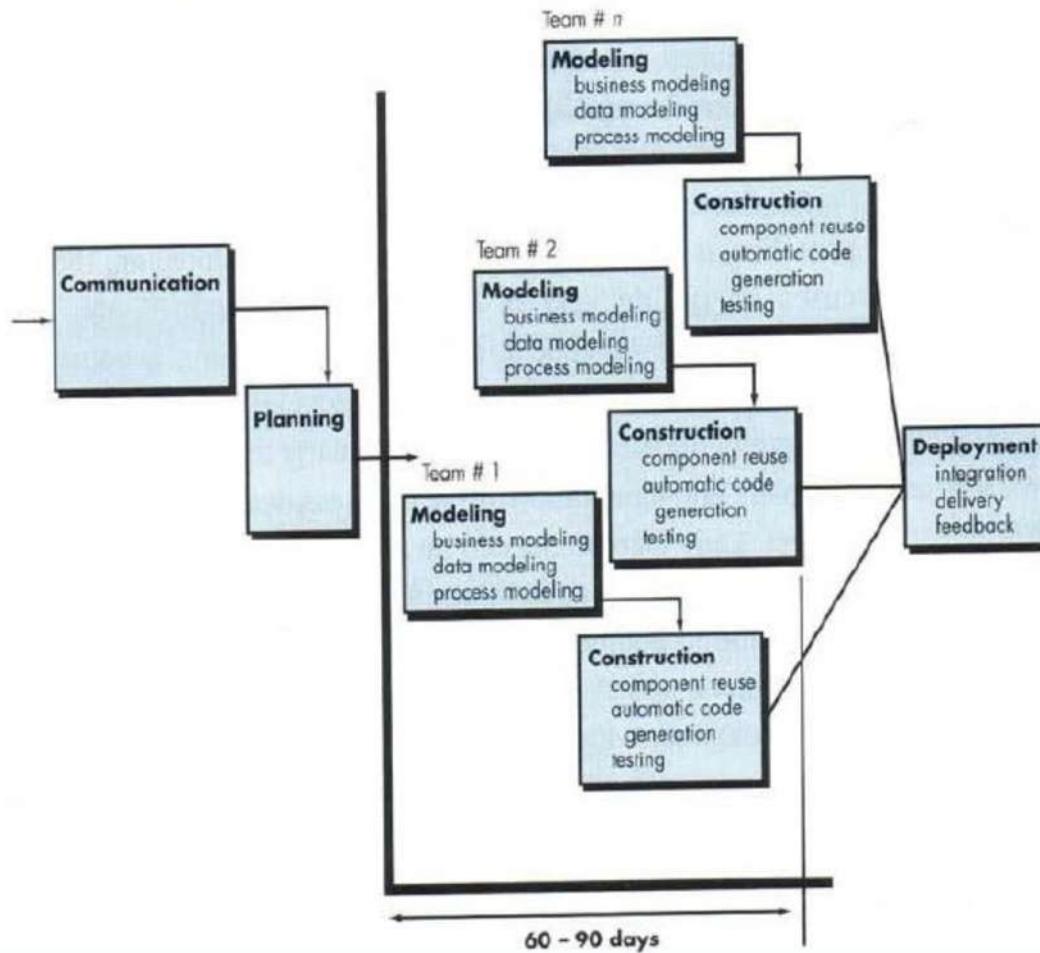


## Drawbacks

- Requires sufficient human resources to create the right number of RAD teams
- If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a limited time, RAD projects fail
- Improper modularization will be problematic
- For high-performance, tuning the interfaces to system components will not work for RAD
- May not be appropriate for high technical risks

**FIGURE 3.3**

The RAD model

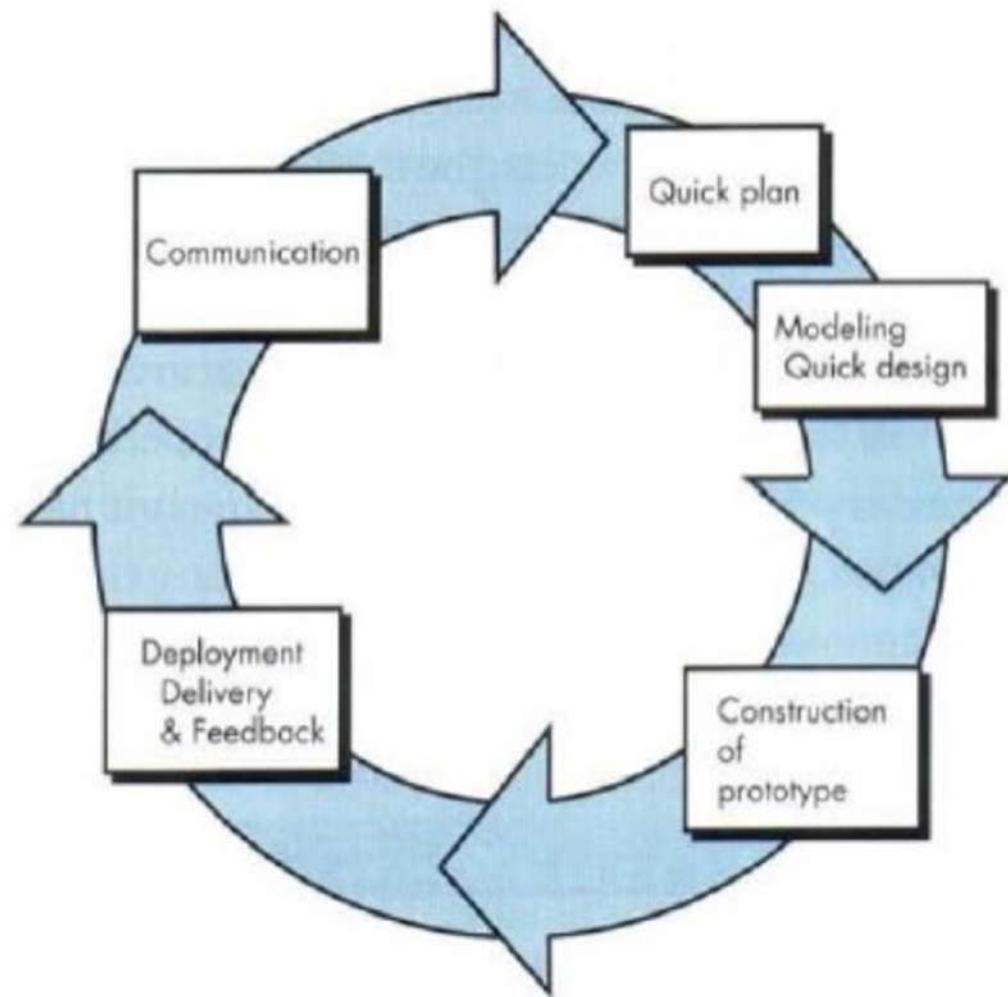




- **Evolutionary Process Models**

- Produces an increasingly more complete version of the software with each iteration
- Iterative
- Prototyping model
- Customer may not identify detailed input, processing or output requirements
- Developer may be unsure of the efficiency of an algorithm
- Adaptability of an OS
- For the above situations, Prototyping Paradigm is the best approach
- Quick Design

**FIGURE 3.4**  
The proto-  
typing model



- The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software.
- Identify whatever requirements are known, and outline areas where further definition is mandatory.
- A prototyping iteration is planned quickly, and modeling (in the form of a “quick design”) occurs.
- A quick design focuses on a representation of those aspects of the software that will be visible to end users.  
(e.g., human interface layout or output display formats).

- 
- The quick design leads to the construction of a prototype.
  - The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.
  - Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done

- 
- Ideally, the prototype serves as a mechanism for identifying software requirements.
  - If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.
  - But what do you do with the prototype when it has served the purpose described earlier? Brooks [Bro95] provides one answer:
    - In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three.
    - There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

- The prototype can serve as “the first system.” The one that Brooks recommends you throw away.
- But this may be an idealized view.
- Although some prototypes are built as “throwaways,” others are evolutionary in the sense that the prototype slowly evolves into the actual system.
- Both stakeholders and software engineers like the prototyping paradigm.
- Users get a feel for the actual system, and developers get to build something immediately.
- Yet, prototyping can be problematic for the following reasons:

- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.
- When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that “a few fixes” be applied to make the prototype a working product.
- Too often, software development management relents.

- As a software engineer, you often make implementation compromises in order to get a prototype working quickly.
- An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability.
- After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate.
- The less-than-ideal choice has now become an integral part of the system.

- Although problems can occur, prototyping can be an effective paradigm for software engineering.
- The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements.
- It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

## Spiral Model

- The classical models do not deal with the uncertainty with the software projects.
- A lot risk assessment and analysis form a part of the software development.
- This was first realized by Barry Boehm, who introduced the factor of “project risk” into the life cycle model which resulted in the spiral model in 1986
- Couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model

Concept Development Project in the first circuit;  
New Product Development Project  
Product Enhancement Project

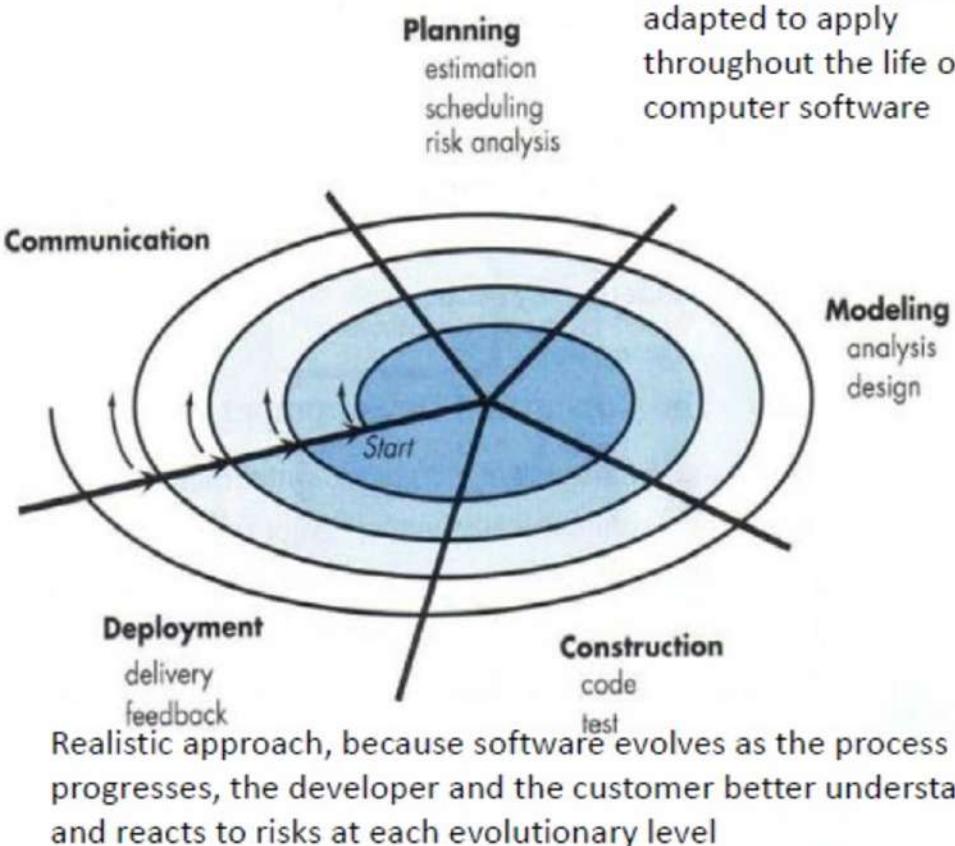
**FIGURE 3.5**

A typical spiral model

Risk is considered as each evolution is made

**Anchor point milestones** - a combination of work products & conditions that are attained along the path of spiral are noted

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of computer software



- Therefore, the first circuit around the spiral might represent a “concept development project” that starts at the core of the spiral and continues for multiple iterations until concept development is complete.
- If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a “new product development project” commences.
- The new product will evolve through a number of iterations around the spiral.
- Later, a circuit around the spiral might be used to represent a “product enhancement project.”

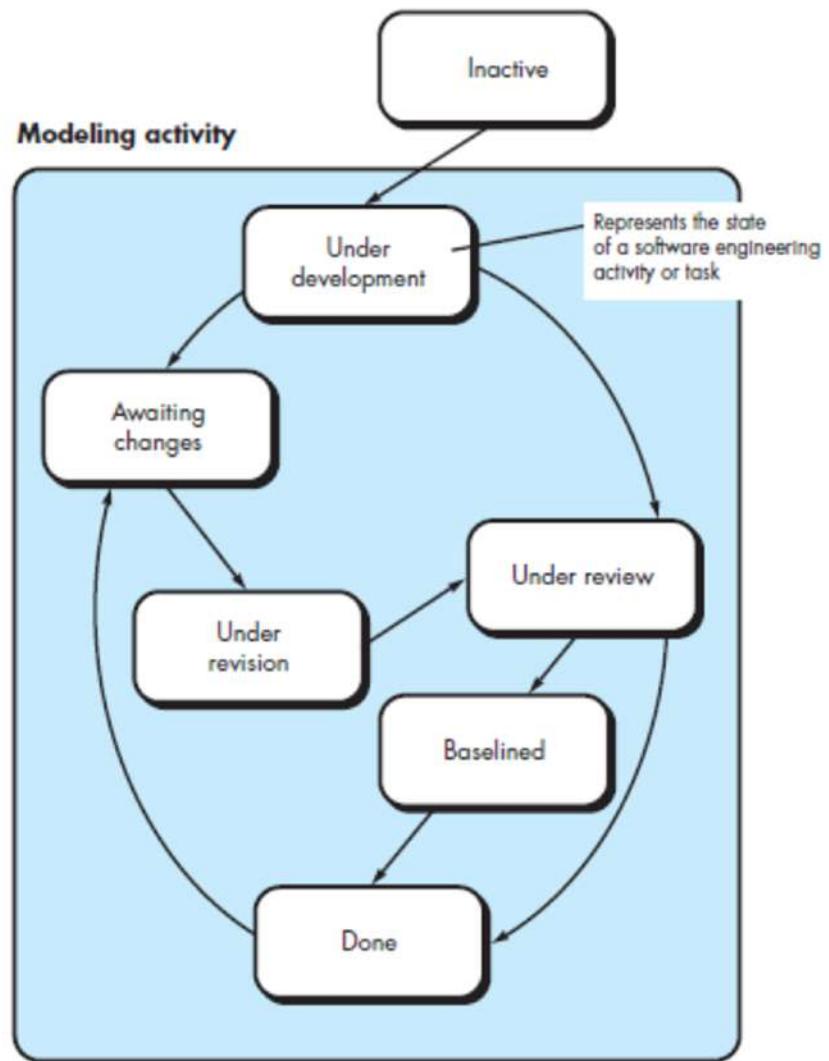
- In essence, the spiral, when characterized in this way, remains operative until the software is retired.
- There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

- The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.
- The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product.

- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

## CONCURRENT MODEL

- The *concurrent development model*, sometimes called *concurrent engineering*, allows a software team to represent iterative and concurrent elements of any of the process models described in this chapter.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.



## CONCURRENT MODEL

- An activity— **modeling** —may be in any one of the states 7 noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., **communication** or **construction** ) can be represented in an analogous manner.
- All software engineering activities exist concurrently but reside in different states.

## CONCURRENT MODEL

- For example, early in a project the communication activity has completed its first iteration and exists in the **awaiting changes** state.
- The modeling activity (which existed in the **none** state while initial communication was completed) now makes a transition into the **under development** state.
- If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.

## CONCURRENT MODEL

- Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.
- For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered.
- This generates the event *analysis model correction*, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.

## **CONCURRENT MODEL:**

- Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project.
- Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network.
- Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks.
- Events generated at one point in the process network trigger transitions among the states associated with each activity.

- Last Lecture:
- What is Software?
- Software Process
- Process Models

# Today's Lecture

- Traditional Approach Vs Agile Approach
- Agile Manifesto
- Agile Methodology

## What is Traditional Process models

- Traditional project management is an established methodology where projects are run in a sequential cycle.
- It follows a fixed sequence: Communication, planning, modelling, construction, and deployment.
- The traditional project management approach puts special emphasis on linear processes, documentation, upfront planning, and prioritization.
- As per the traditional method, time and budget are variable and requirements are fixed due to which it often faces budget and timeline issues.

## What is Traditional Process models

- For every step, there are tools and techniques defined by the standard methodology PMBOK® which are followed by project managers.
- Interestingly, it also includes other methodologies such as PRINCE 2 which is followed by various organizations under UK government and private companies like Vodafone, Siemens and others.
- It is also called the Waterfall model

# Benefits of traditional methodology

- Clearly defined objectives
- Controllable processes
- Clear documentation
- More accountability

# Traditional Software Process Models

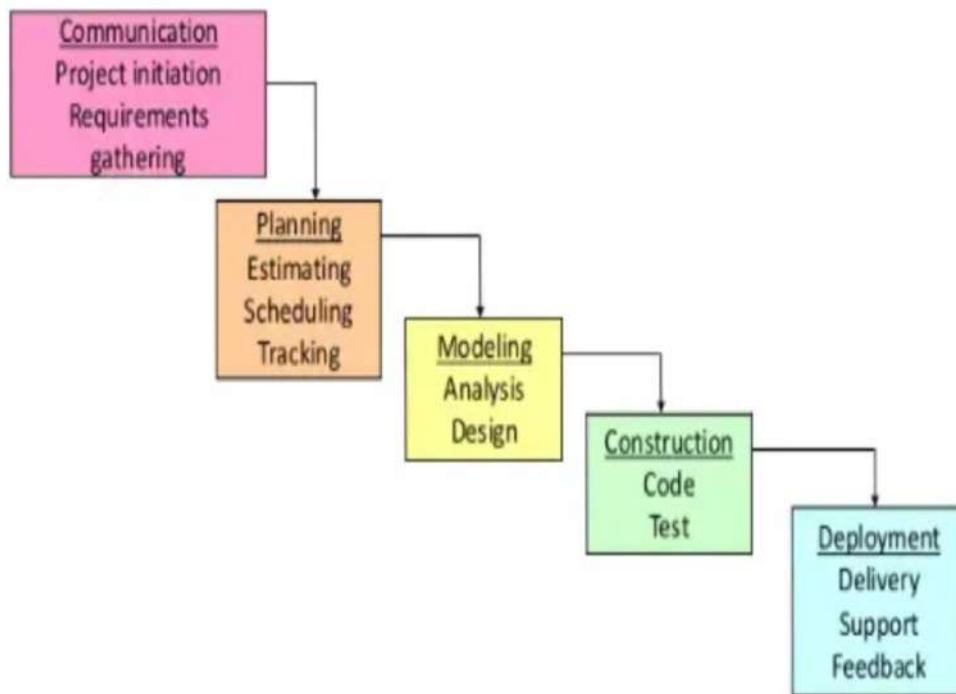
## **Generic Process Framework**

- **Communication**
  - Involves communication among the customer and other stakeholders; encompasses requirements gathering
- **Planning**
  - Establishes a plan for software engineering work; addresses technical tasks, resources, work products, and work schedule
- **Modelling (Analyse, Design)**
  - Encompasses the creation of models to better understand the requirements and the design
- **Construction (Code, Test)**
  - Combines code generation and testing to uncover errors
- **Deployment**
  - Involves delivery of software to the customer for evaluation and feedback

## Process Model

- Defines a distinct set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software
- The activities may be linear, incremental, or evolutionary

# Waterfall Model

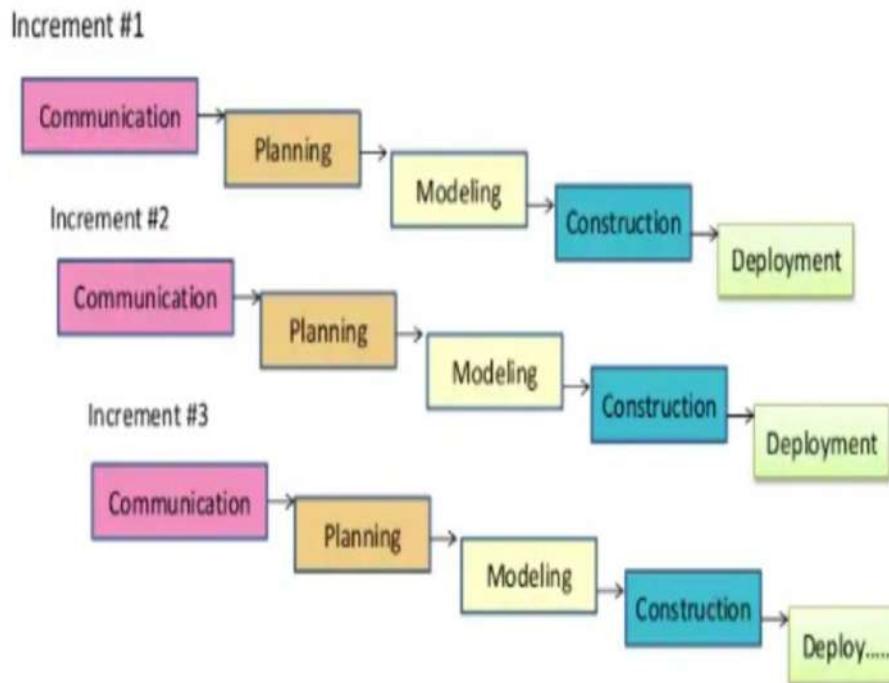


- Oldest software lifecycle model
- Used when requirements are well understood and risk is low
- Work flow is in a linear (i.e., sequential) fashion
- Used often with well-defined adaptations or enhancements to current software

## Waterfall Model- Problems

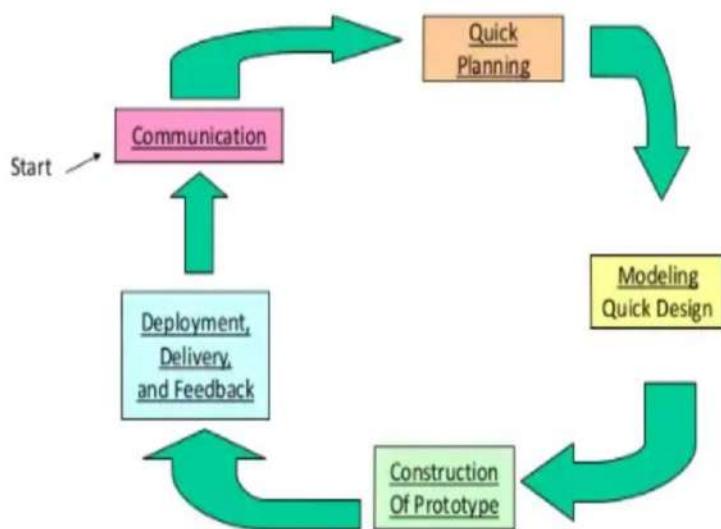
- Doesn't support iteration, so changes can cause confusion
- Difficult for customers to state all requirements explicitly and up front
- Requires customer patience because a working version of the program doesn't occur until the final phase

# Incremental Model



- Used when requirements are well understood
- Multiple independent deliveries are identified
- Work flow is in a linear (i.e., sequential) fashion within an increment and is staggered between increments
- Iterative in nature; focuses on an operational product with each increment
- Provides a needed set of functionality sooner while delivering optional components later
- Useful also when staffing is too short for a full-scale development

# Prototyping Model

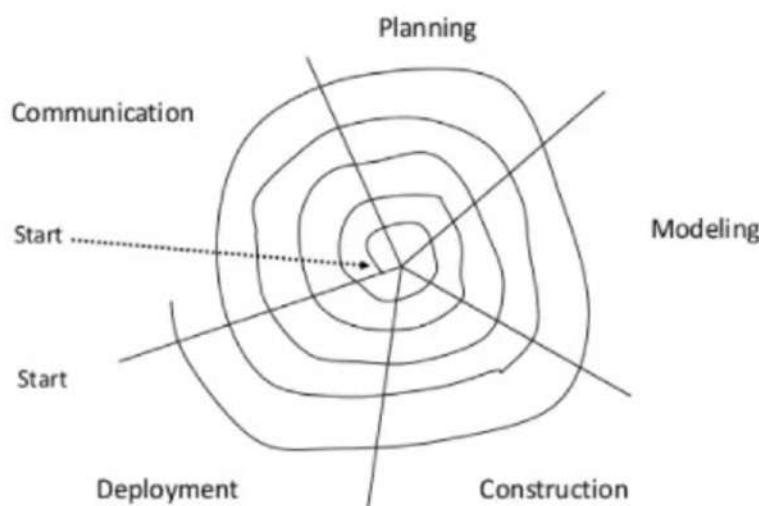


- Follows an evolutionary and iterative approach
- Used when requirements are not well understood
- Serves as a mechanism for identifying software requirements
- Focuses on those aspects of the software that are visible to the customer/user
- Feedback is used to refine the prototype

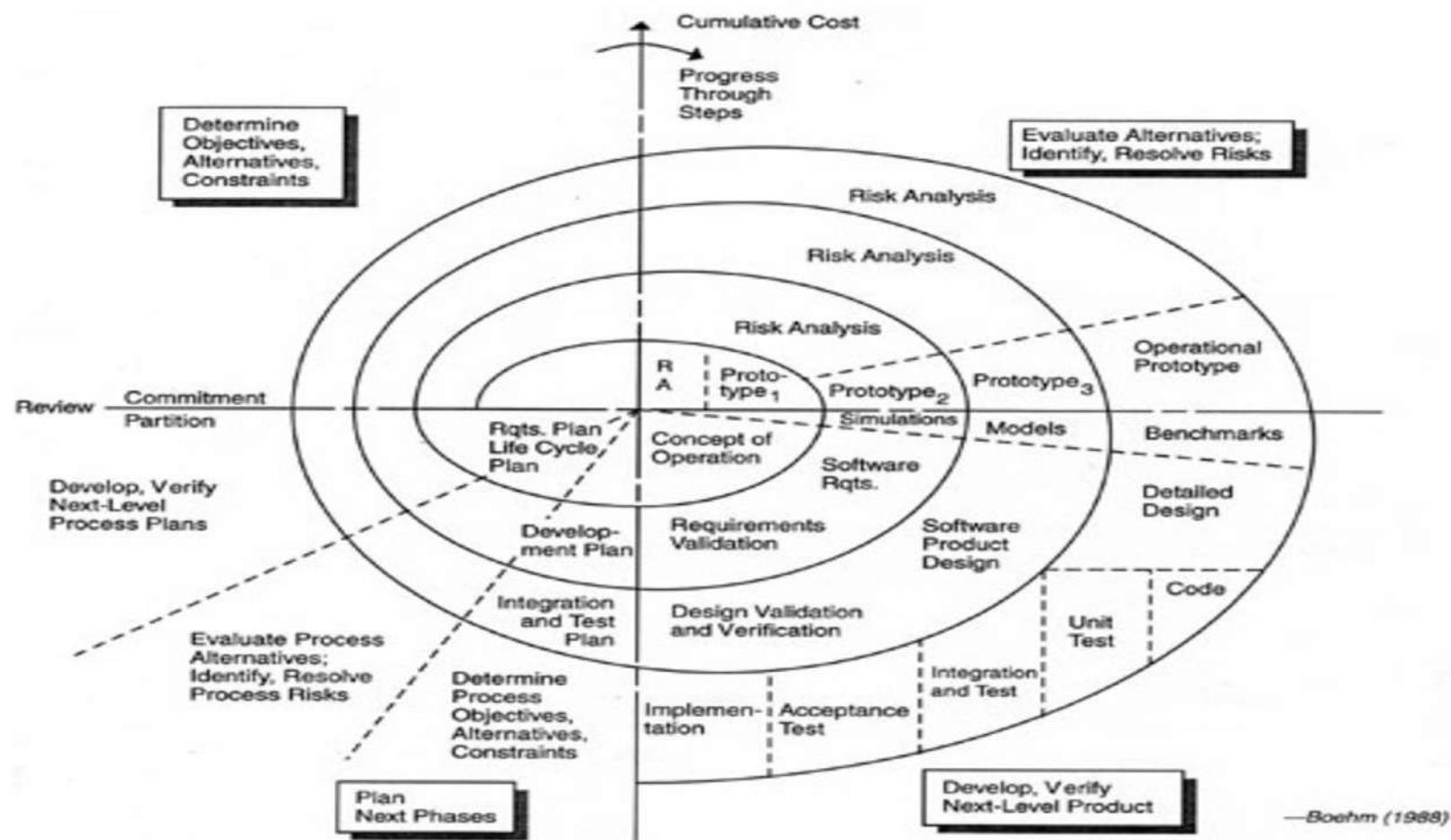
# Prototyping Model- Problems

- The customer sees a "working version" of the software, wants to stop all development and then buy the prototype after a "few fixes" are made
- Developers often make implementation compromises to get the software running quickly (e.g., language choice, user interface, operating system choice, inefficient algorithms)
- Lesson learned
  - Define the rules up front on the final disposition of the prototype before it is built
  - In most circumstances, plan to discard the prototype and engineer the actual production software with a goal toward quality

# Spiral Model



- Follows an evolutionary approach
- Used when requirements are not well understood and risks are high
- Inner spirals focus on identifying software requirements and project risks; may also incorporate prototyping
- Outer spirals take on a classical waterfall approach after requirements have been defined, but permit iterative growth of the software
- Operates as a risk-driven model...a go/no-go decision occurs after each complete spiral in order to react to risk determinations
- Requires considerable expertise in risk assessment
- Serves as a realistic model for large-scale software development



—Boehm (1988)

## General Weakness of Evolutionary Process model

- 1) Prototyping poses a problem to project planning because of the uncertain number of iterations required to construct the product
- 2) Evolutionary software processes do not establish the maximum speed of the evolution
  - If too fast, the process will fall into chaos
  - If too slow, productivity could be affected
- 3) Software processes should focus first on flexibility and extensibility, and second on high quality
  - We should prioritize the speed of the development over zero defects
  - Extending the development in order to reach higher quality could result in late delivery

# What is Agile Process model

- While Agile is a general approach used for software development, it relies heavily on teamwork, collaboration, time boxing tasks, and the flexibility to respond to change as quickly as possible.
- Agile follows an iterative process where projects are divided into sprints of the shorter span.
- Unlike the traditional approach, less time is spent on upfront planning and prioritization as agile is more flexible in terms of changes and developments in the specification.

# Benefits of agile process model

- Flexible prioritization
- Early and predictable delivery
- Predictable costs and schedules
- Improves quality
- More transparency

## Difference between traditional and agile project methodology

Characteristics	Agile approach	Traditional approach
Organizational structure	Iterative	Linear
Scale of projects	Small and medium scale	Large-scale
User requirements	Interactive input	Clearly defined before implementation
Involvement of clients	High	Low
Development model	Evolutionary delivery	Life cycle
Customer involvement	Customers are involved from the time work is being performed	Customers get involved early in the project but not once the execution has started

## Difference between traditional and agile project methodology

Characteristics	Agile approach	Traditional approach
Escalation management	When problems occur, the entire team works together to resolve it	Escalation to managers when problem arise
Model preference	Agile model favors adaption	Traditional model favors anticipation
Product or process	Less focus on formal and directive processes	More serious about processes than the product
Test documentation	Comprehensive test planning	Tests are planned one sprint at a time
Effort estimation	Scrum master facilitates and the team does the estimation	Project manager provides estimates and gets approval from PO for the entire project
Reviews and approvals	Reviews are done after each iteration	Excessive reviews and approvals by leaders

## Why is Agile preferred not traditional PM approach

Many developers and project managers prefer to use the agile methodology for a variety of reasons. Some of them are discussed below:

### **More flexibility**

- When it comes to making changes in the product or a process, agile methodology is much more flexible than the waterfall methodology.
- While working, if team members feel that there is a need to experiment and try something different than as planned, the agile methodology easily allows them to do so.
- The best thing about this methodology is that it focuses more on the product than following a rigid structure.
- Unlike the traditional approach, agile methodology isn't linear or follows a top-down approach. So, any last-minute changes can be accommodated in the process without affecting the end-result and disrupting the project schedule.

## **Transparency**

- In agile methodology, everything is out there and transparent. The clients and decision makers are actively involved from the initiation, planning, review, and in the testing part of a product.
- Whereas in the traditional approach, the project manager is holding reins of the project, thus others don't get to make the major decisions.
- The agile methodology facilitates team members to view the progress right from the start to the end.
- This level of transparency plays a significant role to constitute a healthy work environment.

## ***Ownership and accountability***

- One of the **striking differences** in both project management approaches is the **level of ownership and accountability** that each provides to team members.
- In **traditional project management**, a project manager is the person of the ship which means that the **entire ownership** belongs to him/her. **Customers** are also **involved** during the **planning phase** but their **involvement ends there** and then as soon as the execution starts.
- In the **agile methodology**, every team member **shares ownership of the project**. Each one of them plays an active role to complete the sprint within the estimated time.
- Unlike traditional project management, everyone involved in the project can easily see view the progress from the beginning to an end.

## ***Scope for feedback***

- In the traditional approach, every single process is clearly defined and planned from the beginning of the project. The project has to be completed within the estimated time and budget. So, any big change or feedback that might push the deadline is skipped.
- Whereas agile management allows constant feedback that is helpful in providing better output.
- Due to high acceptance for feedback in agile methodology, it has become the first choice for many project managers and software developers.
- They can respond to customer requests as customers get to validate each iteration that enables them to deliver a high-quality product or service within the delivery time.

## ***Project complexity***

- Owing to its linear approach, the traditional project management methodology is majorly used for small or less complex projects. As discussed earlier, this methodology isn't a fan of sudden changes and avoids them strictly as it would take the team back to square one.
- Agile could be your best bet in terms of managing big and complex projects. Whether your project has multiple interconnected phases or one stage is dependent on many others, choose agile as it is a better fit for complex projects

## How to choose the correct approach

- In reality, there is no ‘one-size-fits-all’ methodology suitable for every project or organization. The choice to implement a methodology largely depends on factors such as the nature of the project, size, resources involved among others.
- Most of the times, smart project managers decide which methodology to adopt during the beginning or initiation of the project. He takes the final call in agreement with other project sponsors and people involved in the project planning process.
- There are some factors you can take into consideration while choosing the right methodology for your project.

- Take a look at the project requirements. Are the requirements clear? If project requirements are unclear or tend to change, choose the agile methodology. And, the traditional methodology fits best to a situation where the requirements are clearly defined and well understood from the first go.
- Consider the technology involved in the project. The traditional project management methodology is more appropriate if no new technology or tools are involved. Agile methodology, being more flexible than the former allows more space for more experimentation with the new technology.

- Is the project prone to unwanted risks and threats? Considering the rigid nature of the traditional methodology, it's not advisable to go this methodology. However, risks can be addressed sooner in the agile approach, it seems like a better option in terms of risk management.
- Another important factor is the availability of resources. The traditional approach works best with big and complex teams and projects. Whereas an agile team usually consists of a limited number of experienced team members.
- The criticality of an end product depends a lot on the nature of the chosen project management methodology. As the traditional method involves a documentation, it is very much suitable for critical products as compared to the agile project management methodology.

# Agile Manifesto

- The Agile Manifesto is a document that identifies four key values and 12 principles that its authors believe software developers should use to guide their work.
- Furthermore, the developers express a desire to find a balance between the existing ways of development and new the alternatives. They admit to accepting modeling and documentation, but only when it has a clear, beneficial use.
- The developers also explain that while planning is important, it is also necessary to accept that plans change and to allow flexibility for these modifications.
- Overall, the Manifesto focuses on valuing individuals and interactions over processes and tools.

# Four values of Agile

The four core values of [Agile software development](#) as stated by the Agile Manifesto are:

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation; and
- responding to change over following a plan.

## Agile Manifesto - 12 Principles



The 12 principles articulated in the Agile Manifesto are:

- Satisfying customers through early and continuous delivery of valuable work.
- Breaking big work down into smaller tasks that can be completed quickly.
- Recognizing that the best work emerges from self-organized teams.
- Providing motivated individuals with the environment and support they need and trusting them to get the job done.
- Creating processes that promote sustainable efforts.
- Maintaining a constant pace for completed work.
- Welcoming changing requirements, even late in a project.
- Assembling the project team and business owners on a daily basis throughout the project.
- Having the team reflect at regular intervals on how to become more effective, then tuning and adjusting behavior accordingly.
- Measuring progress by the amount of completed work.
- Continually seeking excellence.
- Harnessing change for a competitive advantage

# The Agile Manifesto's purpose

- Supporters of Agile methodologies say the four values outlined in the Agile Manifesto promote a software development process that focuses on quality by creating products that meet consumers' needs and expectations.
- The 12 principles are intended to create and support a work environment that is focused on the customer, that aligns to business objectives and that can respond and pivot quickly as user needs and market forces change.

# Agile methodology

Any Agile Methodology that is suitable for any enterprise, can be depending on various factors;

- ***Team types:*** In relevance to the means of Agile processes, an individual is willing to deploy.
- ***Organization size and the condition on which an individual looks to scale agile from lower to top***
- ***Organizational culture:*** To determine whether an organization is ready, or would be interested, for a highly- configurated agile strategy, or looking for more compliant approaches.

# Various Agile Methodologies

- There are various types of agile methodology available in the market to suit every project's wants. Although there are different agile methodologies, everything is based on the main principles in the agile manifesto.
- Therefore, every framework or behavior that adapts these principles is named Agile, and, the agile methodology benefits can be copiously apprehended only with the collaboration of all the involved parties.
- The below agile methodologies list comprises of famous types of agile methodology that one can opt from:

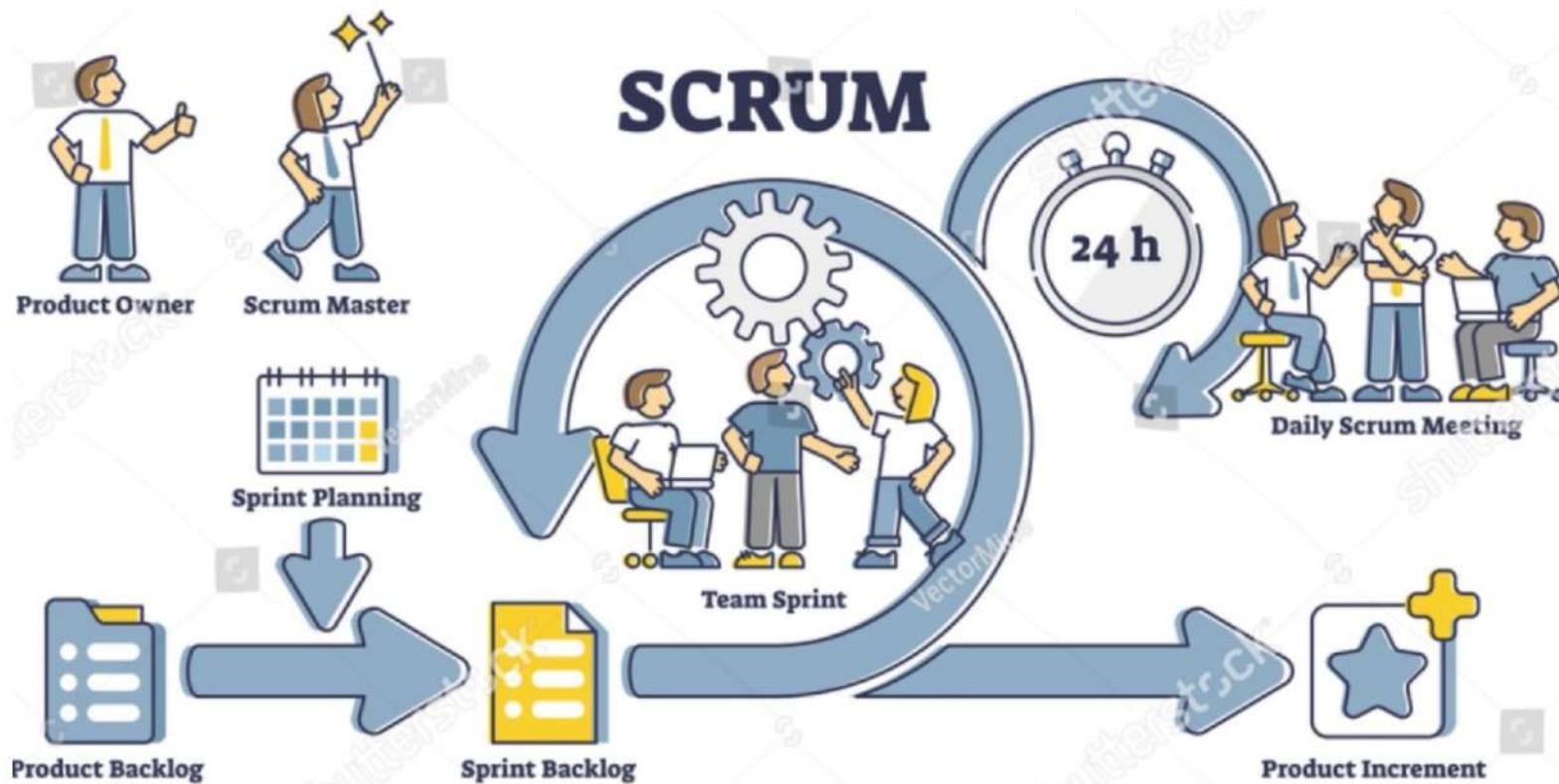
## Agile Scrum Methodology

- Scrum is a lightweight framework of Agile Project Management, it can be adopted to conduct iterative and all types of incremental projects.
- Due to its specific characteristics like simplicity, sustained productivity, and strength for blending several underlying approaches adopted by other agile methods, Scrum has obtained popularity over the years.
- The hands-on system under Scrum includes easy steps and elements, that are the following;

## Agile Scrum Methodology

- **Product owner**, who creates an estimated wish list that is identified as a product backlog.
- **Scrum team**, that takes one little part of the top wish list, termed as **Sprint Backlog** and work out in order to implement it.
- After that scrum team concludes their sprint backlog task in a **Sprint**, i.e., a period of 2-4 weeks. In addition to that, the progress of their work can be accessed through a meeting that is called **Daily Scrum**.
- The **Scrum Master** maintains the team focused toward their targets.
- At the end of a sprint, the task is able to represent or transmit, and team finishes that particular sprint with a review and feedback and initiates with a new one.
- <https://www.youtube.com/watch?v=4Vf4x0RQd74>

# Agile Scrum Methodology



# Agile Scrum Methodology

Advantages:	Disadvantages:
<ul style="list-style-type: none"><li>• There is a lot of motivation in teams, because programmers want to meet the deadline of every sprint;</li><li>• The transparency allows for the project to be followed by all members of a team or even an organisation;</li><li>• The focus on quality is a constant in the scrum method, resulting in fewer mistakes.</li><li>• The dynamics of this method allow developers to reorganise priorities, ensuring that sprints that have not yet been completed get more attention</li></ul>	<ul style="list-style-type: none"><li>• The segmentation of the project and the search for agility of the development can sometimes lead the team to lose track of the project as a whole, focusing only on one part;</li><li>• Each developer's role may not be well defined, resulting in some confusion amongst team members.</li></ul>

## Lean

- It is the iterative, agile methodology that directs the team on addressing customer values by compelling value stream mapping, although, it is a deeply adaptable, emerging methodology with the absence of solid guidelines, laws, or methods.

The fundamental *principles of Lean* including;

- *Uninterrupted advancement,*
- *Respect for other people,*
- *Eradicate waste,*
- *Rapid delivery,*
- *Knowledge-making and*
- *Defer commitment.*
- <https://www.youtube.com/watch?v=xgFBQcQrq-Q>

## Lean

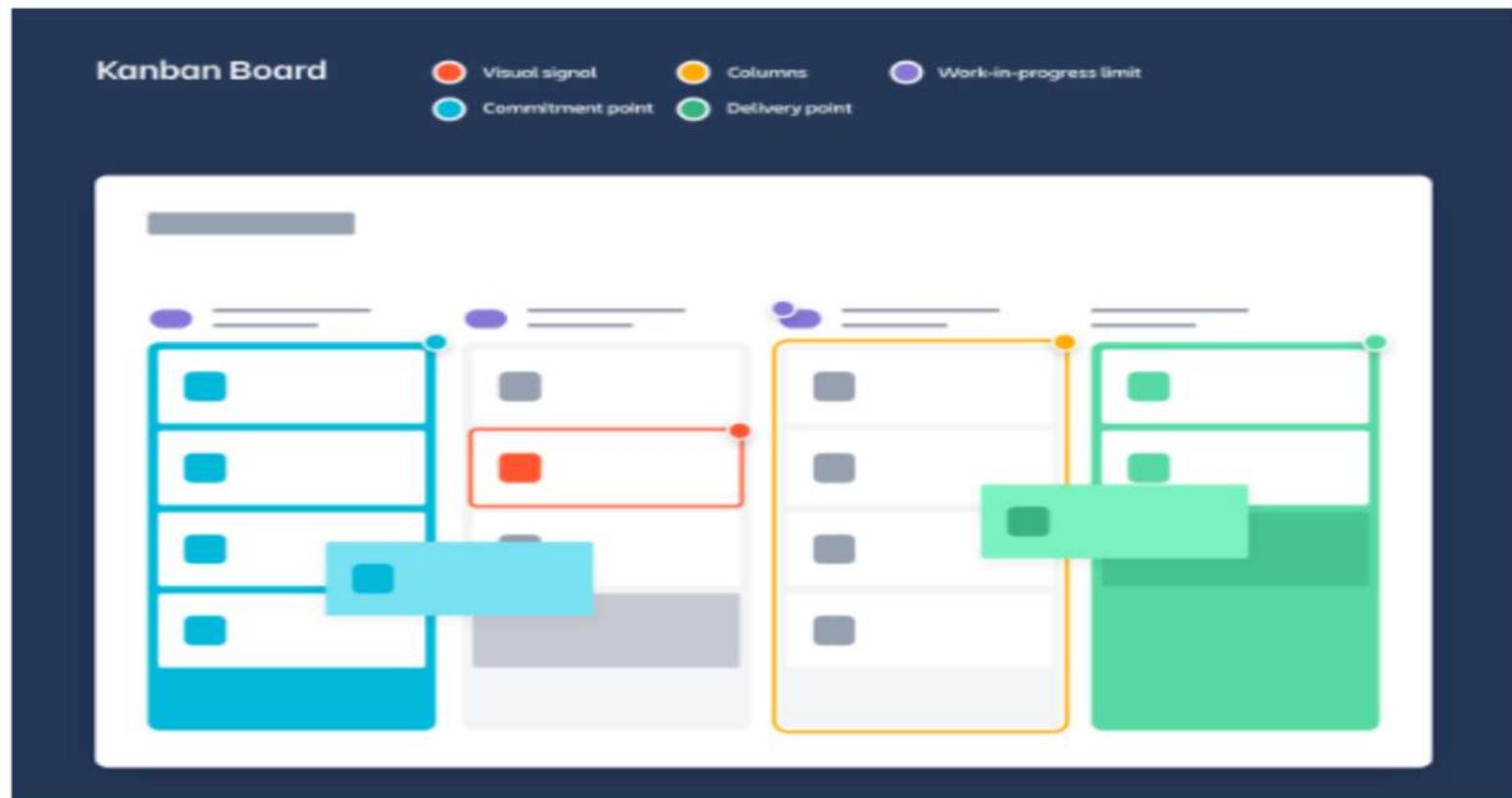
- Lean gives the authority of [decision-making](#) to every individual and small teams as it is considered as the most faster and effective method in comparison to the hierarchical flow of control.
- It focuses on the proficient implementation of team resources and assures to make everyone as productive as possible for maximum time

# Kanban

- Kanban is an eminently visual workflow management approach, famous amidst Lean teams, that can be employed for visualizing and thoroughly maintaining the making of products, it focuses on continual delivery of the product, but is not making stress to the entire software development life cycle.
- Similar to Scrum, Kanban is the process developed for supporting collaborative teamwork more effectively. It works adequately on three principles;
- ***For visualizing what to perform today***, i.e, workflow automation, that specifies all the elements, under the context of each other, could be very informative.
- ***For bounding the quantity of work in progress*** to maintain harmony in the flow-based approach, so that teams can't begin and commit extra work at once.
- ***For boosting flow***, like, when some task is about to complete, the next priority would be item into play from the backlog.
- However, Kanban encourages steadily collaboration and maintains active, continuous learning and enhancement through describing the best plausible team workflow.

## Elements of a Kanban board

Kanban boards can be broken down into five components: Visual signals, columns, work-in-progress limits, a commitment point, and a delivery point.



- **Visual Signals** — One of the first things you'll notice about a kanban board are the visual cards (stickies, tickets, or otherwise).
- Kanban teams write all of their projects and work items onto cards, usually one per card. For agile teams, each card could encapsulate one [user story](#).
- Once on the board, these visual signals help teammates and stakeholders quickly understand what the team is working on.

- **Columns** — Another hallmark of the kanban board are the columns. Each column represents a specific activity that together compose a “workflow”.
- Cards flow through the workflow until completion. [Workflows](#) can be as simple as “To Do,” “In Progress,” “Complete,” or much more complex.

- **Work In Progress (WIP) Limits** — WIP limits are the maximum number of cards that can be in one column at any given time.
- A column with a WIP limit of three, cannot have more than three cards in it.
- When the column is “maxed-out” the team needs to swarm on those cards and move them forward before new cards can move into that stage of the workflow.
- These WIP limits are critical for exposing bottlenecks in the workflow and maximizing flow.
- WIP limits give you an early warning sign that you committed to too much work.

- **Commitment point** — Kanban teams often have a backlog for their board.
- This is where customers and teammates put ideas for projects that the team can pick up when they are ready.
- The commitment point is the moment when an idea is picked up by the team and work starts on the project.

- **Delivery point** — The delivery point is the end of a kanban team's workflow.
- For most teams, the delivery point is when the product or service is in the hands of the customer.
- The team's goal is to take cards from the commitment point to the delivery point as fast as possible.
- The elapsed time between the two is the called Lead Time.
- Kanban teams are continuously improving to decrease their lead time as much as possible.

To Do	In Progress	Done
Try a Kanban tool  Plan a project using Kanban	Learn about Kanban	Get a whiteboard  Find sticky notes



## Kanban vs. scrum board

The difference between kanban and scrum is actually quite subtle. By most interpretations, scrum teams use a kanban board, just with scrum processes, artifacts, and roles along with it. There are, however, some key differences.

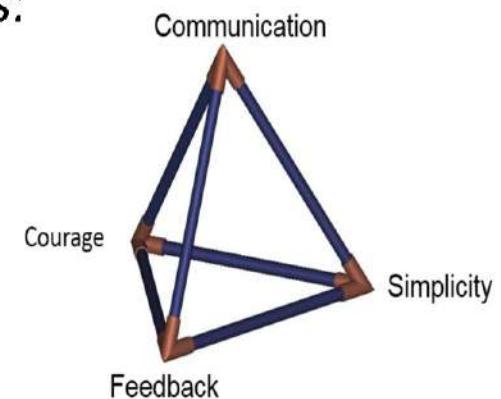
- [Scrum](#) sprints have start and stop dates whereas kanban is an ongoing process.
- Team roles are clearly defined in scrum ([product owner](#), dev team, and [scrum master](#)), while kanban has no formal roles. Both teams are self-organized.
- A kanban board is used throughout the lifecycle of a project whereas a scrum board is cleared and recycled after each sprint.
- A [scrum board](#) has a set number of tasks and strict deadline to complete them.
- Kanban boards are more flexible with regards to tasks and timing. Tasks can be reprioritized, reassigned, or updated as needed.

## Extreme Programming(XP)

- Generally being used with Scrum, it can focus on how Agile can increase customer satisfaction, instead of delivering at the entirety, the customer seeks for the near future, it provides them what they demand at present.
- XP is concentrated on regular propaganda and precise development cycles. In addition to that, it implements code review, pair programming and regular communication with customers.

XP method is basically based on the *four simple values*:

- *Uniformity*,
- *Simplicity*
- *Communication*,
- *Feedback and Endurance*.



# Why do we need XP?

## XP solutions:

### Common problems of software development:

- Schedule slips
- Business misunderstood
- Defect rate
- *Management*
- *Motivation of developers*

- Short iterations, fast delivery
- Whole team
- Test driven development
- *Shared understanding*
- *Humanity and productivity*

# XP features

- XP is the most suitable for:
  - Small and medium size projects
  - New technologies
  - Projects with unclear requirements
  - Risky projects
- XP improves skills by cross training
- No more than 20 developers in a team
- Using of XP in life-critical projects is questionable

# Extreme programming practices 1

---

Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development Tasks.
Small Releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple Design	Enough design is carried out to meet the current requirements and no more.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

---

# Extreme programming practices 2

---

Pair Programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective Ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything.
Continuous Integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of over-time are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site Customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

---

# Pair Programming

- Perhaps the most radical principle of extreme programming is pair programming. Code is developed at the work station by pairs of developers.
- One (the ‘driver’) actually types the code while the other (the ‘navigator’) watches, asks questions, and make suggestions.
- From time to time the pair will change roles.
- In one way this is taking peer review of products a step further.
- Pair programming promotes the production of code that is well structured, easy to understand and relatively free of errors.

## Benefits of Agile Methodology

As defined, Agile is a mindset that directly benefits faster, lighter, and more efficient development processes. The process delivers products and services that customers look for, and the entire product development process is more quicker in response to changes;

- **Faster:** One of the major benefits of Agile methodology is faster/speedy development and response. A quicker software development process significantly reduces times between paying and getting paid and leads to a profitable business.
- **Upgrade customer satisfaction:** There is no requirement for a longer queue to get exactly what customers want with agile development. Infact, a swift set of iterations are done very closely for what they look for, very quickly. The system adjusts rapidly to reshape successful customer solutions and adapt it as it would alter the overall environment of product development.

## Benefits of Agile Methodology

- **Values executives:** Employees are highly valued when providing productive ideas rather than following a fixed set of rules. Agile methodologies enable employees to set their goals and achieve them appropriately. With these methodologies, employees are in the best position to respond to challenges, resolve obstacles and meet the goals and objectives at hand.
- **Eradicate rework:** Implicating more customers into each phase of the requirements and delivery helps in aligning project on-task and in-tune with customer requirements at each step that lead to less backtracking and save time amid development cycle and customer suggested revisions.

## Next Lecture

- Agile Scrum framework -in Detail

- Last Lecture:
- Traditional Approach Vs Agile Approach
- Agile Manifesto
- Agile Methodology

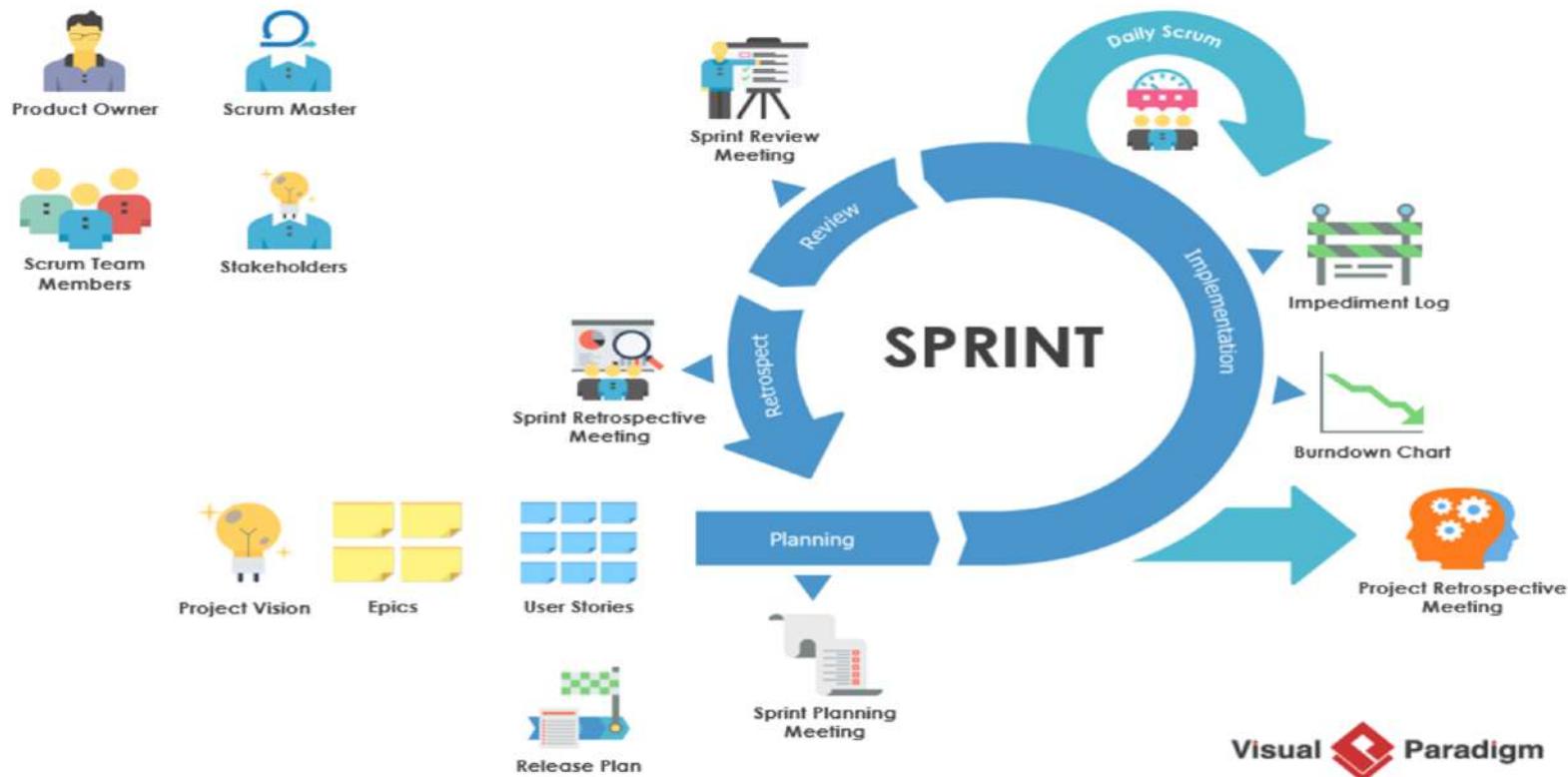
# Today's Lecture

- Agile Scrum Framework in detail

## Scrum –in Detail

- The scrum framework is heuristic; it's based on **continuous learning** and adjustment to fluctuating factors. It acknowledges that the team doesn't know everything at the start of a project and will evolve through experience.
- Scrum is **structured to help teams naturally adapt to changing conditions and user requirements**, with re-prioritization built into the process and short release cycles so your team can constantly learn and improve.
- While **scrum is structured, it is not entirely rigid**. Its execution can be tailored to the needs of any organization. There are many theories about how exactly scrum teams must work in order to be successful.

## The Agile – Scrum Framework



Roles	Artifacts	Ceremonies
<ul style="list-style-type: none"><li>• Product owner</li><li>• Development team</li><li>• Scrum master</li></ul>	<ul style="list-style-type: none"><li>• Increment</li><li>• Product backlog</li><li>• Sprint backlog</li></ul>	<ul style="list-style-type: none"><li>• Sprint planning</li><li>• Sprint review</li><li>• Sprint retrospective</li><li>• Daily scrum</li></ul>

## SCRUM ROLES

---



### PRODUCT OWNER

Represents the client and the business in general for the product on which they're working.



### SCRUM MASTER

Responsible for ensuring the team has everything they need to deliver value.



### DEVELOPMENT TEAM

A group of cross-functional team members all focused on the delivery of working software.

# Three essential roles for scrum success

- A scrum team needs three specific roles:
- **product owner**,
- **scrum master**, and
- **the development team**. [Scrum Team]

And because scrum teams are cross-functional, the development team includes testers, designers, UX specialists, and ops engineers in addition to developers.

# The scrum product owner-Setting clear direction

- Product owners are the champions for their product. They are focused on understanding business, customer, and market requirements, then prioritizing the work to be done by the engineering team accordingly.



# The scrum product owner

Effective product owners:

- Build and manage the product backlog.
- Closely partner with the business and the team to ensure everyone understands the work items in the product backlog.
- Give the team clear guidance on which features to deliver next.
- Decide when to ship the product with a predisposition towards more frequent delivery.
- The product owner is not always the product manager. Product owners focus on ensuring the development team delivers the most value to the business. Also, it's important that the product owner be an individual. No development team wants mixed guidance from multiple product owners.

## The scrum master-Holding it all together

- **Scrum masters are the champions for scrum within their teams.** They coach teams, product owners, and the business on the scrum process, and look for ways to fine-tune their practice of it.
- An effective scrum master deeply understands the work being done by the team and can help the team optimize their transparency and delivery flow.
- As the facilitator-in-chief, he/she schedules the needed resources (both human and logistical) for sprint planning, stand-up, sprint review, and the sprint retrospective.

# The scrum master-Holding it all together



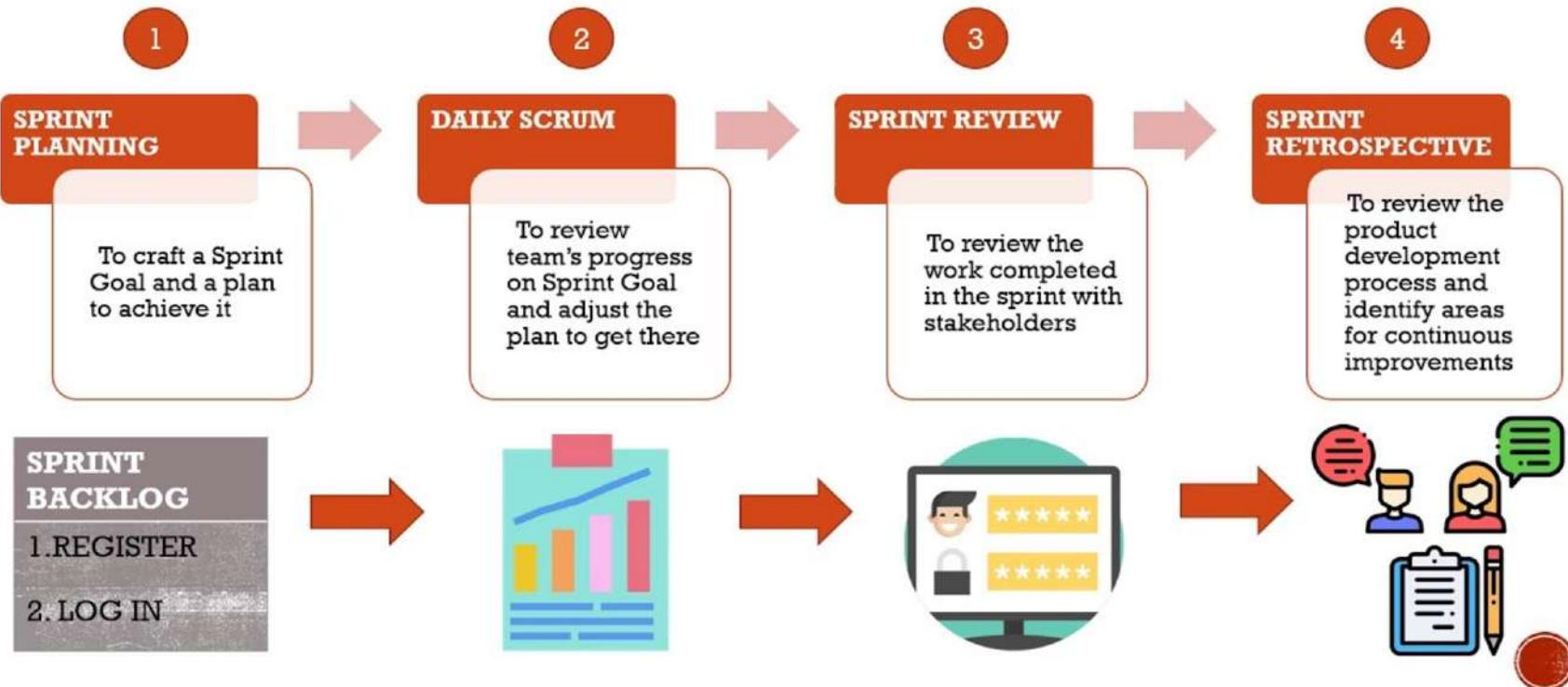
## The scrum development team-Redefining “developer”

- They are **the champions for sustainable development** practices. The most effective scrum teams are tight-knit, co-located, and usually five to seven members.
- Team members have differing skill sets, and cross-train each other so no one person becomes a bottleneck in the delivery of work. Strong scrum teams are self-organising and approach their projects with a clear ‘we’ attitude.
- All members of the team help one another to ensure a successful sprint completion.

- The scrum team drives the plan for each sprint.
- They forecast how much work they believe they can complete over the iteration using their historical velocity as a guide.
- Keeping the iteration length fixed gives the development team important feedback on their estimation and delivery process, which in turn makes their forecasts increasingly accurate over time.



## EVENTS/CEREMONIES IN AGILE METHODOLOGY (SCRUM)



## Scrum ceremonies or events

- Some of the more well-known components of the scrum framework are the set of sequential events, ceremonies, or meetings that scrum teams perform on a regular basis. The ceremonies are where we see the most variations for teams. For example, some teams find doing all of these ceremonies cumbersome and repetitive, while others use them as a necessary check-in.

key events a scrum team might partake in

### **Organize the backlog:**

- Sometimes known as backlog grooming, this event is the responsibility of the product owner.
- The product owner's main jobs are to drive the product towards its product vision and have a constant pulse on the market and the customer.
- Therefore, he/she maintains this list using feedback from users and the development team to help prioritize and keep the list clean and ready to be worked on at any given time

key events a scrum team might partake in

### **Sprint planning:**

- The work to be performed (scope) during the current sprint is planned during this meeting by the entire development team.
- This meeting is led by the scrum master and is where the team decides on the sprint goal.
- Specific use stories are then added to the sprint from the product backlog.
- These stories always align with the goal and are also agreed upon by the scrum team to be feasible to implement during the sprint.

At the end of the planning meeting, every scrum member needs to be clear on what can be delivered in the sprint and how the increment can be delivered.

key events a scrum team might partake in

### **Sprint :**

- A sprint is the actual time period when the scrum team works together to finish an increment.
- Two weeks is a pretty typical length for a sprint, though some teams find a week to be easier to scope or a month to be easier to deliver a valuable increment.
- Dave West, from Scrum.org advises that the “more complex the work and the more unknowns, the shorter the sprint should be. But it’s really up to your team, and you shouldn’t be afraid to change it if it’s not working! ”
- During this period, the scope can be re-negotiated between the product owner and the development team if necessary. This forms the crux of the empirical nature of scrum.

# key events a scrum team might partake in

## **Daily scrum or stand up:**

This is a daily super-short meeting that happens at the same time (usually mornings) and place to keep it simple.

Many teams try to complete the meeting in 15 minutes, but that's just a guideline. This meeting is also called a '**daily stand-up**' emphasizing that it needs to be a quick one.

The goal of the daily scrum is for everyone on the team to be on the same page, aligned with the sprint goal, and to get a plan out for the next 24 hours.

**The stand up is the time to voice any concerns you have with meeting the sprint goal or any blockers.**

A common way to conduct a stand up is for every team member to answers three questions in the context of achieving the sprint goal:

- What did I do yesterday?
- What do I plan to do today?
- Are there any obstacles?

## key events a scrum team might partake in

- **Sprint review:**

- At the end of the sprint, the team gets together for an informal session to view a demo of, or inspect, the increment.
- The development team showcases the backlog items that are now ‘Done’ to stakeholders and teammates for feedback.
- The product owner can decide whether or not to release the increment, although in most cases the increment is released.

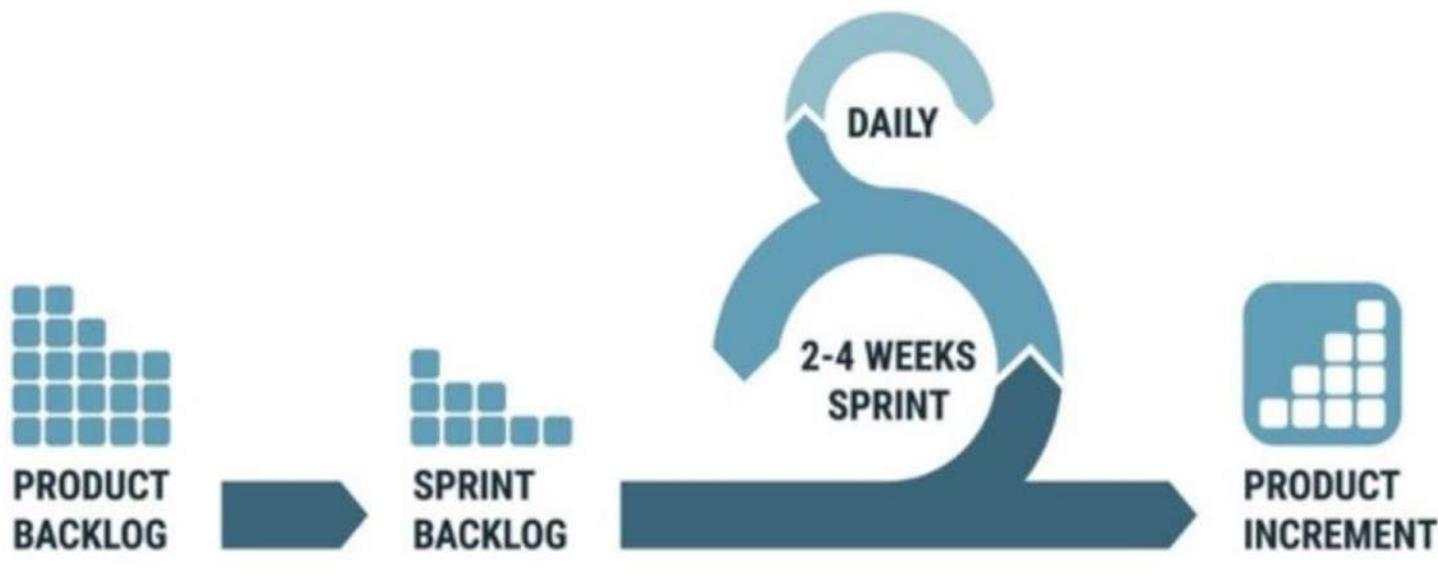
This review meeting is also when the product owner reworks the product backlog based on the current sprint, which can feed into the next sprint planning session.

- For a one-month sprint, consider time-boxing your sprint review to a maximum of four hours.

key events a scrum team might partake in

- **Sprint retrospective:**
- The [retrospective](#) is where the team comes together to document and discuss what worked and what didn't work in a sprint, a project, people or relationships, tools, or even for certain ceremonies.
- The idea is to create a place where the team can focus on what went well and what needs to be improved for the next time, and less about what went wrong.

# SCRUM



Artifacts in Scrum Framework - agile

# Agile scrum artifacts

- Agile scrum artifacts are information that a scrum team and stakeholders use to detail the product being developed, actions to produce it, and the actions performed during the project.
- These artifacts provide metadata points that give insight into the performance of a sprint.
- They are essential tools for every scrum team since they enable core scrum attributes of transparency, inspection, and adaption.

- Artifacts are created during the main activities of a scrum [sprint](#):
- Plan work and future goals
- Create tasks to achieve these goals
- Organize tasks into sprints based on dependency and priority
- Execute the tasks
- Review and analyze results in order to compare to the goals
- Repeat these steps

# The main artifacts of agile scrum



# Product backlog

- The [product backlog](#) is a list of new features, enhancements, bug fixes, tasks, or work requirements needed to build a product. It's compiled from input sources like customer support, competitor analysis, market demands, and general business analysis.

# Sprint backlog

- The sprint backlog is a set of product backlog tasks that have been promoted to be developed during the next product increment.
- Sprint backlogs are created by the development teams to plan deliverables for future increments and detail the work required to create the increment.
- Sprint backlogs are created by selecting a task from the product backlog and breaking that task into smaller, actionable sprint items.

- Consider an example task like “build a shopping cart page”, which requires many design and development subtasks.
- The product backlog is home to the primary task while the supporting tasks like “create a shopping cart visual design mockup” or “program the shopping cart sessions” are housed in the sprint backlog.

- The sprint backlog is updated during the [sprint planning](#) phase of scrum.
- The smaller sprint tasks are assigned to the relevant teams like design and development.
- If a team does not have the capacity to deliver all the sprint tasks, the remaining sprint tasks will standby in the sprint backlog for a future sprint.

## Product increment

- A product increment is the customer deliverables that were produced by completing product backlog tasks during a sprint.
- It also includes the increments of all previous sprints. There is always one increment for each sprint and an increment is decided during the scrum planning phase.
- An increment happens whether the team decides to release to the customer.
- Product increments are incredibly useful and complementary to CI/CD in version tracking and, if needed, version rollback.

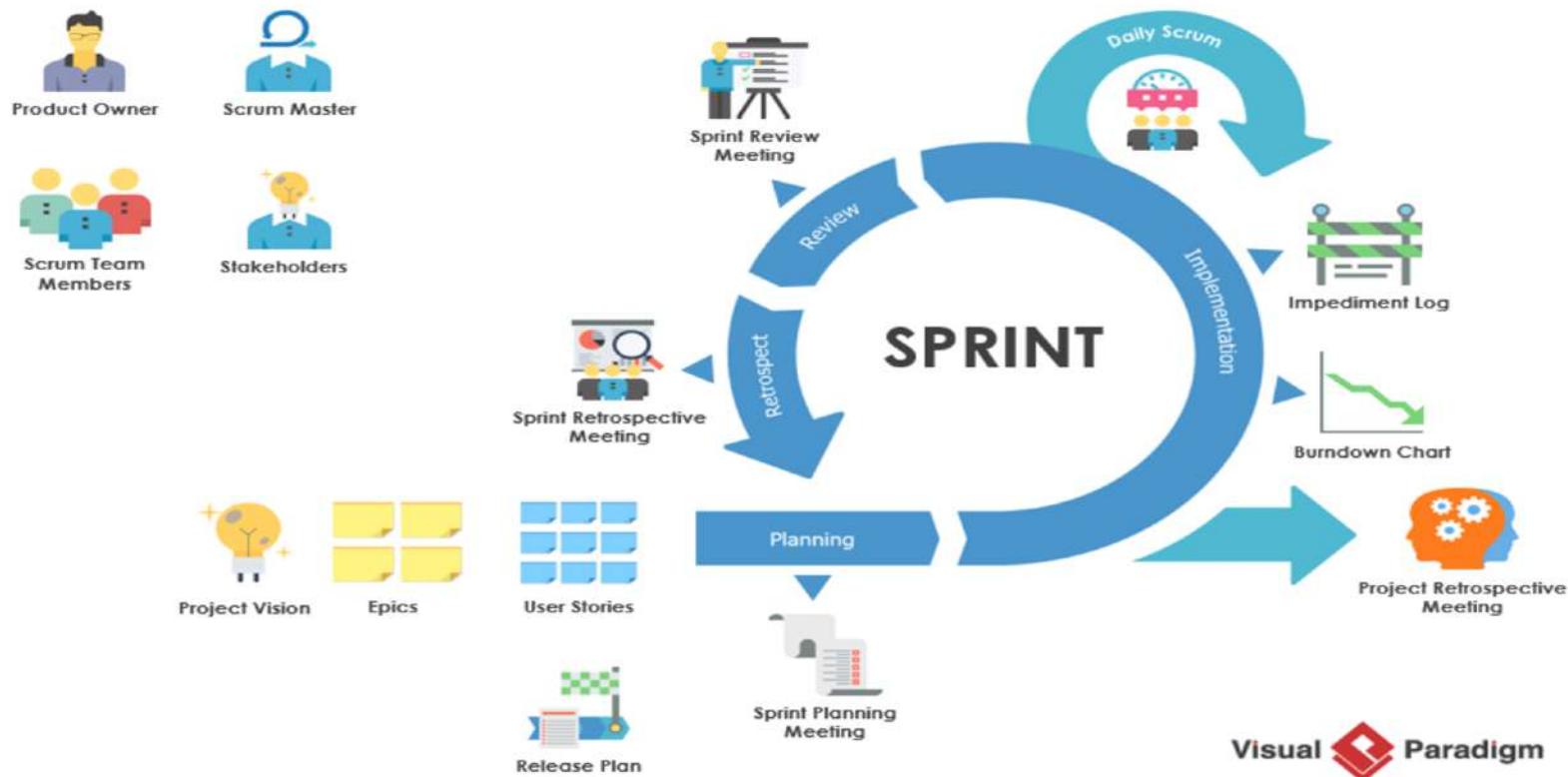
# Artifact transparency

- Scrum artifacts are powerful aids that help teams operate more efficiently. Therefore, it's important all teams have access and visibility into the artifacts.
- Product owners and scrum masters need to make it a regular practice to review and discuss artifacts with development teams.
- This will help teams stay aware of operational inefficiencies and produce creative ways to improve velocity.

# Last Lecture

- **Scrum:**
- Scrum process,
- Scrum roles - Product Owner, ScrumMaster, Team, Project Manager, product manager,
- Scrum Events,
- Scrum artifacts

## The Agile – Scrum Framework



Roles	Artifacts	Ceremonies
<ul style="list-style-type: none"><li>• Product owner</li><li>• Development team</li><li>• Scrum master</li></ul>	<ul style="list-style-type: none"><li>• Increment</li><li>• Product backlog</li><li>• Sprint backlog</li></ul>	<ul style="list-style-type: none"><li>• Sprint planning</li><li>• Sprint review</li><li>• Sprint retrospective</li><li>• Daily scrum</li></ul>

## SCRUM ROLES

---



### PRODUCT OWNER

Represents the client and the business in general for the product on which they're working.



### SCRUM MASTER

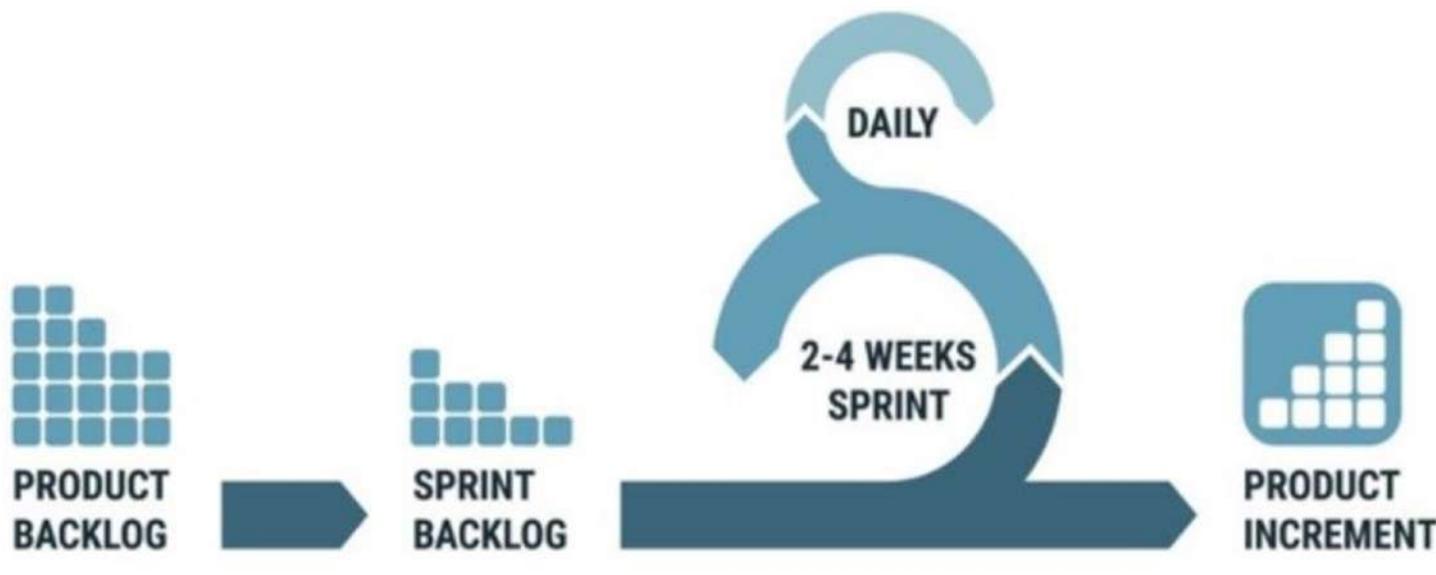
Responsible for ensuring the team has everything they need to deliver value.



### DEVELOPMENT TEAM

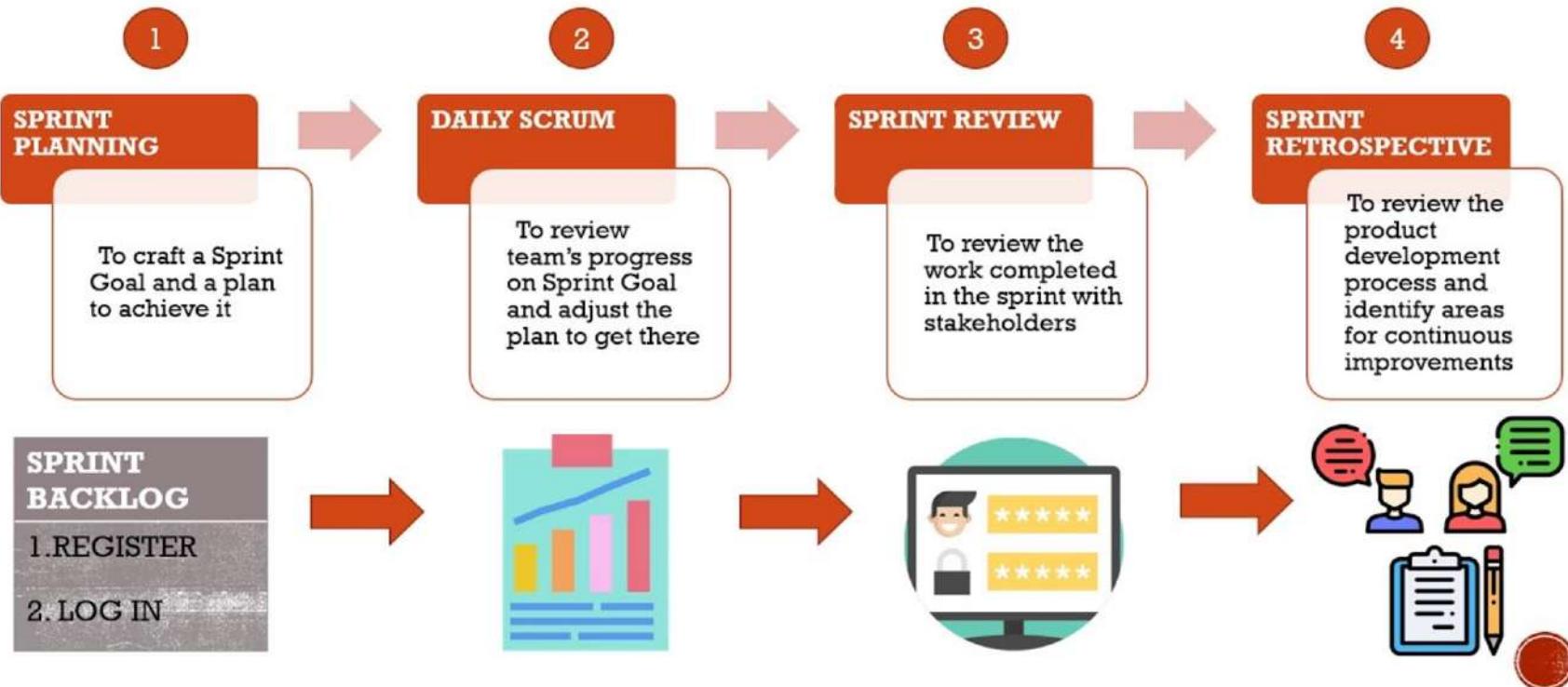
A group of cross-functional team members all focused on the delivery of working software.

# SCRUM



Artifacts in Scrum Framework - agile

## EVENTS/CEREMONIES IN AGILE METHODOLOGY (SCRUM)



# Today's Lecture

- **Product Inception:**
- Product vision,
- stakeholders,
- initial backlog creation

# Product Inception(starting point)

- Every product should begin with a problem that needs to be solved.
- Product Inception is the process by which we pin down the problem in concrete terms and determine how we are going to solve it—all before writing a single line of code.
- Representatives from the design and development team as well as stakeholders must be present throughout the process in order for Product Inception to be effective.
- By the end of a Product Inception session, everyone should be on the same page about where we're going, why we're going there, and how we are going to arrive at our destination.

## Why do we believe in Product Inception?

- Product Inception is crucial to developing a product that truly solves the problem we intended to solve and helps users meet their goals.
- Even if a problem seems obvious, there are always multiple perspectives to consider; there may be aspects of a problem we can't see until a stakeholder brings it to our attention.
- In addition, everyone may have a different idea about what an effective solution looks like.

## Why do we believe in Product Inception?

- Product Inception, then, brings everyone to the table so we can come up with a unified vision for the product and agree on a road map to take us there.
- The collaborative nature of Product Inception ensures that we include everyone's point of view, thus allowing us to produce ideas that are even better than what each of us could have thought of individually.

# How It Works

- Product Inception involves a series of exercises that helps us define why we are building the product, how to best fulfill users' needs, what features the product will include, and which parts of the product are high priorities.

The different phases of Product Inception that we'll explore in this series include:

- Defining the main problem and solution with a [\*\*Product Vision Board\*\*](#). This exercise enables us to articulate what exactly we want to achieve with this product.

## How It Works

- Ensuring the product is relevant and identifying which product attributes to prioritize through **Stakeholder Analysis**.
- Understanding users' goals and related needs through **Persona Analysis**.
- Visualizing the functionality of the product, identifying necessary features, and planning the first release with a **Story Map**.

# The Product Vision Board: The First Step to Discovering a Successful Software Product

- A concrete product vision is crucial for any software product. It tells you where you are going, why you are going there, and who you are going there for.
- A well-defined product vision is the foundation to a successful development process, while an unclear product vision will often leave your development team confused, making it difficult to prioritize user stories and features.
- In the end, without product vision, your users may not even understand what your product does.

# What is Product Vision?

- The product vision is a short statement that encapsulates what you are trying to achieve with a product.
- It sounds simple, but in reality it can be quite challenging.
- Say for example, we want to create a calendar application for finding meeting times. That sounds great, but that's not a clear product vision.
- Product vision isn't actually about the product at all, it's about what the product will do and what problem it will solve. A much better product vision would be:

Take the stress out of  
finding a meeting time  
for a large group.

- This tells us exactly what we want to achieve, which will keep us focused on track throughout the development process.

# Product Vision Board

## The Product Vision Board

**SOPHiLABS**



Vision  
What is your purpose for creating the product?  
What positive change should it bring about?



Target Group

Which market or market segment does the product address?  
Who are the target customers and users?



Needs

What problem does the product solve?  
What benefit does it provide?



Product

What product is it?  
What makes it stand out?  
Is it feasible to develop the product?



Business Goals

How is the product going to benefit the company?  
What are the business goals?

# The Product Vision Board

**SOPHiLABS**



Vision

What is your purpose for creating the product?  
What positive change should it bring about?

Take the stress out of finding a meeting time for a large group.

Target Group	Needs	Product	Business Goals
 <b>Our Company</b>  Employees who plan meetings  Employees who attend meetings	 <b>Easy to use</b>  <b>Simple</b>  Enable employees to quickly find a meeting time that will work for all participants	  Web and mobile app that automatically generates time that everyone invited to a meeting is free.  Sync with major calendar applications  Easy to use, does work for you	  Decrease time spent planning meetings and emailing back and forth about schedules  Increase Productivity  Boost employee morale  Maximize the number of people in important meetings
Which market or market segment does the product address? Who are the target customers and users?	What problem does the product solve? What benefit does it provide?	What product is it? What makes it stand out? Is it feasible to develop the product?	How is the product going to benefit the company? What are the business goals?

Completed Product Vision Board, Product Vision Board example

# **Stakeholder Analysis: Another Key Piece of Product Inception**

- Stakeholders are a vital part of any software project, which makes Stakeholder Analysis a critical part of the Product Inception phase.
- The only way to make sure that we are building a product that is relevant and valuable is to engage people and understand why they may be interested in the product and bring them into the process.
- If we don't involve stakeholders from the very beginning, we can easily build an excellent product with great features, but it's possible that no one will want or need to use it.

# What is a stakeholder?

- A stakeholder is any person or group of people that have a specific interest in the product we are going to build.
- This will of course include end users and customers but can also include, for example, a leadership team, marketing department, investors, or any number of other people who have an interest or "stake" in the product and its developments.
- We've already begun the process of identifying stakeholders on the Product Vision Board, when we listed Target Groups who can also be considered stakeholders.

- Say for example, we want to develop a flower delivery app where users can log in and order flowers from a local florist to be delivered anywhere in the city. Our stakeholders might include:
- **Customers/End Users:** These are the people who will sign up and use your app to order flowers.
- **Flower Shops:** These are the people who you will need to convince to affiliate with your app and be vendors.
- **Marketing Staff:** They have to market the product and make it desirable for users
- **Investors:** They have a financial interest in the success of the product.

# What is Stakeholder Analysis?

- Stakeholder Analysis is a process of understanding your stakeholders' background and context in order to pinpoint what they want and contrast that with what they actually need.
- Once we've identified our stakeholders, we want to ask questions like:
  - Who are the stakeholders?
  - Where do they come from?
  - What is their common background?
  - What are they looking for, as a group, from the project/product?
  - What is important to them?
  - Do they have any specific goals as a group?

## What is Stakeholder Analysis?

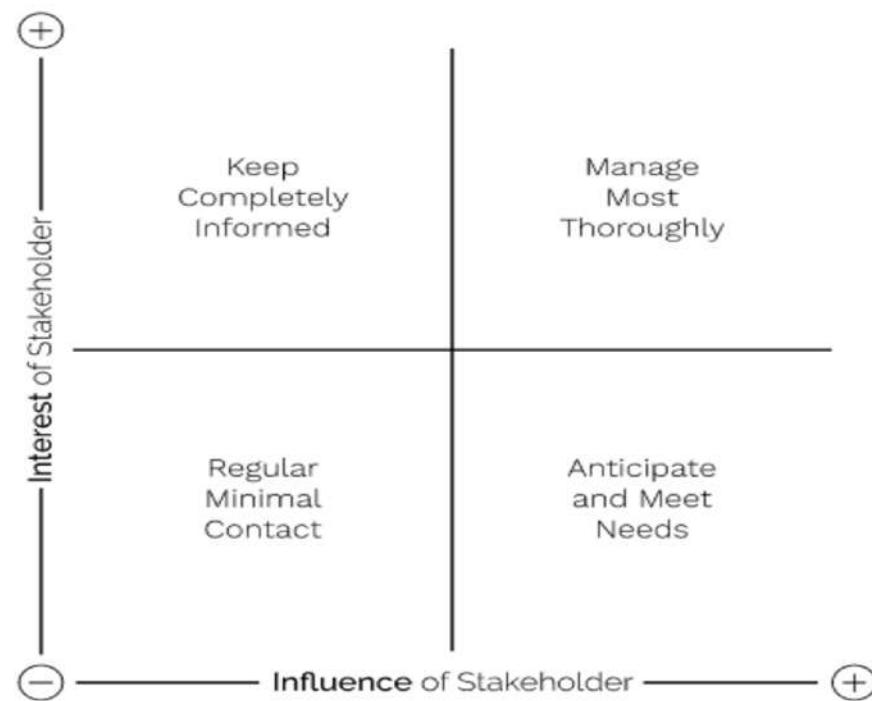
- During Stakeholder Analysis, we need to determine not only who our stakeholders are but also what is their relationship to the project as well as figure out which stakeholders' concerns we will need to prioritize throughout the project.

# Stakeholder Analysis Techniques

- Product Vision Board
- During the process of creating a [product vision board](#), we identified broad target groups, who are ultimately stakeholders
- Stakeholder Mapping: Influence vs. Interest Chart
- Every software development project will have multiple stakeholders and at some point, their needs are going to be in conflict.
- Stakeholder Mapping on an Influence vs. Interest Chart help us to determine each stakeholder's relationship to the product and how we will need to engage them throughout the process.

# Stakeholder Mapping: Influence vs. Interest Chart

Stakeholder Interest vs. Influence Analysis



## Flower App Delivery example revisited

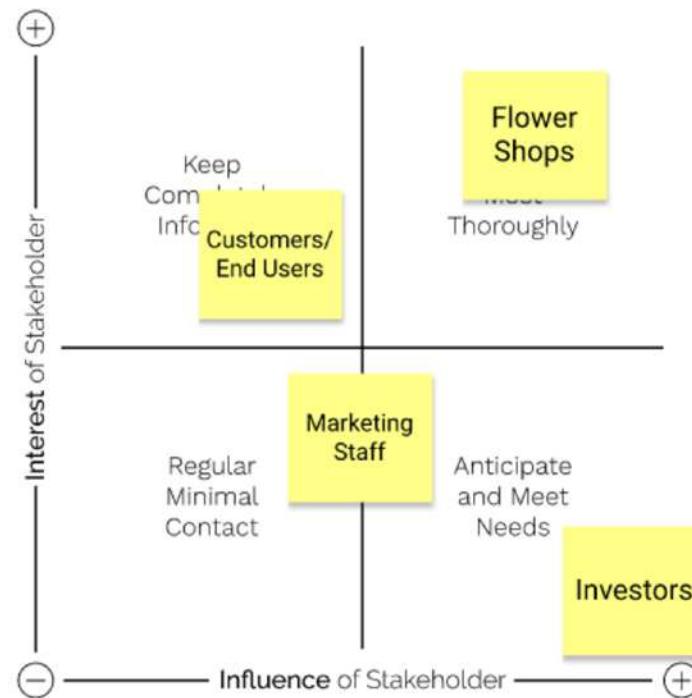
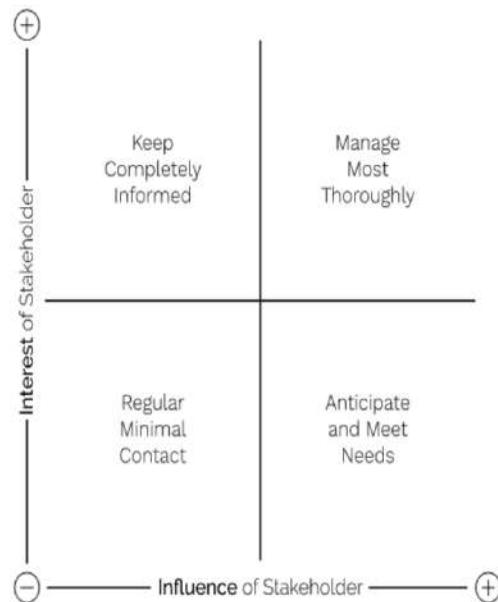
- The chart maps **Influence on the x-axis and Interest on the y-axis**.
- The more Influence a stakeholder has, the more necessary it will be to meet their needs.
- And the more Interest that they have, the more we will need to keep them informed about the product.
- When mapping the stakeholders, we need to keep the product at the center at all times.

## Flower App Delivery example revisited

- Our investors will have a lot of influence over the product, because they are the ones financing it. However, their primary interest in this case is in the revenue received from a successful product, rather than the product itself. For this reason, they are High Influence, Low Interest.
- In contrast, our flower shops have a lot of influence as well, because their buy-in is necessary for the app's success. However, they also have a lot of interest in the product, as it is a tool for them to increase their sales and improve their business.
- This makes them a relatively High Influence, High Interest stakeholder that we will have to manage thoroughly.

In our example, we might map our stakeholders in this way:

Stakeholder Interest vs. Influence Analysis



## Initial backlog creation

- Product Backlog is a dynamic prioritized source for all the work that needs to be done to deliver a product. This artifact is product focused not system focused.
- A product backlog consists of:
  - Features
  - Enhancements
  - Defects
  - Non IT Deliverables

- Example:
- As the mail room I must scan mortgage applications from the customer so that they can be processed
- As the system I must load scanned images so they are available to the indexing user
- As the indexer I want to be able to view a list of all applications requiring indexing
- What information do we capture in the product backlog?

- The details below provide a guideline to what information you may want to capture in a product backlog

Item	Description
Reference	Unique identifier for quick reference during collaboration
Description	A high level description of the item
Type	Feature, New Feature, Enhancement, Defect
Group	This can also be known as a feature set but is simply a grouping of functionality for example: Customer Management Contact Management
Relative Value	A factor that identifies the business value (Scale of 1 to 10, 10 been the highest business value)
Initial Estimate	This can be in story point or ideal days
Adjustment Factor	A summation of factors that may increase the initial estimate
Actual Estimate	Initial Estimate X (1 + Adjustment Factor)
Release	The release number that the PBI will be included in
Sprint (Iteration)	The sprint that the PBI has been allocated to
Notes	Additional comments, references or points of interest, be careful not to get into too much detail, some people prefer this to be separate
Status	Done or Not Done
Committed	Has this item been committed by the product owner i.e it has been decided that it will be incorporated into the product

# Example

Ref	Description	Type	Group	Relative Value	Initial Estimate	Adjust Factor	Estimate	Release	Sprint	Status
01	As the mail room I must scan mortgage applications from the customer so that they can be processed	Feature	Scanning and Imaging	10	3	0	3	1	1	Not Done

# Next Session

- **Agile Requirements –**
- User personas,
- story mapping,
- user stories,
- 3Cs, INVEST,
- Acceptance criteria, sprints, requirements,
- product backlog and backlog grooming;

# Last Lecture

- **Product Inception:**
- Product vision,
- stakeholders,
- initial backlog creation

The different phases of Product Inception that we'll explore in this series include:

- Defining the main problem and solution with a [\*\*Product Vision Board\*\*](#). This exercise enables us to articulate what exactly we want to achieve with this product
- Ensuring the product is relevant and identifying which product attributes to prioritize through [\*\*Stakeholder Analysis\*\*](#).
- [\*\*Initial Product Backlog Creation\*\*](#).

# Today's Session

- **Agile Requirements –**
- User personas,
- story mapping,
- user stories,
- 3Cs, INVEST,
- Acceptance criteria, sprints, requirements,
- product backlog and backlog grooming;

## How It Works

- Ensuring the product is relevant and identifying which product attributes to prioritize through **Stakeholder Analysis**.
- Understanding users' goals and related needs through **Persona Analysis**.
- Visualizing the functionality of the product, identifying necessary features, and planning the first release with a **Story Map**.

- A good requirement should tell each audience member exactly what the expected functionality is.

Good Requirements:

- User Stories
- User Acceptance Tests
- Workflow
- Requirements (Details)
- Wireframes

# User Stories

## What is a User Story?

- A User Story is a requirement expressed from the perspective of an end-user goal. User Stories may also be referred to as Epics, Themes or features but all follow the same format.
- A User Story is really just a well-expressed requirement. The User Story format has become the most popular way of expressing requirements in Agile for a number of reasons:
- It focuses on the viewpoint of a role who will use or be impacted by the solution
- It defines the requirement in language that has meaning for that role
- It helps to clarify the true reason for the requirement
- It helps to define high level requirements without necessarily going into low level detail too early
- User goals are identified and the business value of each requirement is immediately considered within the user story.

# User Stories

User Stories are often deemed to comprise three elements - **the 3C's**

- Card
- Conversation
- Confirmation

# User Acceptance Tests

- These should include all scenarios outlined in the user stories. These should not be too detailed (they don't need to mention specific screens or a complete list of actions to execute the steps). These should read:
- GIVEN that condition 1 and condition 2....  
WHEN I do step 1, and step 2...  
THEN, desired result 1, desired result 2....
- These define a set of actual scenarios a tester could walk through to assert that the feature is complete. These are not detailed test scripts that you find in UAT. They are meant to convey a set of tests that all involved can walk through to understand how the feature will work.

# User Stories

- This states all of the scenarios of the users involved. These should read:
- As a **SOME ROLE**,  
I want to **DO SOMETHING**,  
So that I **CAN GET SOME BENEFIT**
- The user stories are critical to lay out exactly who is going to do what, and for what reason(s).

- The format of the User Story is as follows:

**As a <role>**

**I need**

**So that**

These two examples demonstrate User Stories at different levels, but using the same format:

- At a project level

**As a Marketing Director,**

**I need to improve customer service**

**So that** we retain our customers.

- At a detailed level

**As an Investor,**

**I need to see a summary of my investment accounts,**

**So that** I can decide where to focus my attention.

- User Stories provide another powerful message.
- Choosing User Stories to define requirements demonstrates an intention to work collaboratively with the users to discover what they really need.
- The User Story is brief and intended to be a placeholder for a more detailed discussion later – the Conversation.
- Much of the detail of User Stories emerges during Timeboxes as part of evolutionary development.
- High-level User Stories (Epics) are broken down by the Solution Development Team into more detailed User Stories just before development commences on that group of stories.
- Even then, the User Stories are not intended to be full specifications of the requirements.
- Fine detail may not need to be written down at all, but may simply be incorporated directly into the solution as part of the work within a Timebox.

The user story format helps to ensure that each requirement is captured in a feature-oriented, value oriented way, rather than a solution oriented way.

# User Story – the Card

- User Stories are often printed onto physical cards, for planning purposes and to help the Solution Development Team monitor progress.

## **The Front of the Card**

- On the front of the card, the following information is typically displayed:
  - A unique “Story Identifier”, usually a number or reference
  - A clear, explicit, short name or title
  - “As a I need , so that ”; this section states:
    - who is the primary stakeholder (the role that derives business benefit from the story)
    - what effect the stakeholder wants the story to have
    - what business value the stakeholder will derive from this effect.

# User Story – the Card

- **The Back of the Card**
- On the back, the Confirmation area contains:
  - Acceptance criteria (the test criteria)
  - These acceptance criteria define, at a high level, the test criteria which will confirm that this user story is working as required. These are not intended to be the full test scripts, but will be used to expand into the appropriate test scenarios and test scripts during Timeboxes, as necessary.

For User Stories at the highest level (sometimes called a project Epic), the acceptance criteria may be used to define the aims of the project using criteria that may be measured after the project has completed (as part of the Benefits Assessment).

- Project acceptance criteria example:
- Is customer retention improved by 20% within two years?
- Is product range increased by 10% within 5 years?
- Has speed of dispatch improved to within 24 hours of time of order for 99% of in-stock items within 6 months?

# User Story Example:

- **Story Identifier:** STK001

**Story Name:** Customer Order

**Description:** As a Customer, I need to place an order so that I can have food delivered to my house.

**Confirmation:** Acceptance Criteria examples:

Functional:

- Can I save my order and come back to it later?
- Can I change my order before I pay for it?
- Can I see a running total of the cost of what I have chosen so far?

- Non-functional: availability:

- Can I place an order at any time (24 hours per day or 24/7/365)?
- Can I view the order at any time (24 hours per day or 24/7/365) up to and including delivery?

- Non-functional: security:

- Are unauthorised persons and other customers prevented from viewing my order?

# Well constructed User Stories

Bill Wake's INVEST model provides guidance on creating effective User Stories

Independent	Stories should be as independent as possible from other stories, to allow them to be moved around with minimal impact and potentially to be implemented independently. If stories are tightly dependent, consider combining them into a single user story.
Negotiable	Stories are not a contract. They are "placeholders" for features which the team will discuss and clarify near to the time of development.
Valuable	Stories should represent features providing clear business value to the user/owner of the solution and should be written in appropriate language. They should be features, not tasks.
Estimable	Stories need to be clear enough to estimate (for the appropriate timeframe), without being too detailed.
Small	Stories should be small enough to be estimated. Larger "Epic" stories should be broken down into smaller User Stories as the project progresses. The stories after splitting still follow the INVEST criteria.
Testable	Stories need to be worded clearly and specifically enough to be testable.

- A well-written user story is clear, concise and complete. Some simple checks are:
- It does not combine with, overlap nor conflict with other User Stories
- It conforms to organisational and project standards and policies where applicable
- It is traceable back to the business needs expressed in the business case and project objectives
- Where several User Stories relate to the same feature, but for different users, they are cross-referenced to each other

# What is backlog grooming?

- Backlog grooming refers to the practice of refining the backlog by selecting the important work items, prioritizing them to the top of the backlog and cutting out the unimportant stories and tasks.
- This practice is important because it helps product owners keep the backlog stable, while making it easy to pick and choose which items to work on when the team is ready to start a sprint.

## Who should groom the backlog?

- While the sprint planning meeting is an official scrum meeting, there is no compulsion to have a separate meeting to groom the backlog.
- The backlog is usually groomed by the person responsible for owning and maintaining the backlog.
- The product owner is generally the best person to groom the backlog as he/she has the best understanding of the product that is being built.
- This makes sure that the right items are trimmed out and the most important ones are moved up the backlog.

## Who should groom the backlog?

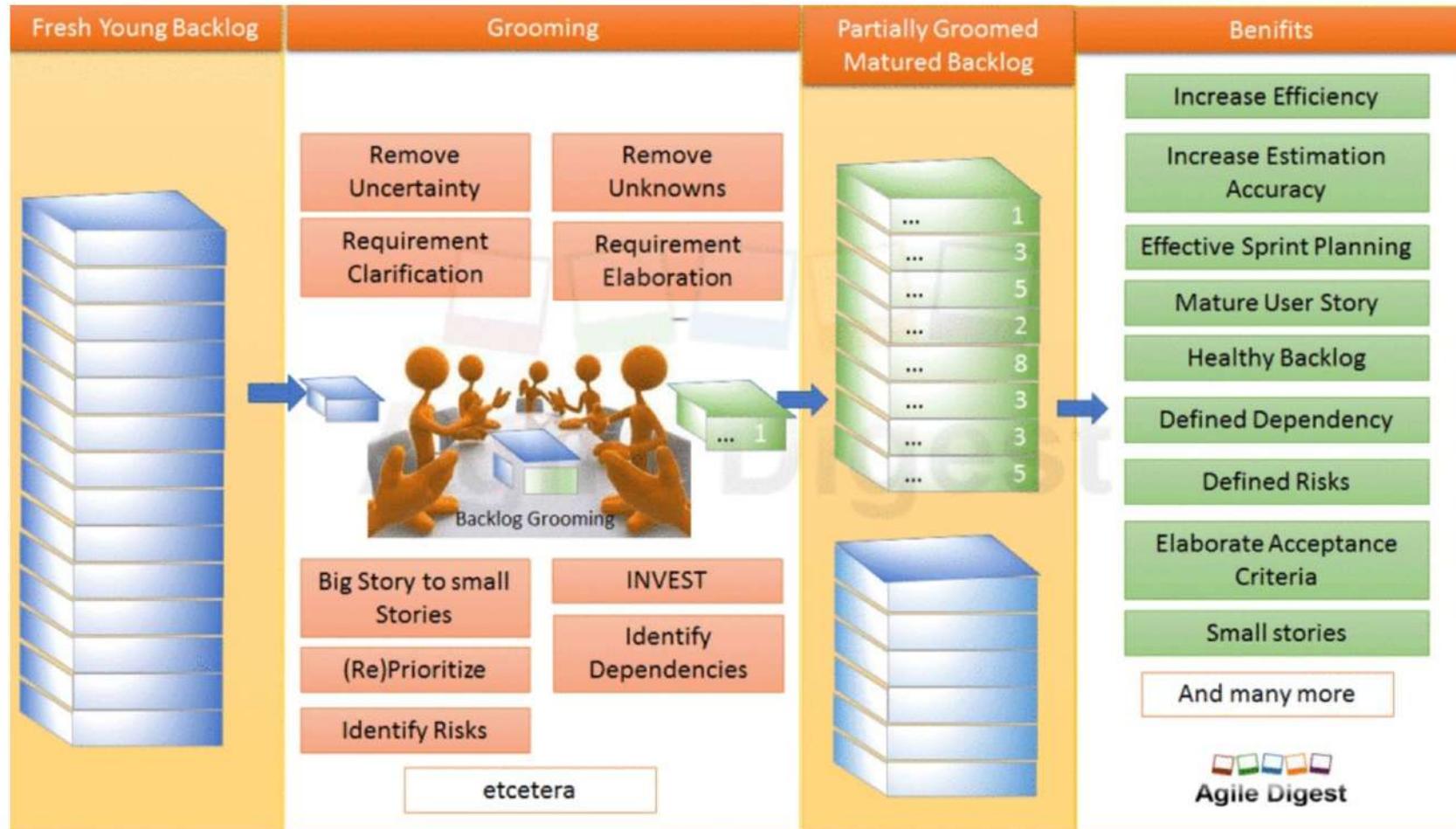
- In addition to the product owner, an engineering leader or manager could be present for a backlog grooming meeting as they add a much needed technical perspective towards estimating and prioritizing items or cutting items from the backlog.
- In this type of grooming session, the product owner acts as a facilitator of the grooming meeting.

# Back log Grooming



# Importance of Backlog Grooming





# The backlog grooming process

- In the course of the development process, a sprint backlog grows ever bigger. Backlog grooming keeps it organized. Before a sprint planning meeting, product owners should review the backlog to ensure:
- The set priorities are correct.
- The prioritized items have all the correct information.
- The feedback from previous meetings has been incorporated.

Sometimes, The backlog can quickly become overcrowded and out of control. To keep it relevant, grooming refines the backlog by cutting out unimportant stories and tasks and maintaining only priority ones.

## Key backlog grooming activities include:

- Eliminating unwanted user stories that don't fit the current product direction.
- Reprioritizing stories to move lower priority items to the bottom of the backlog.
- Breaking down large work items into smaller ones.
- Updating estimates.
- Adding new work items.

The benefits of frequent backlog grooming quickly become apparent as development teams can readily access all the information they need to put together a [sprint action plan](#).

# Benefits of backlog grooming

There are several important reasons to adopt backlog grooming:

- **Keeps the backlog clean**
- **Keeps the backlog relevant**
- **Keeps the whole product team up to date**
- **Increases work velocity**

# Backlog grooming best practices

## Make your product backlog **DEEP**

Roman Pichler, author of “Agile Product Management with Scrum,” used the acronym DEEP to sum up the essential traits of a good product backlog:

- **Detailed Appropriately** — User stories and other items in the product backlog that will be done soon need to be sufficiently well understood by cross-functional teams. Items and initiatives that will not be delivered for a while should be described with less detail.
- **Estimated** — Backlog items at the top should include an accurate estimation of the work needed to deliver them. Conversely, items down the backlog should only be roughly estimated as they are not that well understood yet.

# Backlog grooming best practices

Make your product backlog **DEEP**

- **Emergent** — A product backlog changes over time. As you capture new user insights, the backlog will be changed to adapt to customer needs.
- **Prioritized** — The product backlog should be ordered with the most valuable items at the top and the least valuable at the bottom. Every single backlog item is ranked in relation to its business value and alignment with the strategic goal of the company.

## Sprint planning

- The purpose of sprint planning is to agree on a goal for the next sprint and the set of backlog items (aka [sprint backlog](#)) to achieve it. This involves more than just communicating requirements.
- It's about **learning, considering options and making decisions together.**
- Sprint planning consists of two main components: (1) prioritizing backlog items and (2) agreeing on the number of backlog items in the sprint based on team capacity.

## Sprint planning

- Traditionally, (1) prioritizing backlog items is the responsibility of the [product owner](#), whereas (2) deciding how much to pull in the sprint is the development team's decision. Both should be a discussion with the entire team, while the responsibilities (and the final say) stay with the respective roles.
- The input for a sprint planning session is the product backlog. In order to make the meeting as effective as possible, the top of the backlog — the most important backlog items that should be tackled next — should be well “groomed” or rather “refined” ahead of time.

# Last Lecture

- **Product Inception:**
- Product vision,
- stakeholders,
- initial backlog creation

# Today's Session

- **Agile Requirements –**
- User personas,
- story mapping,
- user stories,
- 3Cs, INVEST,
- Acceptance criteria, sprints, requirements,
- product backlog and backlog grooming;

The different phases of Product Inception that we'll explore in this series include:

- Defining the main problem and solution with a [\*\*Product Vision Board\*\*](#). This exercise enables us to articulate what exactly we want to achieve with this product
- Ensuring the product is relevant and identifying which product attributes to prioritize through [\*\*Stakeholder Analysis\*\*](#).
- [\*\*Initial Product Backlog Creation\*\*](#).

## How It Works

- Ensuring the product is relevant and identifying which product attributes to prioritize through **Stakeholder Analysis**.
- Understanding users' goals and related needs through **Persona Analysis**.
- Visualizing the functionality of the product, identifying necessary features, and planning the first release with a **Story Map**.

- A good requirement should tell each audience member exactly what the expected functionality is.

Good Requirements:

- User Stories
- User Acceptance Tests
- Workflow
- Requirements (Details)
- Wireframes

# User Stories

## What is a User Story?

- A User Story is a requirement expressed from the perspective of an end-user goal. User Stories may also be referred to as Epics, Themes or features but all follow the same format.
- A User Story is really just a well-expressed requirement. The User Story format has become the most popular way of expressing requirements in Agile for a number of reasons:
- It focuses on the viewpoint of a role who will use or be impacted by the solution
- It defines the requirement in language that has meaning for that role
- It helps to clarify the true reason for the requirement
- It helps to define high level requirements without necessarily going into low level detail too early
- User goals are identified and the business value of each requirement is immediately considered within the user story.

# User Stories

User Stories are often deemed to comprise three elements - **the 3C's**

- Card
- Conversation
- Confirmation

# User Acceptance Tests

- These should include all scenarios outlined in the user stories. These should not be too detailed (they don't need to mention specific screens or a complete list of actions to execute the steps). These should read:
- GIVEN that condition 1 and condition 2....  
WHEN I do step 1, and step 2...  
THEN, desired result 1, desired result 2....
- These define a set of actual scenarios a tester could walk through to assert that the feature is complete. These are not detailed test scripts that you find in UAT. They are meant to convey a set of tests that all involved can walk through to understand how the feature will work.

# User Stories

- This states all of the scenarios of the users involved. These should read:
- As a **SOME ROLE**,  
I want to **DO SOMETHING**,  
So that I **CAN GET SOME BENEFIT**
- The user stories are critical to lay out exactly who is going to do what, and for what reason(s).

- The format of the User Story is as follows:

**As a <role>**

**I need**

**So that**

These two examples demonstrate User Stories at different levels, but using the same format:

- At a project level

**As a Marketing Director,**

**I need to improve customer service**

**So that** we retain our customers.

- At a detailed level

**As an Investor,**

**I need to see a summary of my investment accounts,**

**So that** I can decide where to focus my attention.

- User Stories provide another powerful message.
- Choosing User Stories to define requirements demonstrates an intention to work collaboratively with the users to discover what they really need.
- The User Story is brief and intended to be a placeholder for a more detailed discussion later – the Conversation.
- Much of the detail of User Stories emerges during Timeboxes as part of evolutionary development.
- High-level User Stories (Epics) are broken down by the Solution Development Team into more detailed User Stories just before development commences on that group of stories.
- Even then, the User Stories are not intended to be full specifications of the requirements.
- Fine detail may not need to be written down at all, but may simply be incorporated directly into the solution as part of the work within a Timebox.

The user story format helps to ensure that each requirement is captured in a feature-oriented, value oriented way, rather than a solution oriented way.

# User Story – the Card

- User Stories are often printed onto physical cards, for planning purposes and to help the Solution Development Team monitor progress.

## **The Front of the Card**

- On the front of the card, the following information is typically displayed:
  - A unique “Story Identifier”, usually a number or reference
  - A clear, explicit, short name or title
  - “As a I need , so that ”; this section states:
    - who is the primary stakeholder (the role that derives business benefit from the story)
    - what effect the stakeholder wants the story to have
    - what business value the stakeholder will derive from this effect.

# User Story – the Card

- **The Back of the Card**
- On the back, the Confirmation area contains:
  - Acceptance criteria (the test criteria)
  - These acceptance criteria define, at a high level, the test criteria which will confirm that this user story is working as required. These are not intended to be the full test scripts, but will be used to expand into the appropriate test scenarios and test scripts during Timeboxes, as necessary.

For User Stories at the highest level (sometimes called a project Epic), the acceptance criteria may be used to define the aims of the project using criteria that may be measured after the project has completed (as part of the Benefits Assessment).

- Project acceptance criteria example:
- Is customer retention improved by 20% within two years?
- Is product range increased by 10% within 5 years?
- Has speed of dispatch improved to within 24 hours of time of order for 99% of in-stock items within 6 months?

# User Story Example:

- **Story Identifier:** STK001

**Story Name:** Customer Order

**Description:** As a Customer, I need to place an order so that I can have food delivered to my house.

**Confirmation:** Acceptance Criteria examples:

Functional:

- Can I save my order and come back to it later?
- Can I change my order before I pay for it?
- Can I see a running total of the cost of what I have chosen so far?

- Non-functional: availability:

- Can I place an order at any time (24 hours per day or 24/7/365)?
- Can I view the order at any time (24 hours per day or 24/7/365) up to and including delivery?

- Non-functional: security:

- Are unauthorised persons and other customers prevented from viewing my order?

# Well constructed User Stories

Bill Wake's INVEST model provides guidance on creating effective User Stories

Independent	Stories should be as independent as possible from other stories, to allow them to be moved around with minimal impact and potentially to be implemented independently. If stories are tightly dependent, consider combining them into a single user story.
Negotiable	Stories are not a contract. They are "placeholders" for features which the team will discuss and clarify near to the time of development.
Valuable	Stories should represent features providing clear business value to the user/owner of the solution and should be written in appropriate language. They should be features, not tasks.
Estimable	Stories need to be clear enough to estimate (for the appropriate timeframe), without being too detailed.
Small	Stories should be small enough to be estimated. Larger "Epic" stories should be broken down into smaller User Stories as the project progresses. The stories after splitting still follow the INVEST criteria.
Testable	Stories need to be worded clearly and specifically enough to be testable.

- A well-written user story is clear, concise and complete. Some simple checks are:
- It does not combine with, overlap nor conflict with other User Stories
- It conforms to organisational and project standards and policies where applicable
- It is traceable back to the business needs expressed in the business case and project objectives
- Where several User Stories relate to the same feature, but for different users, they are cross-referenced to each other

# What is backlog grooming?

- Backlog grooming refers to the practice of refining the backlog by selecting the important work items, prioritizing them to the top of the backlog and cutting out the unimportant stories and tasks.
- This practice is important because it helps product owners keep the backlog stable, while making it easy to pick and choose which items to work on when the team is ready to start a sprint.

## Who should groom the backlog?

- While the sprint planning meeting is an official scrum meeting, there is no compulsion to have a separate meeting to groom the backlog.
- The backlog is usually groomed by the person responsible for owning and maintaining the backlog.
- The product owner is generally the best person to groom the backlog as he/she has the best understanding of the product that is being built.
- This makes sure that the right items are trimmed out and the most important ones are moved up the backlog.

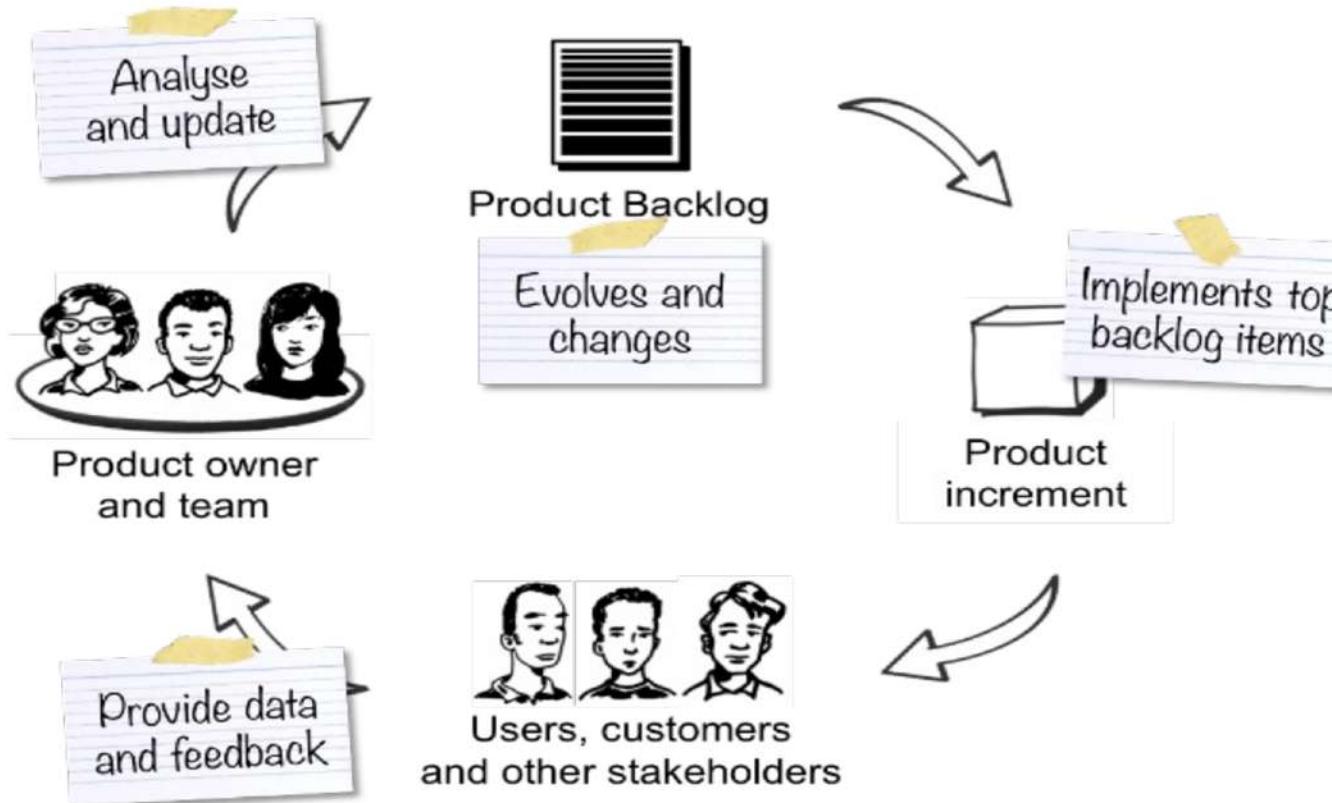
## Who should groom the backlog?

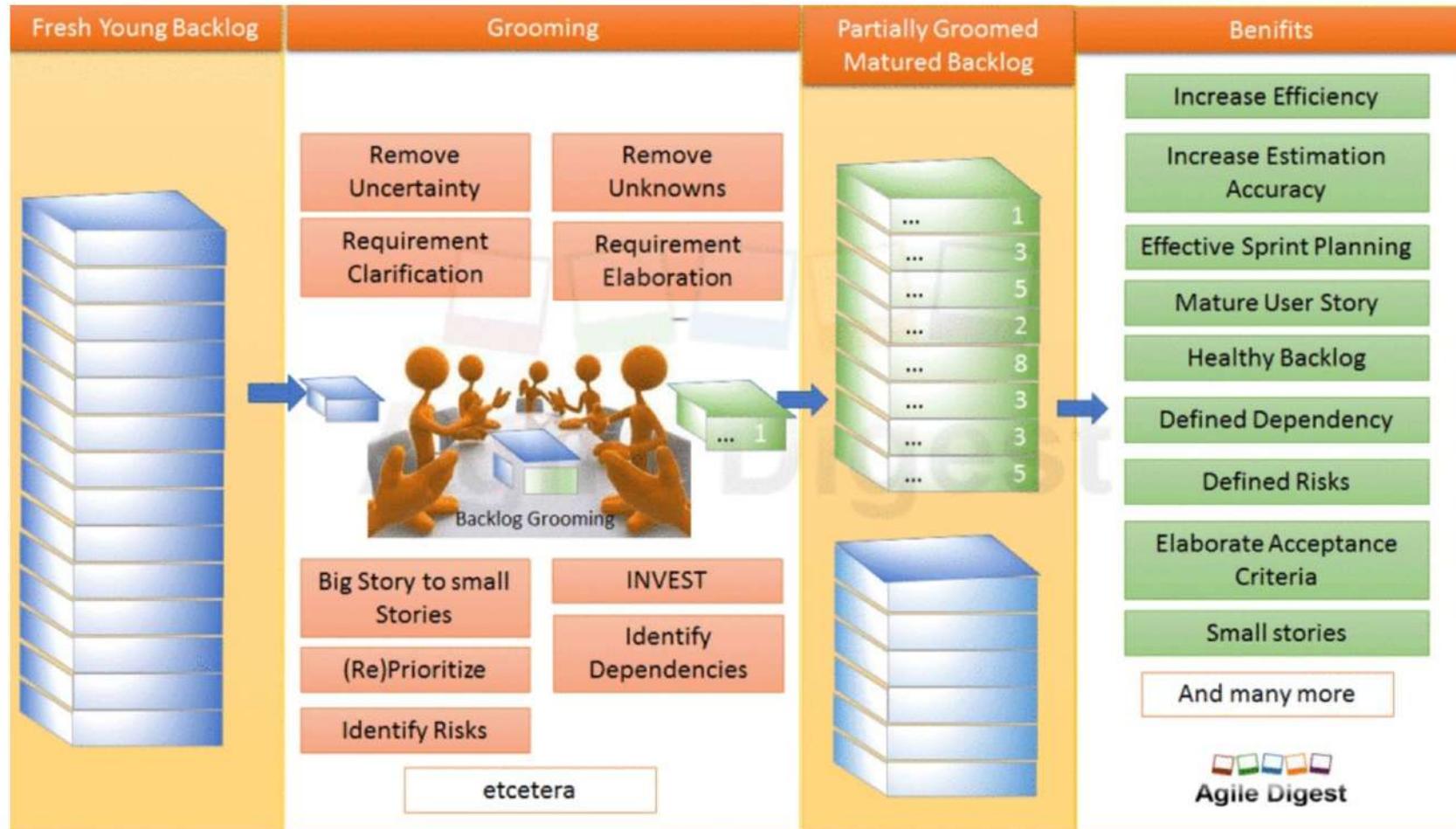
- In addition to the product owner, an engineering leader or manager could be present for a backlog grooming meeting as they add a much needed technical perspective towards estimating and prioritizing items or cutting items from the backlog.
- In this type of grooming session, the product owner acts as a facilitator of the grooming meeting.

# Back log Grooming



# Importance of Backlog Grooming





# The backlog grooming process

- In the course of the development process, a sprint backlog grows ever bigger. Backlog grooming keeps it organized. Before a sprint planning meeting, product owners should review the backlog to ensure:
- The set priorities are correct.
- The prioritized items have all the correct information.
- The feedback from previous meetings has been incorporated.

Sometimes, The backlog can quickly become overcrowded and out of control. To keep it relevant, grooming refines the backlog by cutting out unimportant stories and tasks and maintaining only priority ones.

## Key backlog grooming activities include:

- Eliminating unwanted user stories that don't fit the current product direction.
- Reprioritizing stories to move lower priority items to the bottom of the backlog.
- Breaking down large work items into smaller ones.
- Updating estimates.
- Adding new work items.

The benefits of frequent backlog grooming quickly become apparent as development teams can readily access all the information they need to put together a [sprint action plan](#).

# Benefits of backlog grooming

There are several important reasons to adopt backlog grooming:

- **Keeps the backlog clean**
- **Keeps the backlog relevant**
- **Keeps the whole product team up to date**
- **Increases work velocity**

# Backlog grooming best practices

## Make your product backlog **DEEP**

Roman Pichler, author of “Agile Product Management with Scrum,” used the acronym DEEP to sum up the essential traits of a good product backlog:

- **Detailed Appropriately** — User stories and other items in the product backlog that will be done soon need to be sufficiently well understood by cross-functional teams. Items and initiatives that will not be delivered for a while should be described with less detail.
- **Estimated** — Backlog items at the top should include an accurate estimation of the work needed to deliver them. Conversely, items down the backlog should only be roughly estimated as they are not that well understood yet.

# Backlog grooming best practices

Make your product backlog **DEEP**

- **Emergent** — A product backlog changes over time. As you capture new user insights, the backlog will be changed to adapt to customer needs.
- **Prioritized** — The product backlog should be ordered with the most valuable items at the top and the least valuable at the bottom. Every single backlog item is ranked in relation to its business value and alignment with the strategic goal of the company.

## Sprint planning

- The purpose of sprint planning is to agree on a goal for the next sprint and the set of backlog items (aka [sprint backlog](#)) to achieve it. This involves more than just communicating requirements.
- It's about **learning, considering options and making decisions together.**
- Sprint planning consists of two main components: (1) prioritizing backlog items and (2) agreeing on the number of backlog items in the sprint based on team capacity.

## Sprint planning

- Traditionally, (1) prioritizing backlog items is the responsibility of the product owner, whereas (2) deciding how much to pull in the sprint is the development team's decision. Both should be a discussion with the entire team, while the responsibilities (and the final say) stay with the respective roles.
- The input for a sprint planning session is the product backlog. In order to make the meeting as effective as possible, the top of the backlog — the most important backlog items that should be tackled next — should be well “groomed” or rather “refined” ahead of time.

# Last Session

- **Agile Requirements –**
- User personas,
- story mapping,
- user stories,
- 3Cs, INVEST,
- Acceptance criteria, sprints, requirements,
- product backlog and backlog grooming;
- Sprint Planning

# User Stories

User Stories are often deemed to comprise three elements - **the 3C's**

- Card
- Conversation
- Confirmation

# User Stories

## Front of the CARD

- This states all of the scenarios of the users involved. These should read:
- As a **SOME ROLE**,  
I want to **DO SOMETHING**,  
So that I **CAN GET SOME BENEFIT**
- The user stories are critical to lay out exactly who is going to do what, and for what reason(s).

# User Story – the Card

- **The Back of the Card**
- On the back, the Confirmation area contains:
  - Acceptance criteria (the test criteria)
  - These acceptance criteria define, at a high level, the test criteria which will confirm that this user story is working as required. These are not intended to be the full test scripts, but will be used to expand into the appropriate test scenarios and test scripts during Timeboxes, as necessary.

For User Stories at the highest level (sometimes called a project Epic), the acceptance criteria may be used to define the aims of the project using criteria that may be measured after the project has completed (as part of the Benefits Assessment).

# Well constructed User Stories

Bill Wake's INVEST model provides guidance on creating effective User Stories

**Independent**

Stories should be as independent as possible from other stories, to allow them to be moved around with minimal impact and potentially to be implemented independently. If stories are tightly dependent, consider combining them into a single user story.

**Negotiable**

Stories are not a contract. They are "placeholders" for features which the team will discuss and clarify near to the time of development.

**Valuable**

Stories should represent features providing clear business value to the user/owner of the solution and should be written in appropriate language. They should be features, not tasks.

**Estimable**

Stories need to be clear enough to estimate (for the appropriate timeframe), without being too detailed.

**Small**

Stories should be small enough to be estimated. Larger "Epic" stories should be broken down into smaller User Stories as the project progresses. The stories after splitting still follow the INVEST criteria.

**Testable**

Stories need to be worded clearly and concisely so they can be tested.

# Back log Grooming



# Backlog grooming best practices

## Make your product backlog **DEEP**

Roman Pichler, author of “Agile Product Management with Scrum,” used the acronym DEEP to sum up the essential traits of a good product backlog:

- **Detailed Appropriately** — User stories and other items in the product backlog that will be done soon need to be sufficiently well understood by cross-functional teams. Items and initiatives that will not be delivered for a while should be described with less detail.
- **Estimated** — Backlog items at the top should include an accurate estimation of the work needed to deliver them. Conversely, items down the backlog should only be roughly estimated as they are not that well understood yet.

# Backlog grooming best practices

Make your product backlog **DEEP**

- **Emergent** — A product backlog changes over time. As you capture new user insights, the backlog will be changed to adapt to customer needs.
- **Prioritized** — The product backlog should be ordered with the most valuable items at the top and the least valuable at the bottom. Every single backlog item is ranked in relation to its business value and alignment with the strategic goal of the company.

## Sprint planning

- The purpose of sprint planning is to agree on a goal for the next sprint and the set of backlog items (aka [sprint backlog](#)) to achieve it. This involves more than just communicating requirements.
- It's about **learning, considering options and making decisions together.**
- Sprint planning consists of two main components: (1) prioritizing backlog items and (2) agreeing on the number of backlog items in the sprint based on team capacity.

## Sprint planning

- Traditionally, (1) prioritizing backlog items is the responsibility of the product owner, whereas (2) deciding how much to pull in the sprint is the development team's decision. Both should be a discussion with the entire team, while the responsibilities (and the final say) stay with the respective roles.
- The input for a sprint planning session is the product backlog. In order to make the meeting as effective as possible, the top of the backlog — the most important backlog items that should be tackled next — should be well “groomed” or rather “refined” ahead of time.

# Today's Session

- Scrum Story Point
- Benefits of Story point
- Pitfalls of Story points
- Definition of Ready
- Definition of Done
- Tracking Iteration Progress

# What Are Scrum Story Points?

- A [story](#) is the smallest unit of work for a [Scrum team](#), expressed from the perspective of the end-user. Stories are often written as a simple sentence, for example: “As an online banking customer, I want to be able to add payees to my account so I can transfer money.”
- Story points in Scrum are units of measurement used to estimate the effort required to complete a story. When [planning](#) for an [upcoming sprint](#), Scrum teams use story point estimation to gauge how much effort is needed to develop a new software feature or update.
- Why focus on effort instead of hours? The idea is that if you ask two developers how long it will take to complete a story, they may give you two completely different answers. However, if you ask them how much effort a product backlog item will take to complete, they’re more likely to agree.

# How do you estimate story points in Scrum?

- Scrum story points are usually represented using the [Fibonacci sequence](#).
- Many Scrum and Agile teams now use a modified version of the sequence:
- **1, 2, 3, 5, 8, 13, 20, 40, and 100**
- The space between each number makes it easier for Scrum teams to distinguish between them and agree on which one to use.
- When using story point estimation, Scrum teams consider the complexity of the story, the potential risks involved, and the familiarity of the tasks. Then, they assign values to the story points using the following steps:
- **Choose a previous story as a reference point.** Say, for example, you picked base stories with values of two and five respectively — team members can determine a new value of three, as the task being estimated is bigger than the story with two but smaller than five.

**Create a matrix to visualize story point values.** Make a row for each number in the Fibonacci sequence. When you assign values to your story points, place them in the corresponding row.

Story point	Amount of effort required	Amount of time required	Task complexity	Task risk or uncertainty
1	Minimum effort	A few minutes	Little complexity	None
2	Minimum effort	A few hours	Little complexity	None
3	Mild effort	A day	Low complexity	Low
5	Moderate effort	A few days	Medium complexity	Moderate
8	Severe effort	A week	Medium complexity	Moderate
13	Maximum effort	A month	High complexity	High

- **Play story points [planning poker](#).** This Scrum estimation technique helps teams assign values to story points using playing cards to denote the numbers on the Agile Fibonacci sequence. Team members discuss upcoming user stories, then pick the card they feel represents the appropriate value for the story. If everyone chooses the same number, that number is assigned to the story. If even one member chooses a different number, the team discusses the story further until consensus is reached.

Here's how to run a successful planning poker meeting.

- Give your team a defined story point matrix for reference, as well as a set of cards that depict your story point sequence. You can create the cards yourself or download a set.
- Select a user story.
- Discuss the story with your team, like what's involved and what success looks like.
- Have each team member privately select the story point card they feel represents the amount of effort required to complete the story.
- Have your team reveal their card selections at the same time. If the story points align, move on to the next user story. If the story points don't align, continue to discuss the user story until you reach an agreement.
- Repeat the process until you've assigned story points to all the tasks in your product backlog.
- Using your story point matrix as a baseline, determine how many tasks your team can complete in the upcoming sprint.

# Why use story points in Scrum?

- Scrum story points are considered to be **more accurate** than estimating in hours. Greater accuracy enables the product owner to plan sprints more efficiently, ensuring stories are delivered on time.
- **Story points also consider that not every team member works the same way** — one employee could require a day to complete a certain task, while another may only take a few hours. By focusing on effort and doing away with strict due dates, story points take the pressure off team members.
- Finally, **story points promote teamwork**. Estimating story points requires that everyone work together and each team member has a say. This, in turn, leads to happier, more productive Scrum teams.

# Why Use Scrum Story Points?

- Why not just use estimated hours? Similar to the process of using [agile story points](#), traditional software teams give estimates in a time format such as days, weeks, and months.
- Many teams have transitioned to story points, however, in an effort to improve efficiency;

## Few specific reasons to use scrum story points:

- Dates and straight time measurements don't account for non-project related work that creeps into our working days: emails, meetings, interviews that a team member may be involved in.
- The same is true of [unexpected non-work time](#). If someone on the team is late due to traffic or out sick, it can throw a whole date-based schedule off completely.
- Dates often have an emotional attachment to them. "I'll have it done by Tuesday at ten," is fine on the face of it. But if you can't get it done by Tuesday at ten, there may be guilt, stress, and other emotions that further impair your efficiency. Relative estimation will remove the emotional attachment.

- Let's say your user story is "As a user, I want to be able to submit feedback and questions through the site to better understand product features." You'd assign this user story a story point—again, the amount of effort you think is required to complete the story. You can then break down the story into smaller tasks, such as scoping and designing the feedback form, writing the code for the form, staging the page and testing the form, and publishing the page.

# Benefits of using Story points

- **Drive faster planning.** Story points are relative, meaning you calculate the value of one story point by comparing it to similar, already estimated points. Using a relative scoring method leads to faster estimation over time—a big win for your team.
- **Take unpredictability and risk into account.** Story points account for elements like unpredictability and risk. Using these factors in your planning takes the guesswork out of estimating, letting you more accurately scope effort.
- **Remove skills bias from your planning and get your team on the same page.** Relying on individual team member estimations isn't always best. After all, a senior team member will probably give a pretty different effort estimation than a junior team member. Story points prevent these issues by encouraging collaboration in the form of planning poker meetings.
- **Create meaningful deadlines.** No one likes arbitrary deadlines, but that's often what you get when you use other estimation techniques. Since story points are more nuanced, they result in meaningful due dates.
- **Build better estimations going forward.** One of the major perks of story points is they're adaptable and reusable. That means once you've created a story point matrix and held your first sprint, you can use your learnings to reevaluate your original story point values and develop more accurate estimations.

# Pitfalls of using Story points

- **Using story points that aren't relative.** The relative nature of story points makes understanding how tasks compare to each other easier for your team. That's why you shouldn't assign points arbitrarily. Remember: story points should scale relative to each other.
- **Using hours for your story points.** Since time estimation doesn't account for factors like complexity and uncertainty, using hour estimates or days as your story points is contrary to their goal. Instead, factor in the three components we've gone over—complexity, risk, and repetition—to determine your story point values.
- **Taking the average of your team's scores when planning poker.** If your team's story point estimations don't match up, don't take the average of the points. Instead, open the floor and discuss the discrepancy. Once you have a better understanding of why the estimations don't align, you can find a story point everyone agrees on.
- **Assigning story points to user stories that are too large.** Not everything needs a story point. If a user story is so large you feel none of the story point values in your matrix account for the effort required, it may be worth breaking it down further.
- **Failing to clarify expectations with team members about story point values.** If your team doesn't understand story points, they're not useful. Luckily, there's an easy solution: talk to your team about the estimation method. Walk them through an example story point matrix and chat through each task so they accurately assign story points.

## Estimation is hard

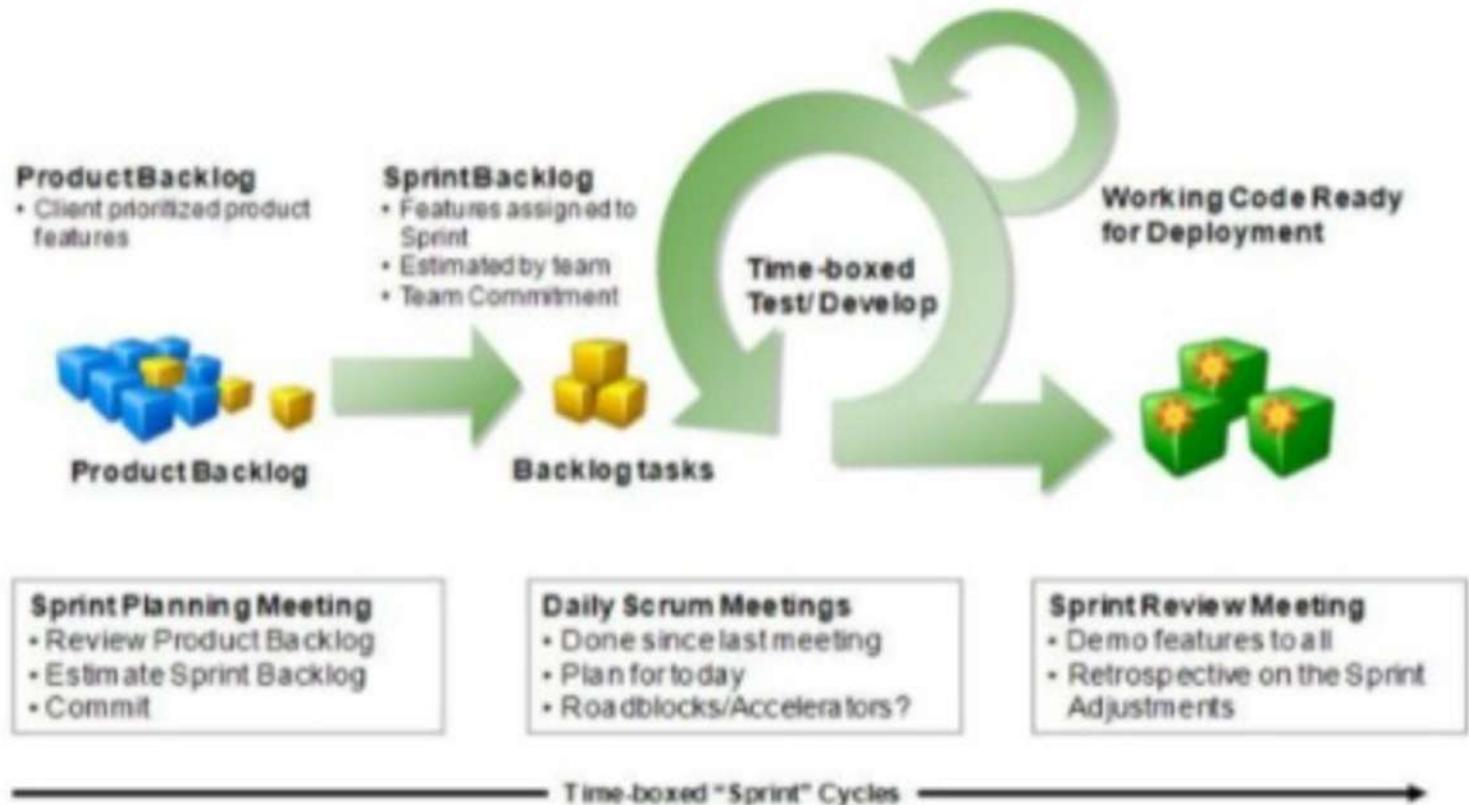
- For software developers, it can be one of the most difficult aspects of their job, as it involves the one thing that cannot easily be measured - the customer's wishes.
- The line between what a customer wants and what they need can be a very fine one, and it is not always simple for developers to determine which is which.
- This is why user stories are important - **scrum story points are used primarily to get the customer's input in a measurable way** - and then a good development team has a metric to work with.

# Collaborating with the Product Owner

- In an agile development framework such as scrum, **the product owner prioritizes the backlog; this is the ordered list of work**, that contains short descriptions of all desired features and fixes for a product.
- **Product owners** capture requirements from the customer, but they don't always understand the details of implementation, because they are **not always software designers**; it is their role to translate user needs into something measurable that developers can use.
- So a **good estimation is essentially the bridge between customer needs and developer duties**; it can give the product owner insight to help them make an accurate assessment of each item's relative priority.

# Collaborating with the Product Owner

- When the engineering team begins its estimation process, additional questions often arise about customer requirements and user stories.
- This is a good thing; these questions help the entire team understand the work fully.
- For product owners, breaking down work items into pieces using story points helps them prioritize all areas of work.
- They may even find some new ones in the process! And once they have estimates from the development team, it is very common for a product owner to reorder items on the backlog as the story points help them gain new insights.



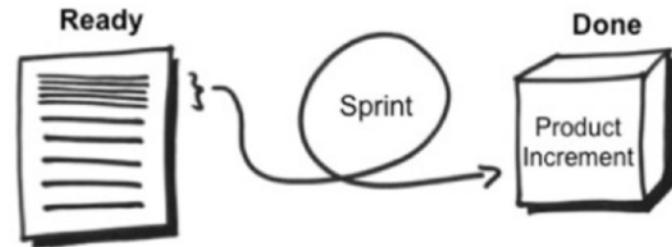
## Definition of Ready (DoR) vs. Definition of Done (DoD)

- A **sprint** is a **time-boxed development cycle** that takes high-priority items off the Sprint Backlog and turns them into a product increment.
- However, in order to successfully pull items into the **current sprint**, it is important that the **defined user stories** are “**ready**” – pulling unfinished or unrefined user stories into a sprint causes problems during the implementation cycle, as it follows the old principle of “garbage in, garbage out”.
- If developers work off of insufficiently detailed or defined user stories, they are unlikely to produce high quality code.

A “ready” backlog item needs to be clear, feasible and testable:

- A user story is **clear** if all Scrum team members have a **shared understanding** of what it means. Collaboratively writing user stories, and adding acceptance criteria to the high-priority ones facilitates clarity
- An item is **testable** if there is an effective way to determine if the functionality works as expected. Acceptance criteria ensure that each story can be tested
- A user story is **feasible** if it can be completed in one sprint, according to the Definition of Done. If this is not achievable, it needs be broken down further

- Simply stated, the Definition of Ready defines the criteria that a specific user story has to meet before being considered for estimation or inclusion into a sprint.



- Whereas a Definition of Ready is focused on user story level characteristics, the Definition of Done is focused on the sprint or release level.
- Essentially, a DoD represents the acceptance criteria for a sprint or release. It spells out what the Development Team has to cover in order for the product increment to be considered “done”.

## **Sample Definition of Ready (DoR)**

- User Story is clear
- User Story is testable
- User Story is feasible
- User Story defined
- User Story Acceptance Criteria defined
- User Story dependencies identified
- User Story sized by Development Team
- Scrum Team accepts User Experience artefacts
- Performance criteria identified, where appropriate
- Scalability criteria identified, where appropriate
- Security criteria identified, where appropriate
- Person who will accept the User Story is identified
- Team has a good idea what it will mean to Demo the User Story

- The Definition of Done is an agreement between Development Team and the Product Owner on what needs to be completed for each user story – and it is often standardized across the company in order to guarantee consistent delivery of quality.

Things that **commonly addressed in the Definition of Done** are:

- **Operating environments** and at what level of integration are user stories expected to work (what specific version of Linux, what specific version of Android, iOS, or browser)?
- **What level of documentation is required** (automatically generated Javadoc vs. fully edited end user documentation)?
- **What are the quality expectations** (basic functionality works for demo purposes vs. fully designed and bullet proofed app)?
- **What are the security expectations** (no security implemented vs. security examined at all levels, from code reviews, code scans, up through network security configuration)?
- **Scalability expectations** (scalable for demo purposes up to 10 concurrent users vs. scalable to 100,000 concurrent users)?

- Essentially the Definition of Done are the agreed upon acceptance criteria that the Product Owner will use to accept the product increment at the end of the sprint.
- Please note that the DoD may be different for sprints vs. releases, meaning intermediate sprints might have a less stringent DoD than the final couple of sprints before you are planning to release to market.

## **Sample Definition of Done**

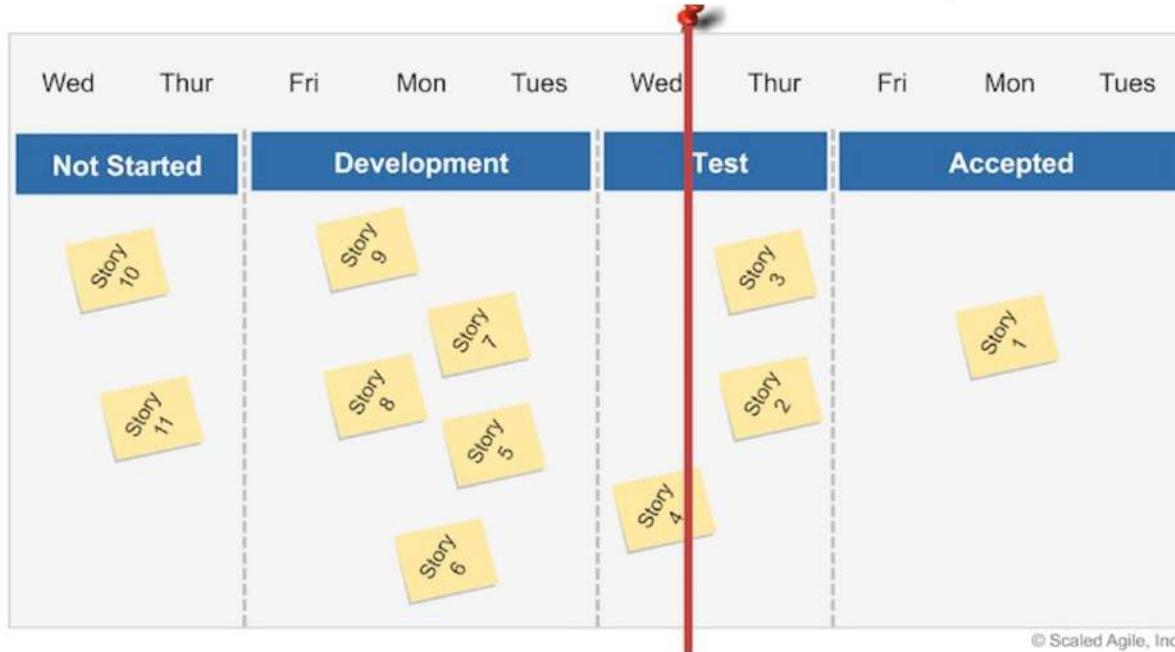
- Code produced (all 'to do' items in code completed)
- Code commented, checked in and run against current version in source control
- Peer reviewed (or produced with pair programming) and meeting development standards
- Builds without errors
- Unit tests written and passing
- Deployed to system test environment and passed system tests
- Passed UAT (User Acceptance Testing) and signed off as meeting requirements
- Any build / deployment / configuration changes are implemented / documented / communicated
- Relevant documentation / diagrams produced and / or updated
- Remaining hours for task set to zero and task closed

- **Definition of Ready (DoR):** It is defined so as to keep track of the items at the top of the Product Backlog that has fulfilled certain pre-conditions and can be added to a Sprint so that the Developers could complete it before the end of the Sprint.
- **Definition of Done(DoD):** It is defined as a checklist for all the Sprint Backlog items that have passed all the conditions and acceptance criteria and are ready to be accepted by the users, consumers, or teams.

- The main elements of a successful iteration execution include:
- **Tracking iteration progress** – using story and Kanban boards to follow the progress of the iteration.
- **Building stories serially and incrementally** – this avoids mini-waterfalls within the iteration
- **Constant communication** – continuous communication and synchronization via DSU events(Daily Stand Up)
- **Improving flow** – optimizing flow by managing Work in Process (WIP), building quality in, and continuously accepting stories throughout the iteration
- **Program execution** – working together as an ART to achieve program PI objectives

# Tracking Iteration progress

- Tracking iteration progress requires visibility into the status of user stories, defects, and other team activities. For this purpose, most teams use a Big Visible Information Radiator (BHIR) on a wall in the team room. Kanban teams use their Kanban board, while ScrumXP teams would use a storyboard.



- With this simple storyboard, the team just moves the red ribbon to the current day, providing an immediate and easy-to-understand visual assessment of iteration progress.
- It's now clear that the iteration shown in Figure , which has only 5 days remaining, is at risk and the team should figure out the best way of completing the iteration.
- The storyboard can be shared with remote participants or stakeholders using a webcam, email, wiki, or Agile project management tooling. But this is usually in addition to the physical BViR.

**Building stories serially and incrementally**: Teams should avoid the tendency to waterfall the iteration and instead ensure that they are completing multiple **define-build-test** cycles in the course of the iteration

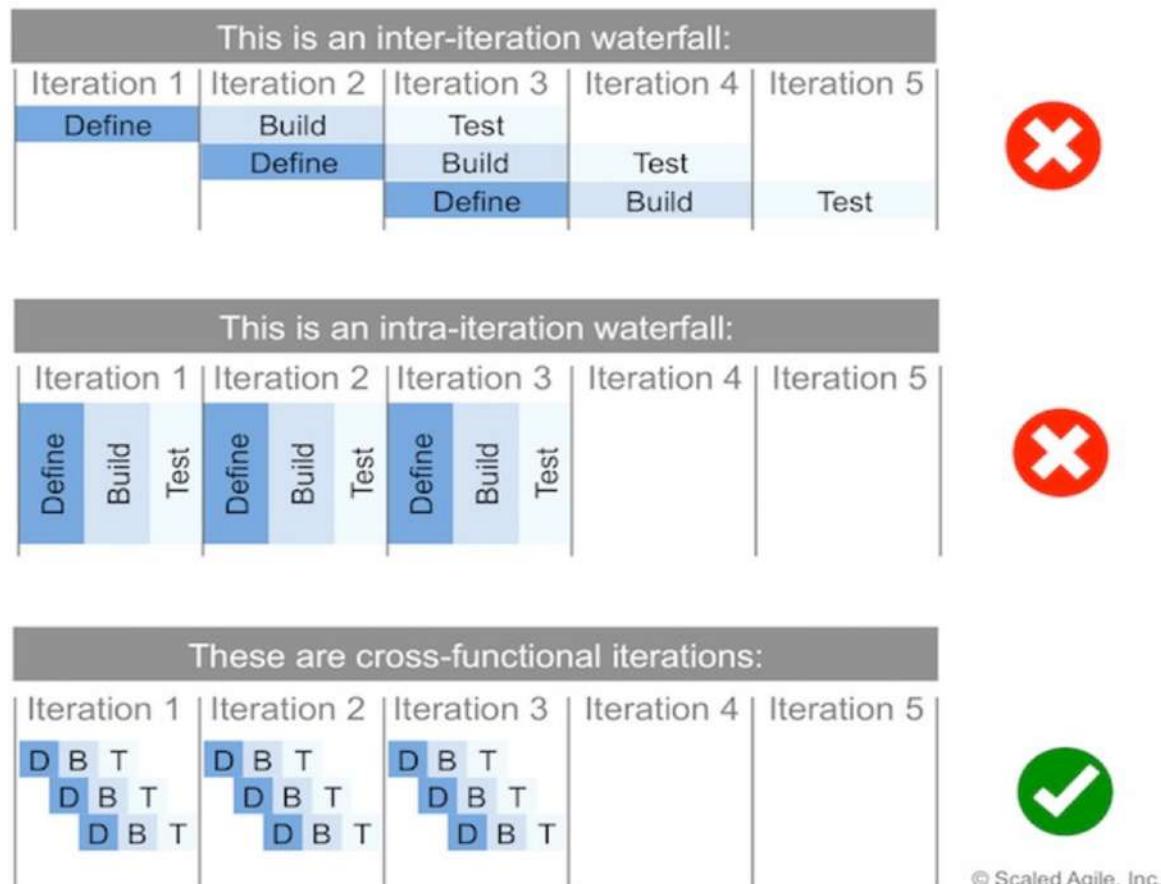


Figure 2. Avoid the mini-waterfall with cross-functional Iterations

# Building stories Incrementally

Figure 3 illustrates how implementing stories in thin, vertical slices is the foundation for incremental development, integration, and testing.

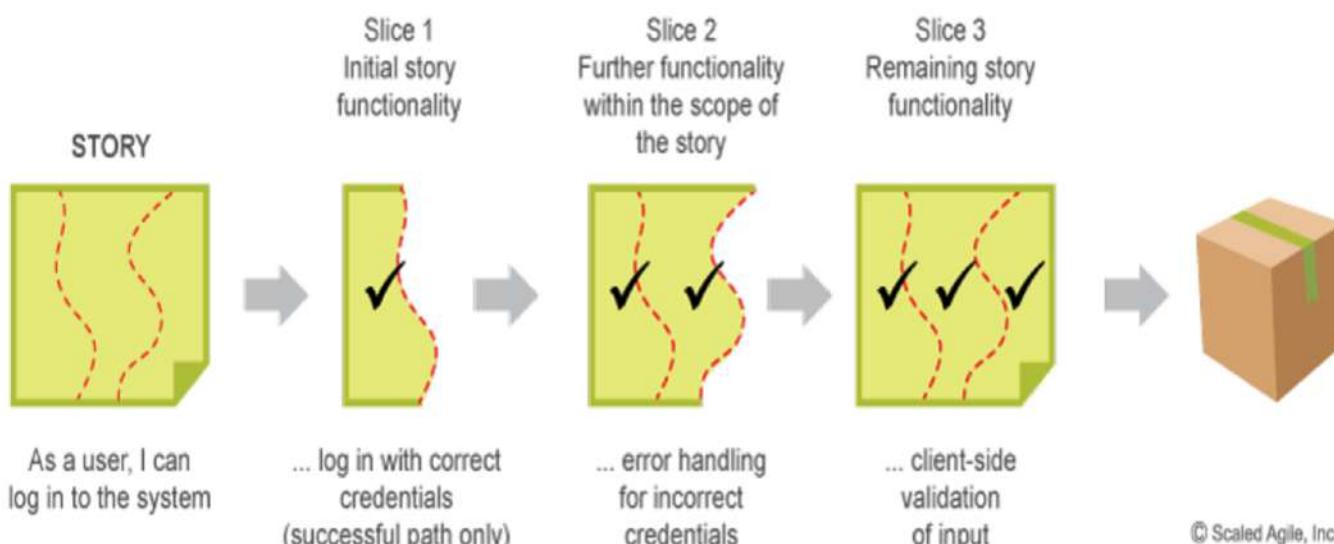


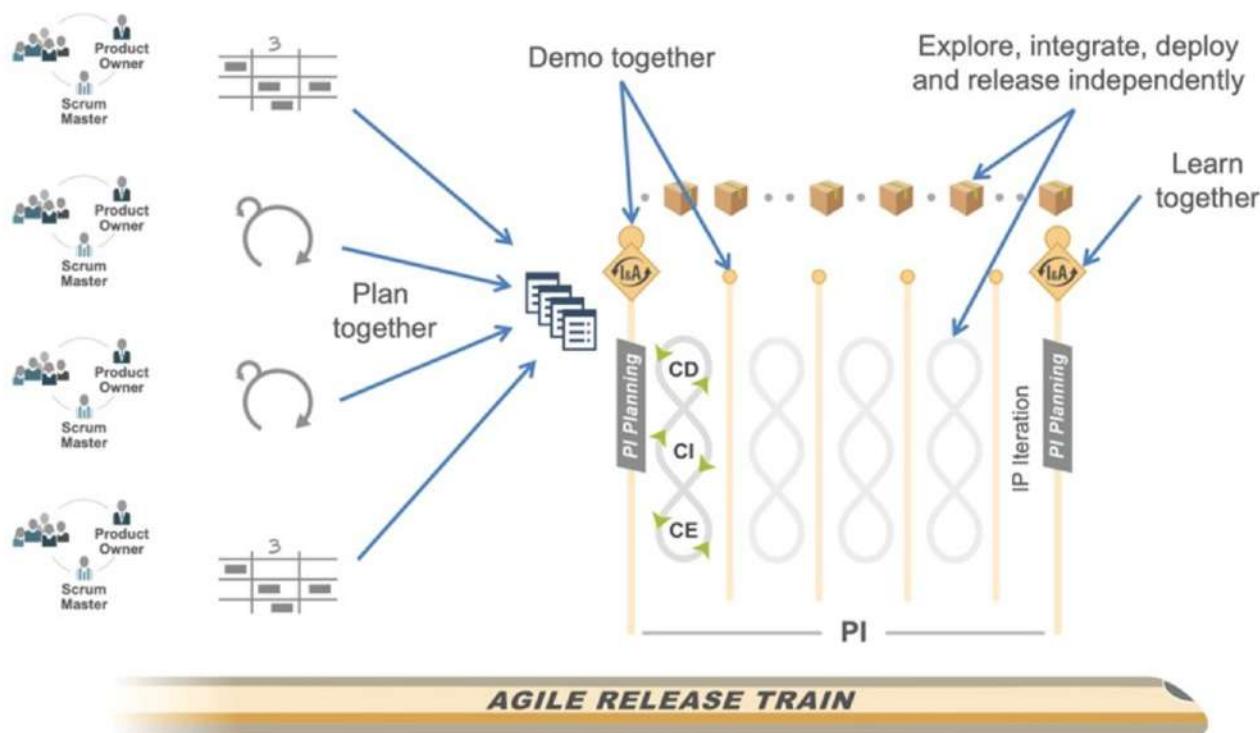
Figure 3. Implementing stories in vertical slices is the key to incremental development

© Scaled Agile, Inc.

- **Building stories this way enables a short feedback cycle** and allows Agile teams to operate with a smaller increment of the working system, supporting continuous integration and testing.
- It allows team members to refine their understanding of the functionality, and it facilitates pairing and more frequent integration of working systems.
- The dependencies within and across teams and even trains can be managed more effectively, as the dependent teams can consume the new functionality sooner.
- Incrementally implementing stories helps reduce uncertainty and risk, validates architectural and design decisions, and promotes early learning and knowledge sharing.

## Focusing on Program Execution

- The ultimate goal of all Agile teams is the successful execution of the ART's (Agile Release Train) program PI (Program Increment) objectives. The teams plan, demo, and learn together, as illustrated in Figure 4, which avoids them focusing solely on local concerns. This alignment enables teams to more independently explore, integrate, deploy, and release value.



© Scaled Agile, Inc.

## Chapter 7

# Requirements Engineering

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

I CAN'T START THE  
PROJECT BECAUSE THE  
USER WON'T GIVE ME  
HIS REQUIREMENTS.

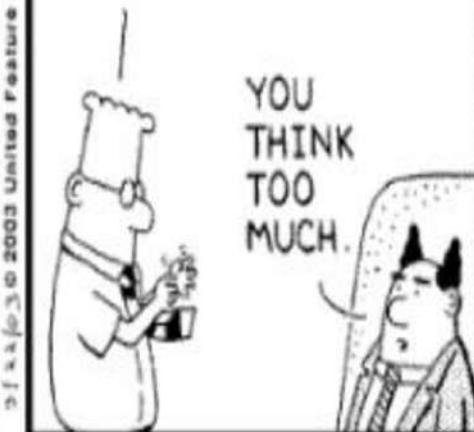


[www.dilbert.com](http://www.dilbert.com) scottadams@aol.com

START MAKING  
SOMETHING ANYWAY.  
OTHERWISE WE'LL  
LOOK UNHELPFUL.

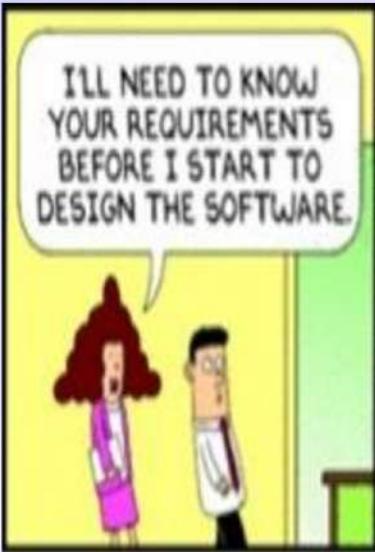


SO, OUR PLAN IS TO  
CLEVERLY HIDE OUR  
COMPETENCE.



YOU  
THINK  
TOO  
MUCH.





# The Problems with our Requirements Practices



- We have trouble understanding the requirements that we do acquire from the customer
- We often record requirements in a disorganized manner
- We spend far too little time verifying what we do record
- We allow change to control us, rather than establishing mechanisms to control change
- Most importantly, we fail to establish a solid foundation for the system or software that the user wants built





How the customer explained it



How the project leader understood it



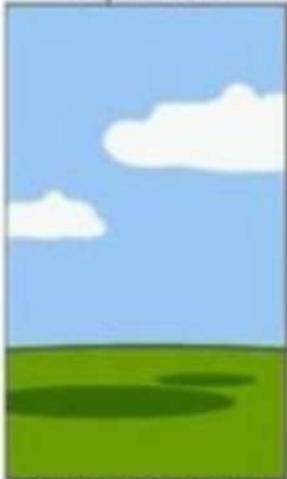
How the engineer designed it



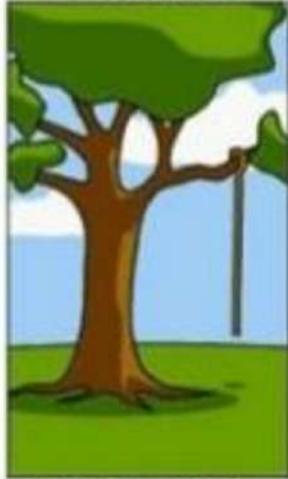
How the programmer wrote it



How the sales executive described it



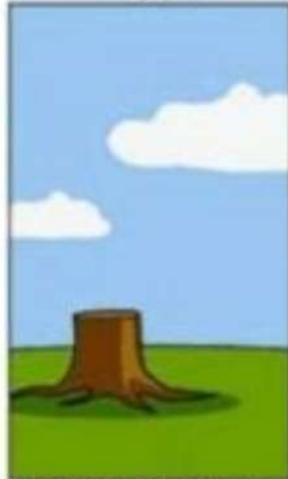
How the project was documented



What operations installed



How the customer was billed



How the helpdesk supported it



What the customer really needed



# The Problems with our Requirements Practices



- We have trouble understanding the requirements that we do acquire from the customer
- We often record requirements in a disorganized manner
- We spend far too little time verifying what we do record
- We allow change to control us, rather than establishing mechanisms to control change
- Most importantly, we fail to establish a solid foundation for the system or software that the user wants built



# A Solution: Requirements Engineering



- Begins during the communication activity and continues into the modeling activity
- Builds a bridge from the system requirements into software design and construction
- Allows the requirements engineer to examine
  - the context of the software work to be performed
  - the specific needs that design and construction must address
  - the priorities that guide the order in which work is to be completed
  - the information, function, and behavior that will have a profound impact on the resultant design



## Requirement Engineering



- RE helps software engineer to better understand the problem they will work to solve
- Participant : Software Engineers, managers, customers and end users
- RE is a software engineering action that begin during the communication activity and continues into the modeling activity



## **RE Provides the appropriate mechanism for**

- Understanding what the customer want
- Analyzing need
- Assessing feasibility
- Negotiating a reasonable solution
- Specifying a solution unambiguously
- Validating the specification
- Managing the requirement as they are transformed into an operational system

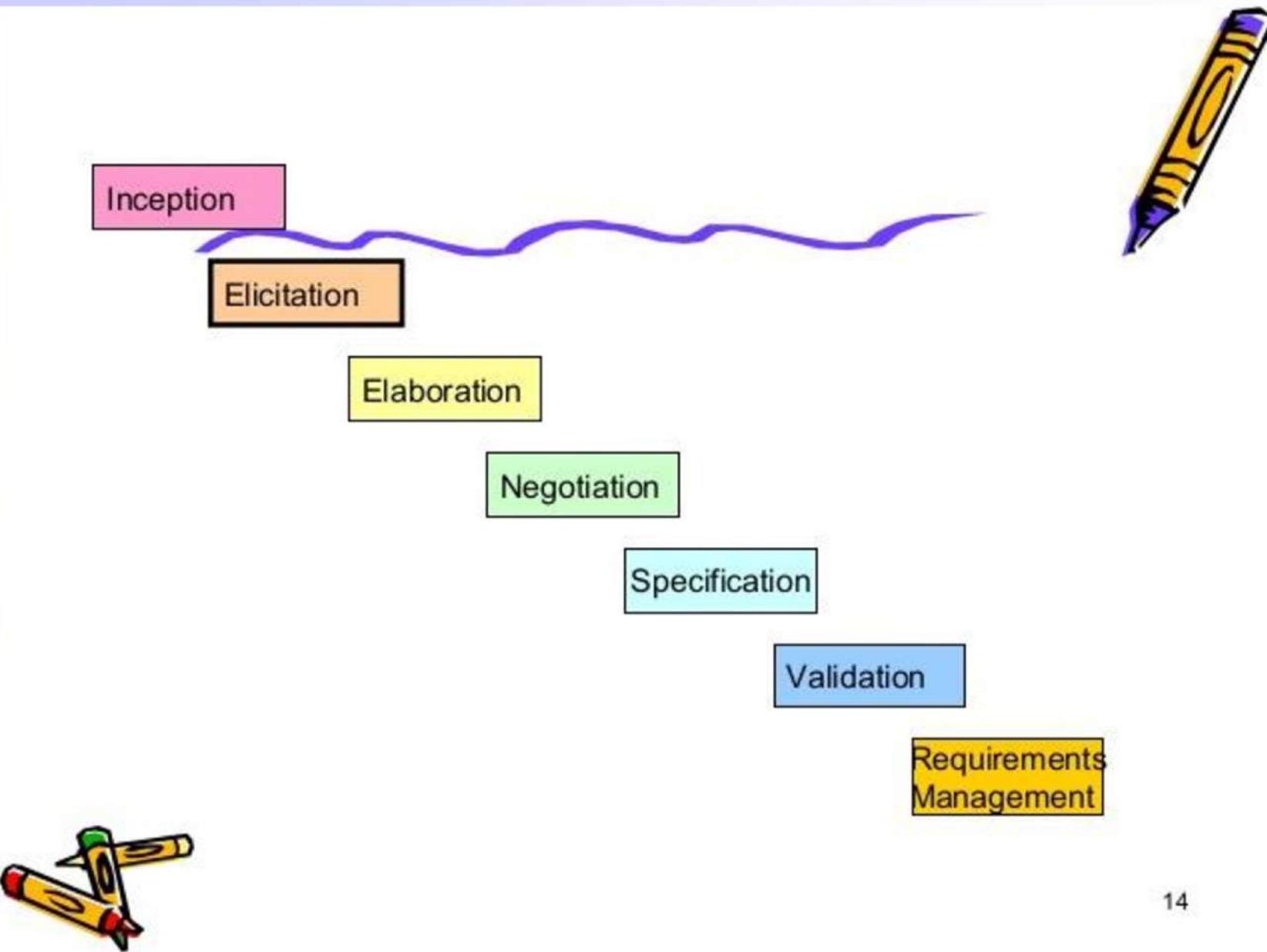


# Requirements Engineering Tasks



- Seven distinct tasks
  - Inception
  - Elicitation
  - Elaboration
  - Negotiation
  - Specification
  - Validation
  - Requirements Management
- Some of these tasks may occur in parallel and all are adapted to the needs of the project
- All strive to define what the customer wants
- All serve to establish a solid foundation for the design and construction of the software





## Requirement Engineering Tasks



- **Inception**—Establish a basic understanding of the problem and the nature of the solution.
- **Elicitation**—Draw out the requirements from stakeholders.
- **Elaboration**—Create an analysis model that represents information, functional, and behavioral aspects of the requirements.
- **Negotiation**—Agree on a deliverable system that is realistic for developers and customers.
- **Specification**—Describe the requirements formally or informally.
- **Validation**—Review the requirement specification for errors, ambiguities, omissions, and conflicts.
- **Requirements management**—Manage changing requirements.



## Inception Task



- During inception, the requirements engineer asks a set of questions to establish...
  - A basic understanding of the problem
  - The people who want a solution
  - The nature of the solution that is desired
  - The effectiveness of preliminary communication and collaboration between the customer and the developer
- Through these questions, the requirements engineer needs to...
  - Identify the stakeholders
  - Recognize multiple viewpoints
  - Work toward collaboration
  - Break the ice and initiate the communication



## The First Set of Questions



These questions focus on the customer, other stakeholders, the overall goals, and the benefits

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?



## The Next Set of Questions



These questions enable the requirements engineer to gain a better understanding of the problem and allow the customer to voice his or her perceptions about a solution

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?



## The Final Set of Questions



These questions focus on the effectiveness of the communication activity itself

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?



## Elicitation



- It certainly simple enough, but...
- Why difficult
  - **Problem of Scope**
    - The boundary of the system is ill-defined
  - **Problem of Understanding**
    - The customer/users are not completely sure of what is needed
  - **Problem of volatility**
    - The requirement change over time
- To help overcame these problem, requirement engineers must approach the requirement gathering activity in an organized manner





YOUR USER REQUIREMENTS INCLUDE FOUR HUNDRED FEATURES.



[scottadams@aol.com](mailto:scottadams@aol.com)

DO YOU REALIZE THAT NO HUMAN WOULD BE ABLE TO USE A PRODUCT WITH THAT LEVEL OF COMPLEXITY?

[www.dilbert.com](http://www.dilbert.com)



© 2001 United Feature Syndicate, Inc.

GOOD POINT.  
I'D BETTER ADD  
"EASY TO USE"  
TO THE LIST.



## Elicitation Process Guideline



- meetings are conducted and attended by both software engineers and customers
  - rules for preparation and participation are established
  - an agenda is suggested
  - a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
  - a "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used
  - the goal is
    - to identify the problem
    - propose elements of the solution
    - negotiate different approaches, and
    - specify a preliminary set of solution requirements
- 

## Elaboration



- Expand requirement into analysis model
  - Elements of the analysis model
    - Scenario-based elements
      - Functional—processing narratives for software functions
      - Use-case—descriptions of the interaction between an “actor” and the system
    - Class-based elements
      - Implied by scenarios
    - Behavioral elements
      - State diagram
  - Flow-oriented elements
    - Data flow diagram



## ENGINEERING LIAISON

TELL ME YOUR  
PROJECT STATUS AND  
I'LL TRANSLATE FOR  
OUR CLIENTS.

THE PROJECT WILL  
NEVER BE COMPLETED  
BECAUSE OUR IDIOT  
CLIENTS CHANGE  
THE REQUIREMENTS  
EVERY OTHER DAY.

I'LL JUST  
SAY YOU'RE  
DRUNK.



## Negotiation



- Agree on a deliverable system that is realistic for developers and customers
- SW team & other project stakeholders negotiate the priority, availability, and cost of each requirement
- The Process are :
  - Identify the key stakeholders
    - These are the people who will be involved in the negotiation
  - Determine each of the stakeholders "win conditions"
    - Win conditions are not always obvious



## Negotiate

- Work toward a set of requirements that lead to "win-win"

# The Art of Negotiation



- Recognize that it is not competition
- Map out a strategy
- Listen actively
- Focus on the other party's interests
- Don't let it get personal
- Be creative
- Be ready to commit



## Specification



- Final work product produced by requirement engineer.
- Can be any one (or more) of the following:
  - A written document
  - A set of models
  - A formal mathematical
  - A collection of user scenarios (use-cases)
  - A prototype



## *Technically Speaking, "requirement" ≠ "specification"*

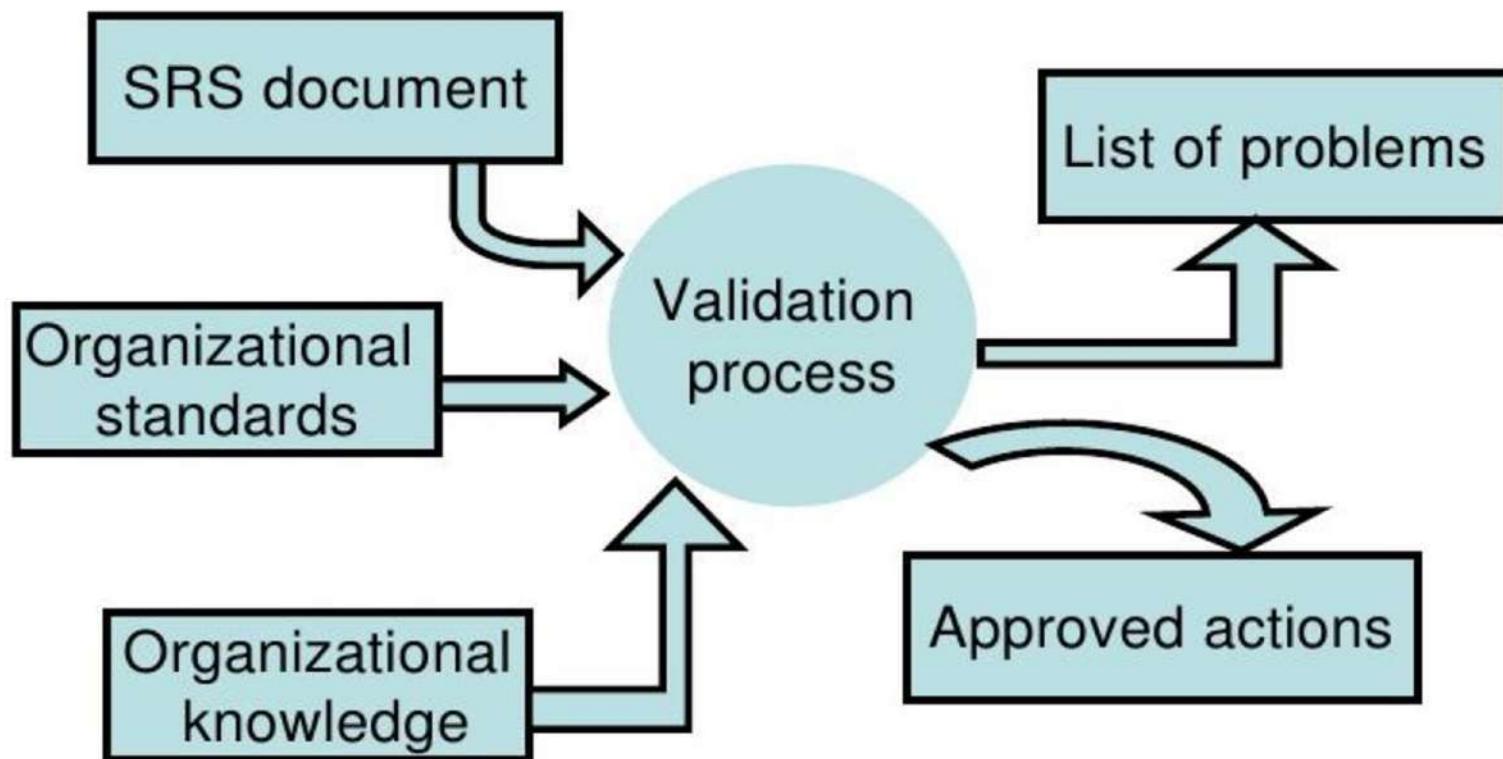


- Requirement - understanding between customer and supplier
- Specification - what the software must do
- Requirements that are not in the SRS
  - Costs
  - Delivery dates
  - Acceptance procedures
  - etc



## *Requirements Validation*

---



## Validation



examine the specification to ensure that SW requirement is not ambiguous, consistent, error free etc

### A review mechanism that looks for

- errors in content or interpretation
- areas where clarification may be required
- missing information
- inconsistencies (a major problem when large products or systems are engineered)
- conflicting or unrealistic (unachievable) requirements.



# Validation Vs. Verification



- Validation: "Am I building the right product?"  
checking a work product against higher-level work products or authorities that frame this particular product.
  - Requirements are validated by stakeholders
- Verification: "Am I building the product right?"  
checking a work product against some standards and conditions imposed on this type of product and the process of its development.
  - Requirements are verified by the analysts mainly



## Validation Cont'd



- A review of the analysis model addresses the following question :
  - Is each requirement consistent with the overall objective for the system/product?
  - Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
  - Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
  - Is each requirement bounded and unambiguous?
  - Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
  - Do any requirements conflict with other requirements?



# Requirements Management Task

- During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds
- Each requirement is assigned a unique identifier
- The requirements are then placed into one or more traceability tables
- These tables may be stored in a database that relate features, sources, dependencies, subsystems, and interfaces to the requirements
- A requirements traceability table is also placed at the end of the software requirements specification

- Features traceability table
  - How req relate to product features
- Source traceability table
  - Identify source of req
- Dependency traceability table
  - Dependency between req
- Subsystem traceability table
  - Categorize req by sub-systems
- Interface traceability table
  - How req relate to interface

		Specific aspect of the system or its environment					
Requirement		A01	A02	A03	A04	A05	Aii
		R01			✓	✓	
R02		✓		✓			
R03		✓			✓		✓
R04			✓			✓	
R05		✓	✓		✓		✓
Rnn		✓		✓			

# Summary

Inception

Elicitation

Elaboration

Negotiation

Specification

Validation

Requirements  
Management





**The problems is not that there are problems.  
The problem is expecting otherwise and  
thinking that having problems is a problem**

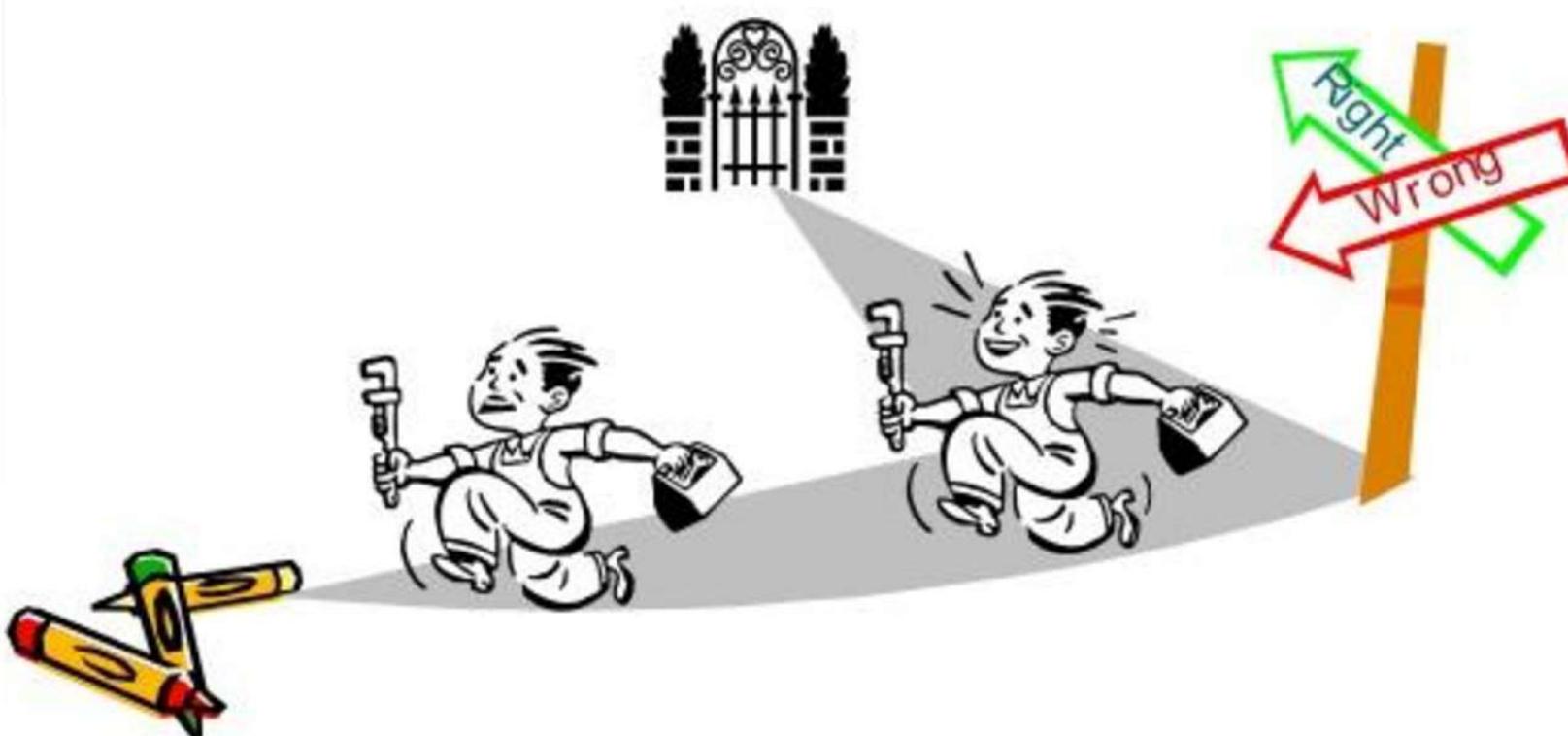
*Theodore Rubin*



## Need to focus



Moving in the wrong direction at a fast pace is still moving in the wrong direction.



# Requirements Elicitation

- Collaborative Requirements Gathering
- QFD
- User Scenarios
- Elicitation Work Products

# Quality Function Deployment



## QFD



# Quality Function Deployment Is:



# 1. Introduction to QFD



Requirements Engineer

?

**QFD**

## 1(a) QFD - Definition

**VOICE OF THE  
CUSTOMER**

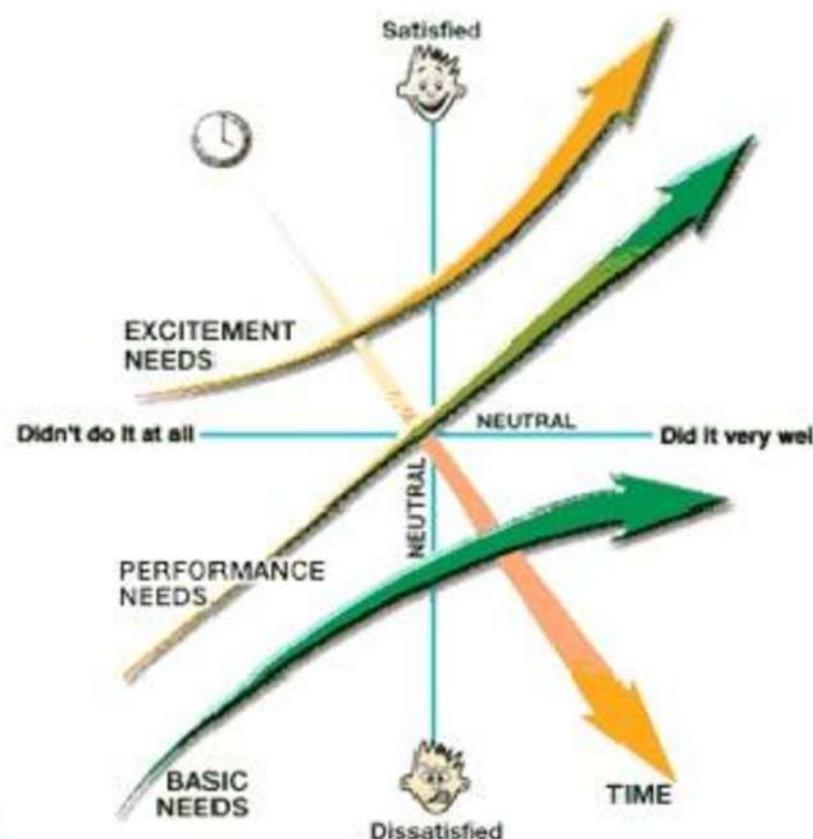


**QFD**



**CUSTOMER  
SATISFACTION**

## 1(b) QFD - Benefits



[ASI, 2000]

## Information on QFD....



- Developed in Japan in the mid 1970s
- Introduced in USA in the late 1980s
- Toyota was able to reduce 60% of cost to bring a new car model to market
- Toyota decreased 1/3 of its development time
- Used in cross functional teams
- Companies feel it increased customer satisfaction



## Yoji Akao

Developed QFD in  
Japan in 1966

## Why....?



- Product should be designed to reflect customers' desires and tastes.
- House of Quality is a kind of a conceptual map that provides the means for interfunctional planning and communications
- To understand what customers mean by quality and how to achieve it from an engineering perspective.
- HQ is a tool to focus the product development process



## Important points



- Should be employed at the beginning of every project (original or redesign)
- Customer requirements should be translated into measurable design targets
- It can be applied to the entire problem or any subproblem
- First worry about what needs to be designed then how
- It takes time to complete





What The  
Customer Wanted



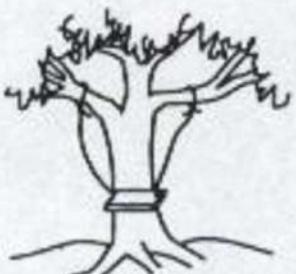
As Sales  
Ordered It



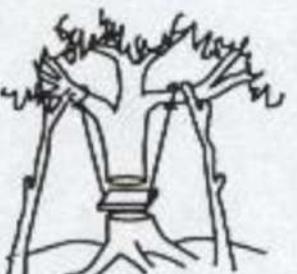
As Market Planning  
Requested It



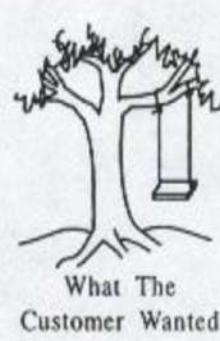
As Engineering  
Designed It



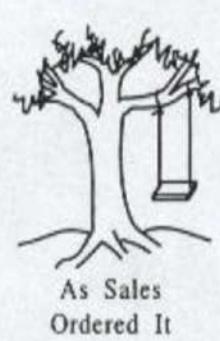
As Manufacturing  
Made It



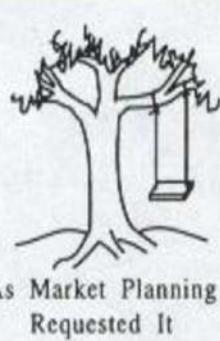
As Field Service  
Installed It



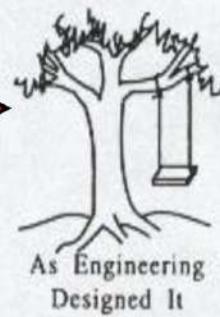
What The  
Customer Wanted



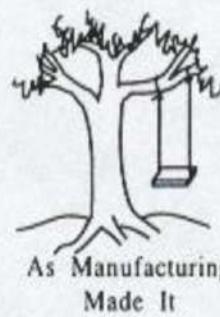
As Sales  
Ordered It



As Market Planning  
Requested It



As Engineering  
Designed It



As Manufacturing  
Made It



As Field Service  
Installed It

**QFD Target**



# QFD identifies 3 types of req:

- Normal requirements
- Expected requirements
- Exciting requirements

# Normal Requirements

- Objectives stated for a product
- Easily identified during meeting with customer
- eg. Requested type of graphical display

# Expected Requirements

- Customer does not explicitly state them
- If not present->system would be meaningless
- A software package for presentation (like Microsoft Power Point )must have option of ‘new slide insert’.

# Exciting Requirements

- Beyond customer expectation
- Eg. Spell check facility in Microsoft word

- **Function deployment:** determines the “value” (as perceived by the customer) of each function required of the system
- **Information deployment:** identifies data objects and events, ties them to functions
- **Task deployment:** examines the behavior of the system
- **Value analysis:** determines the relative priority of requirements

- Raw data- customer interviews, surveys, examination of historical data
- Translated into customer voice table
- Diagrams, matrices, evaluation methods used to extract expected requirements

# Elicitation Work Products

- Bounded statement of scope for the system
- List of customers, stakeholders and users who participated in req. elicitation
- list of requirements
- Set of usage scenarios

# Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error

# Requirements checking

- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology
- Verifiability. Can the requirements be checked?

# Requirements validation techniques

- Requirements reviews
  - Systematic manual analysis of the requirements
- Prototyping
  - Using an executable model of the system to check requirements. Covered in Chapter 8
- Test-case generation
  - Developing tests for requirements to check testability
- Automated consistency analysis
  - Checking the consistency of a structured requirements description

# Requirements reviews

- Regular reviews should be held while the requirements definition is being formulated
- Both client and contractor staff should be involved in reviews
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage

# Review checks

- Verifiability. Is the requirement realistically testable?
- Comprehensibility. Is the requirement properly understood?
- Traceability. Is the origin of the requirement clearly stated?
- Adaptability. Can the requirement be changed without a large impact on other requirements?

# Requirements

- Functional
- Non-functional

# Functional Req.

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations
- Library system- Users can search for, download and print the articles for personal study.

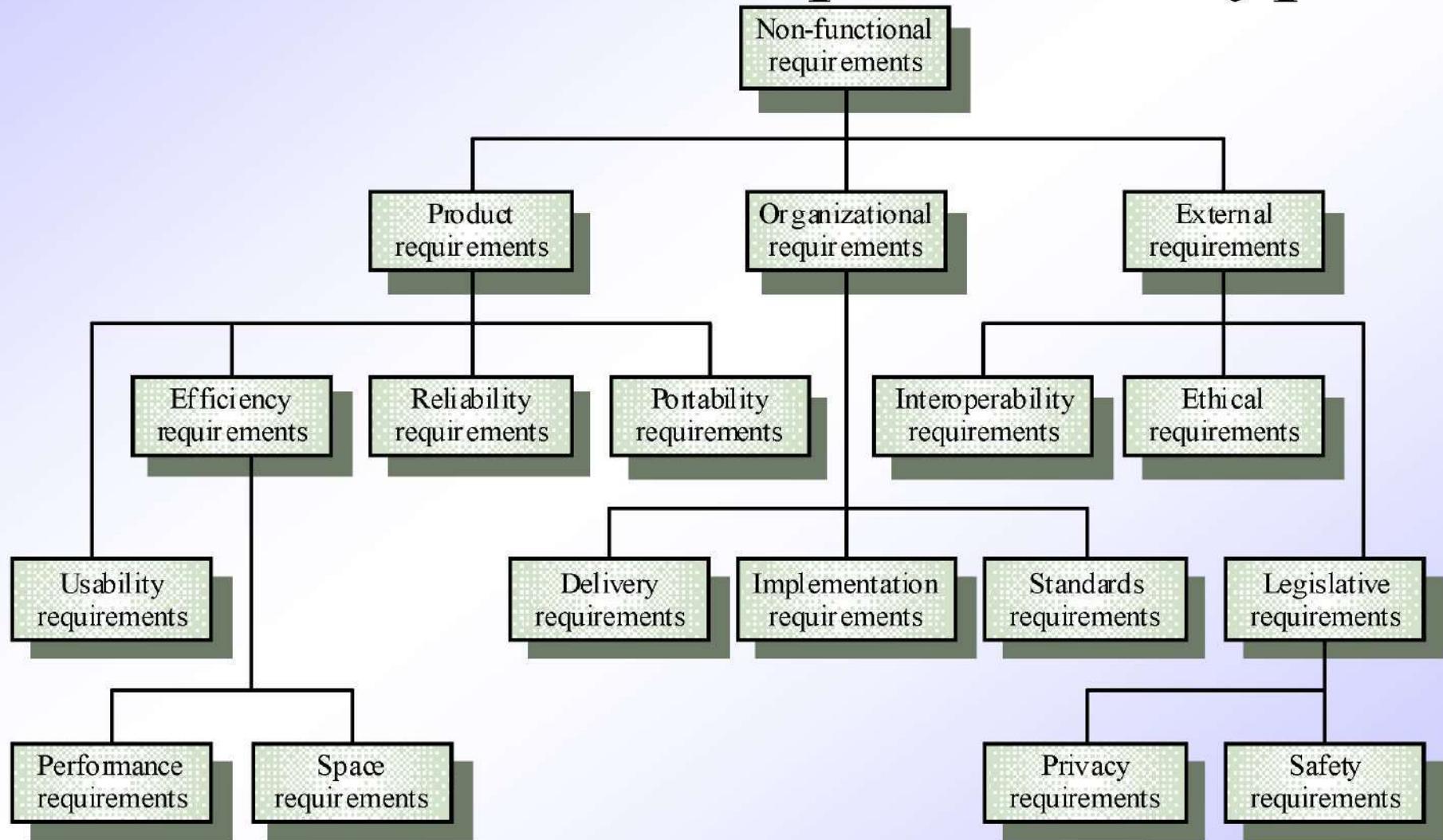
# Non-functional req.

- constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Describe functionality or system services
  - Eg. System is expected to run on a computer with less than 128 MB RAM.
  - Eg. You have to conform to CMM level 4.
  - Eg. System shall not disclose any personal information about customers apart from their name and reference number to the operators of the system

# Non-functional classifications

- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

# Non-functional requirement types



# Elements of Analysis Model

- Intent of analysis model is to provide description of required informational, functional and behavioural domains for a system
- Model changes dynamically
- Analysis model is snapshot of req at any time. It may change

- Different modes of representation force to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity

## A Set of Models



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



# Elements of the Analysis Model



## Object-oriented Analysis

### Scenario-based modeling

*Use case text*  
*Use case diagrams*  
Activity diagrams  
Swim lane diagrams

## Structured Analysis

### Flow-oriented modeling

Data structure diagrams  
Data flow diagrams  
Control-flow diagrams  
Processing narratives

### Class-based modeling

*Class diagrams*  
Analysis packages  
CRC models  
Collaboration diagrams

### Behavioral modeling

*State diagrams*  
Sequence diagrams



# Scenario-based

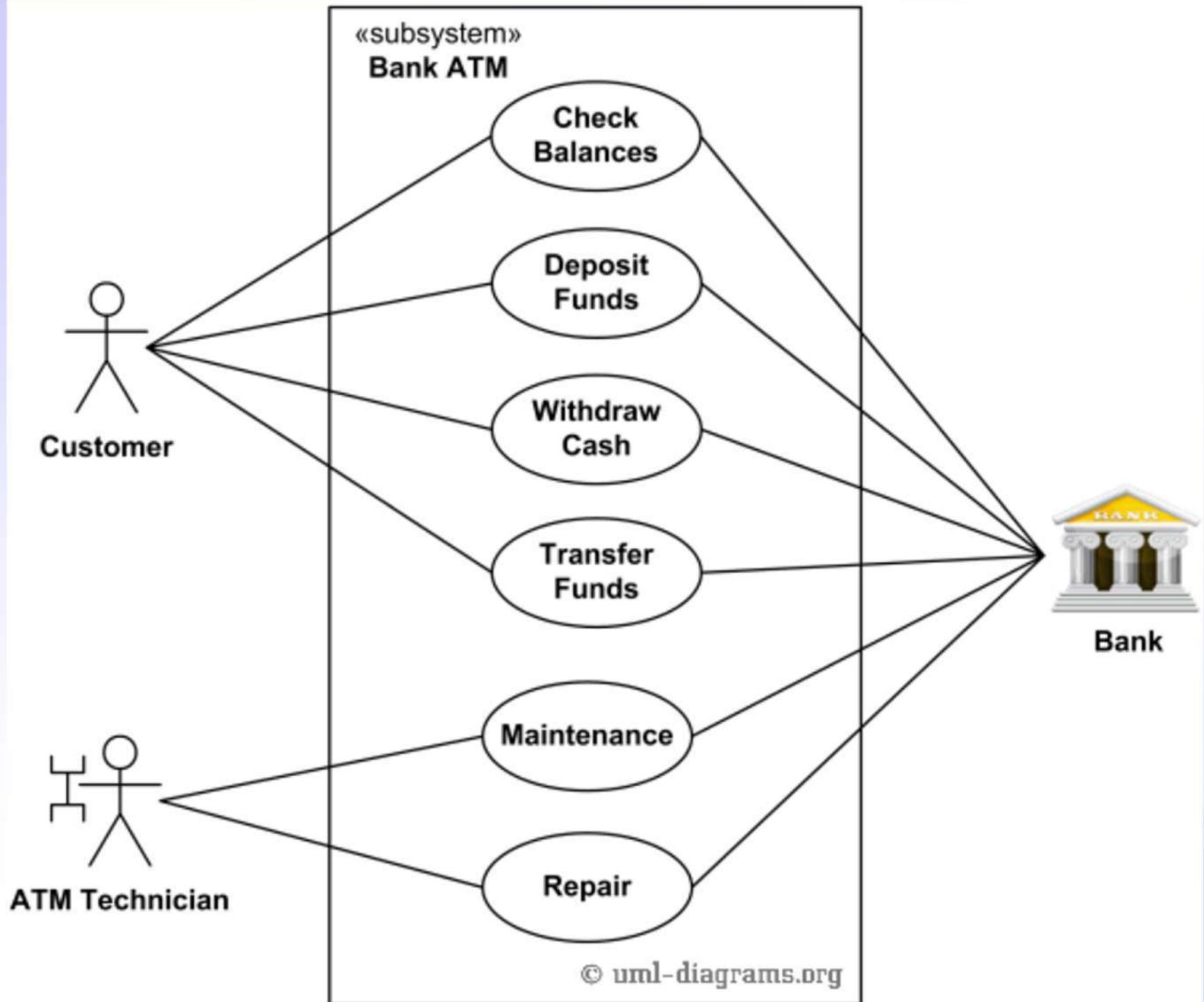
- User scenarios
  - describe how system will be used
  - How end user interacts with the system

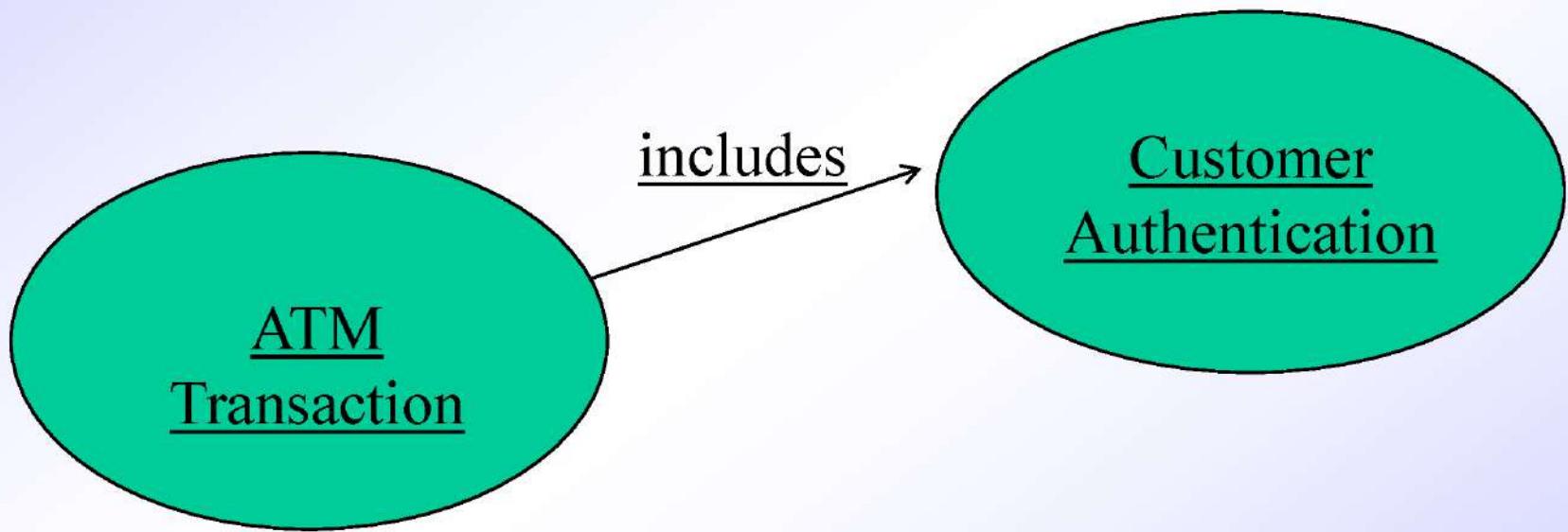
# Developing a Use case

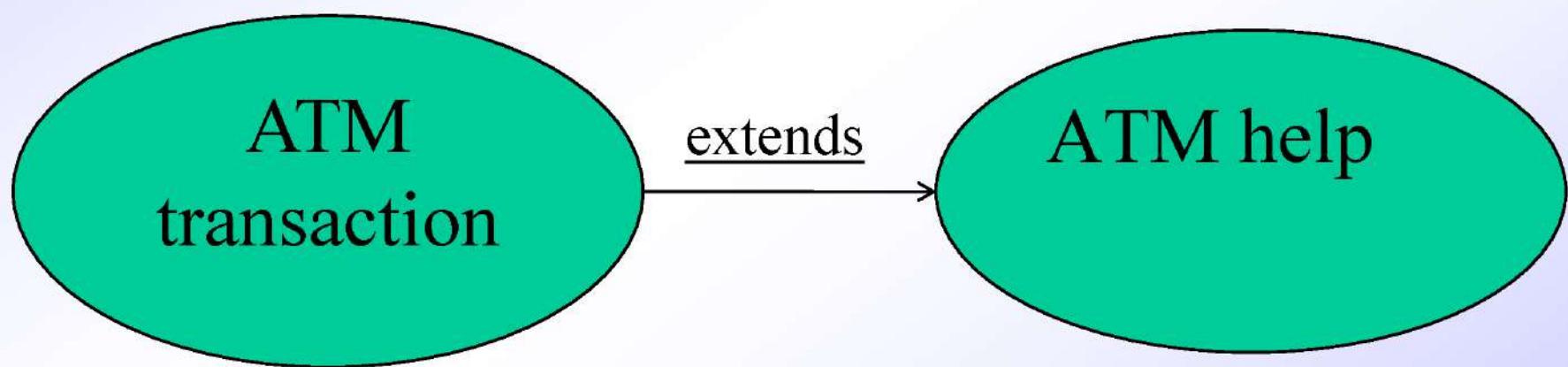
- Define set of actors
  - Actors is a role that people / devices play as they interact with the software.
  - Identify primary actors in first iteration
  - Secondary actors as more is learned about the system

## Questions to be answered by use-case

- Who is primary,secondary actors?
- Goals of actor
- Tasks performed by actor
- What system information will the actor acquire,produce or change?
- What information does actor desire from system







# Class Diagram

# Example Class Box

Class Name

Component

Attributes

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

Operations

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

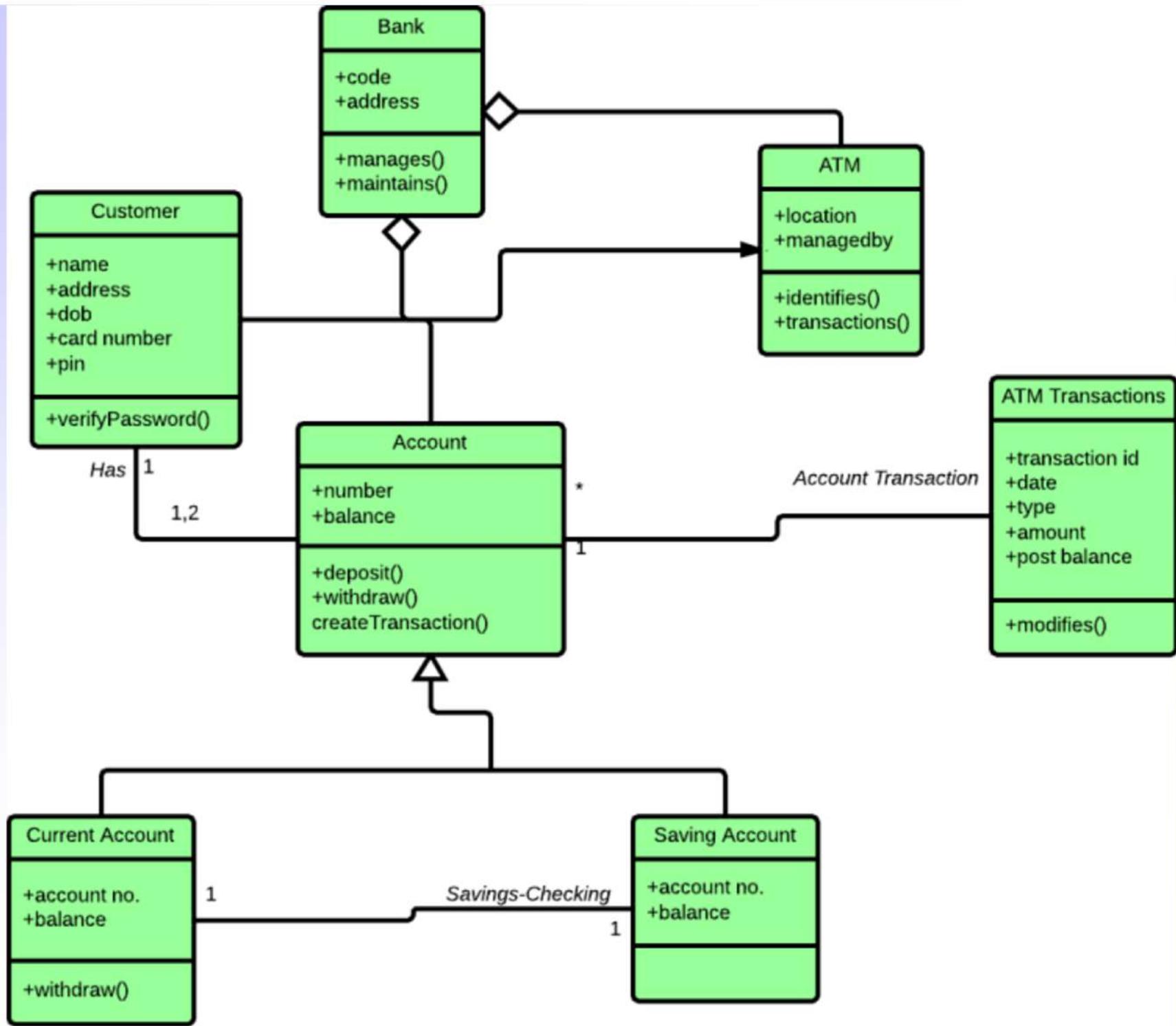


# Association, Generalization and Dependency (Ref: Fowler)



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



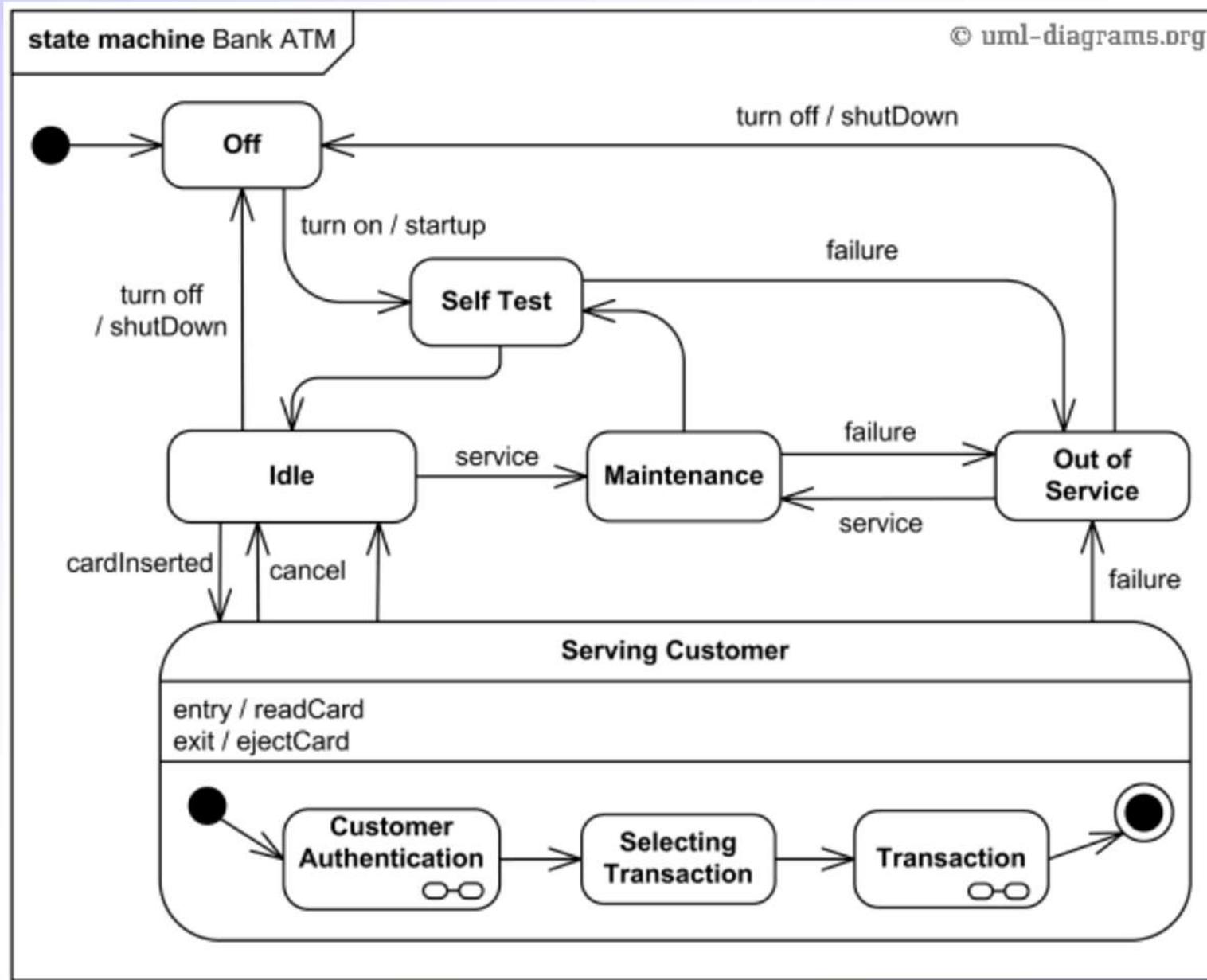


## Building a State Diagram



- A state is represented by a rounded rectangle
- A transition (i.e., event) is represented by a labeled arrow leading from one state to another

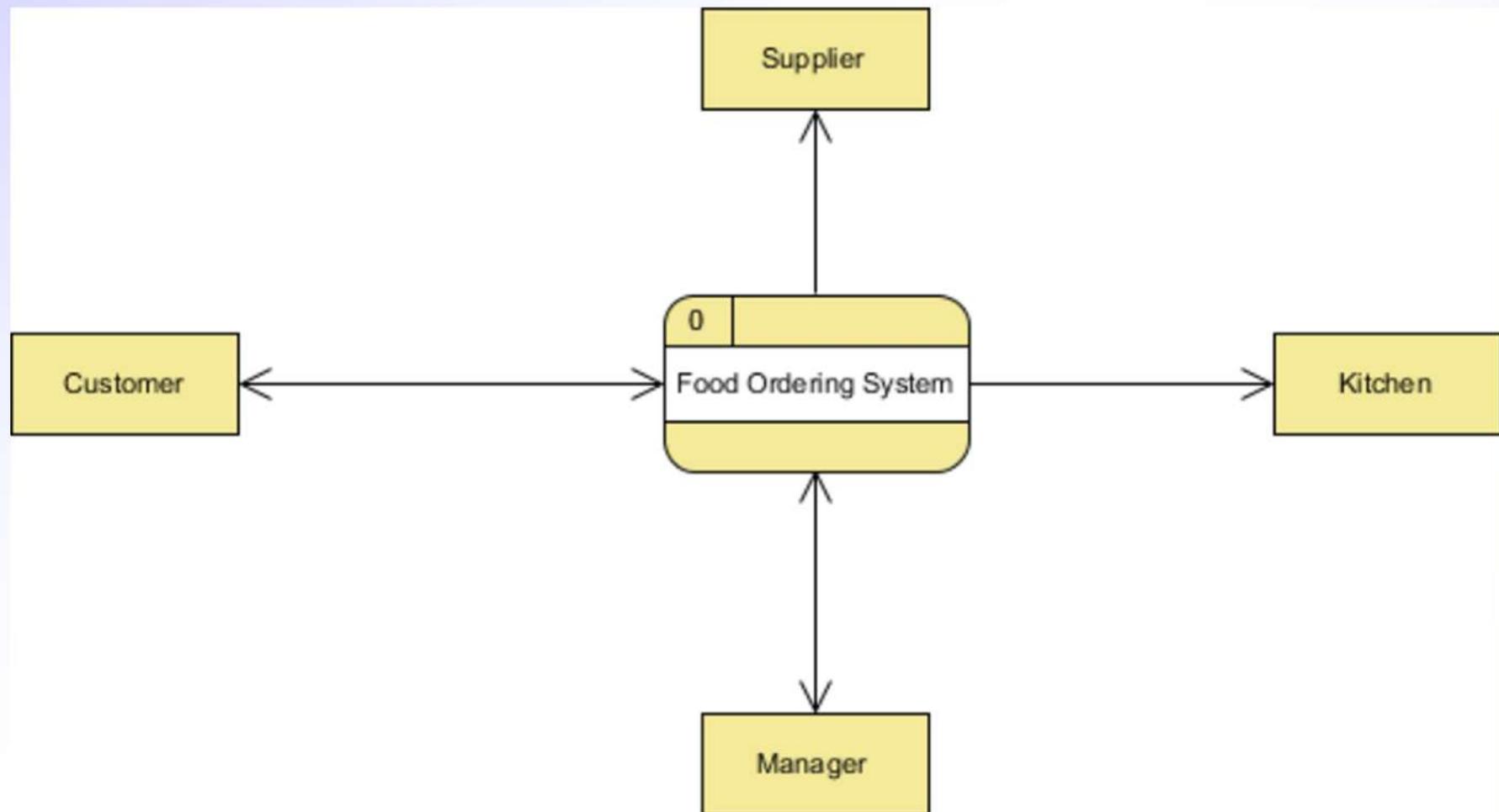
# State Diagram



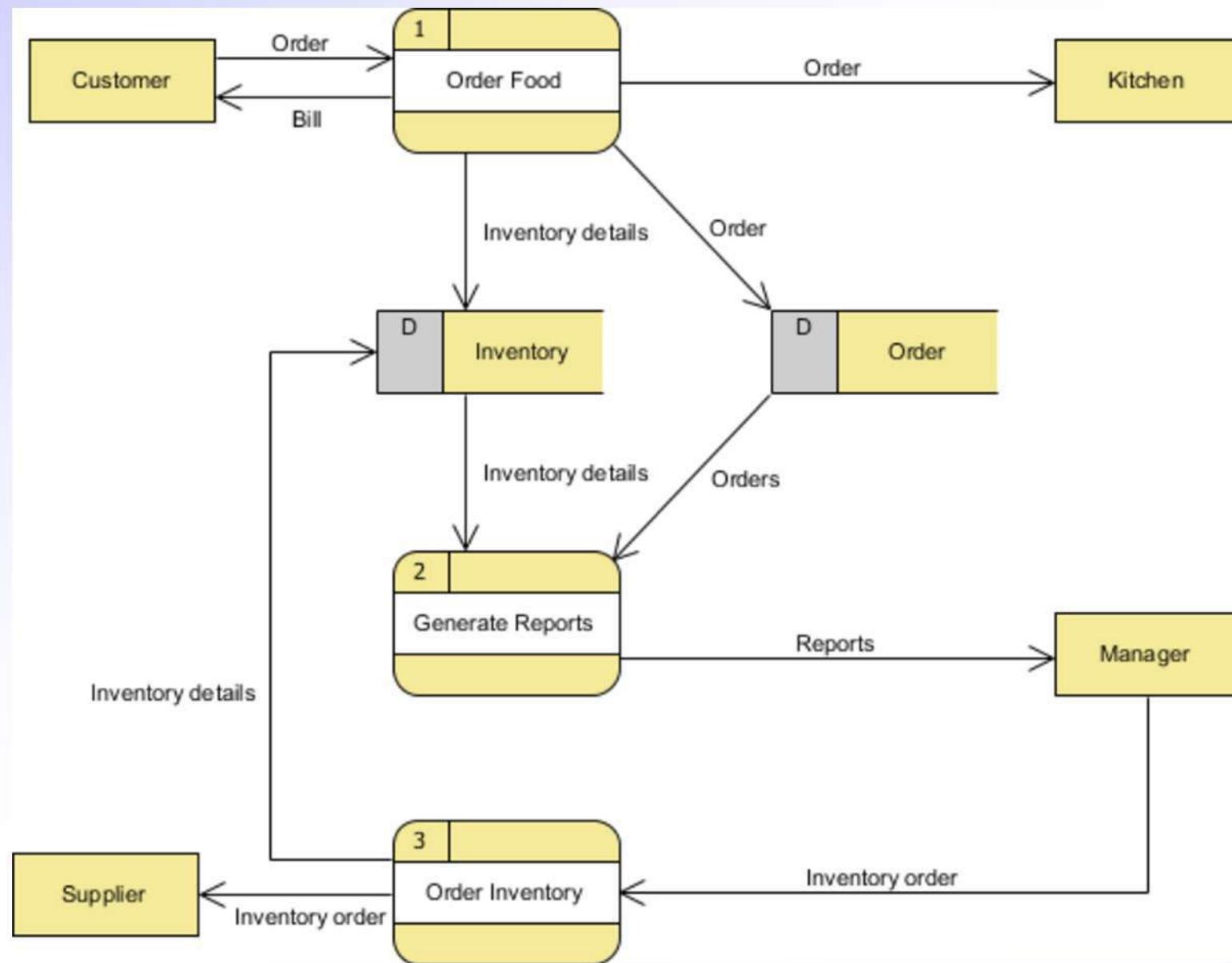
# Data Flow Diagram

- visual representation of the flow of information (i.e. data) within a system
- Level 0
  - Top Level
  - 1 process(entire system )and the external entities to which it interacts
  - data flow (connectors)

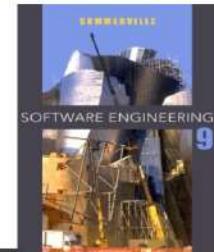
# Context DFD (DFD level 0)



# Level 1 DFD



- Process labels should be verb phrases; data stores are represented by nouns
- A data store must be associated to at least a process
- An external entity must be associated to at least a process
- Don't let it get too complex; normally 5 - 7 average people can manage processes
- DFD is non-deterministic - The numbering does not necessarily indicate sequence, it's useful in identifying the processes when discussing with users
- Data stores should not be connected to an external entity, otherwise, it would mean that you're giving an external entity direct access to your data files
- Data flows should not exist between 2 external entities without going through a process
- A process that has inputs but no outputs is considered to be a black-hole process



---

## Chapter 4 – Requirements Engineering

Lecture 13  
Dr Tripty Singh  
Ms. Sreevidya B



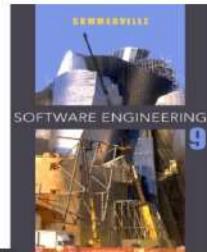
## Topics covered

---

- Functional and non-functional requirements
- The software requirements document
- Requirements specification
- Requirements engineering processes
- Requirements elicitation and analysis
- Requirements validation
- Requirements management

# Requirements Engineering

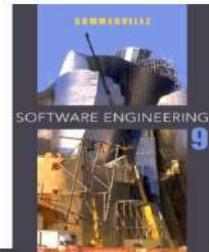
---



- The process of **establishing the services** that the customer requires from a system **and the constraints** under which it operates and is developed.
- The requirements themselves are the **descriptions of the system services and constraints** that are generated during the requirements engineering process.

# What is a Requirement?

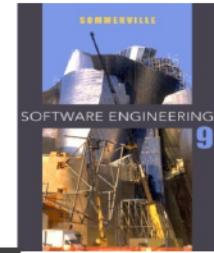
---



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

# Requirements Abstraction (Davis)

---

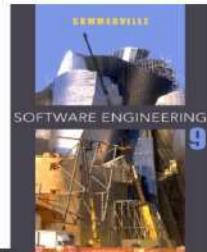


“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined.

The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs.

Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do.

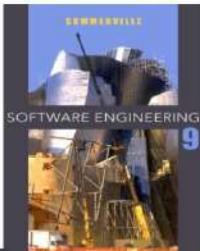
Both of these documents may be called the requirements document for the system.”



# Types of Requirements

---

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints.
  - **Written for customers.**
- System requirements
  - A **structured** document setting out **detailed descriptions** of the system's **functions, services and operational constraints**.
  - Defines what should be implemented so may be part of a contract between client and contractor.
  - **Whom do you think these are written for?**
  - **These are higher level than functional and non-functional requirements, which these may subsume.**



# User and System Requirements

## User requirement definition

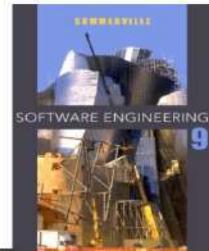
1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

## 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 13:30 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.

# Functional and Non-functional requirements

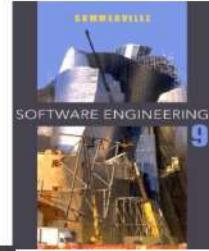
---



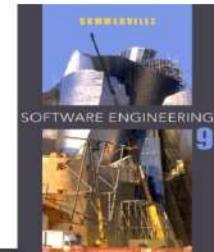
- Functional requirements
  - Statements of **services** the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system **should not do**.
- Non-functional requirements
  - **Constraints** on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - 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

---

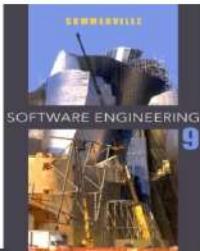


- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services 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.

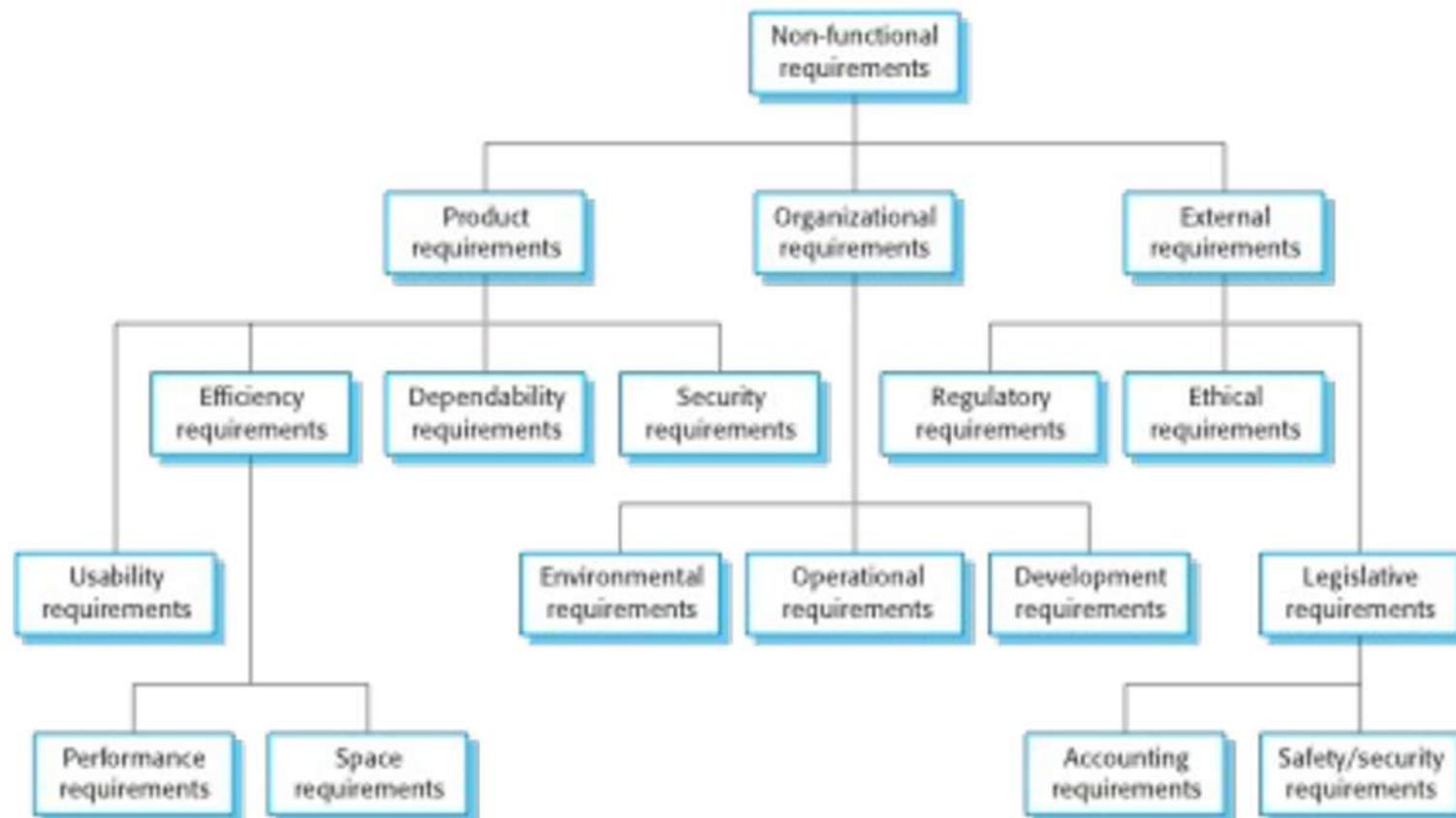


# Non-functional Requirements

- These define system properties and constraints e.g. reliability, response time, maintainability, 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.
- (Often internal to an organization or required for fit / compatibility with other comparable systems.)
- Non-functional requirements **may be more critical** than functional requirements. If these are not met, the system may be useless.

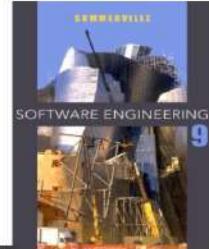


# Types of Nonfunctional Requirements



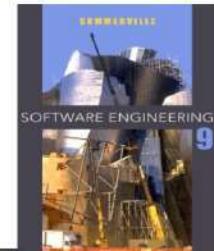
# Non-functional Requirements Implementation

---



- Non-functional requirements may affect the **overall architecture of a system** rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- A single non-functional requirement, such as a security requirement, may generate a **number** of related functional requirements that define system services that are required.
  - It may also generate requirements that **restrict** existing requirements.

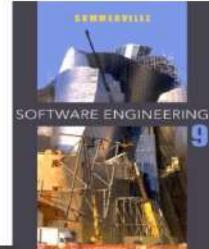
# Metrics for specifying nonfunctional requirements



Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	<b>Mean time to failure (MTTF)</b> Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure ( <b>MTTR</b> ) Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

# Domain requirements problems

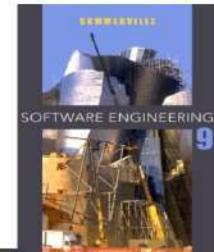
---



- Understandability
  - Requirements are expressed in the language of the application domain;
    - Application written for mortgage banking people need to express functionality in terms of home loans, mortgage balances, escrow, investor accounting, foreclosure, etc.
  - 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!!!**

## Key points

---



- **Requirements** for a software system set out what the system should do and define constraints on its operation and implementation.
- **Functional requirements** are statements of the **services** that the system must provide or are **descriptions of how some computations** must be carried out.
- **Non-functional requirements** often **constrain** the system being developed and the development process being used.
- They often relate to the emergent properties of the system and therefore apply to the system as a whole.

# Chapter 4, Requirements Elicitation



Dr Tripty Singh  
Srividhya B

# Requirements elicitation

- ◆ In requirements engineering, requirements elicitation is **the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders**. The practice is also sometimes referred to as "requirement gathering". The term elicitation is used in books and research to raise the fact that good requirements cannot just be collected from the customer, as would be indicated by the name requirements gathering. Requirements elicitation is non-trivial because you can never be sure you get all requirements from the user and customer by just asking them what the system should do or not do (for Safety and Reliability). Requirements elicitation practices include interviews, questionnaires, user observation, workshops, brainstorming, use cases, role playing and prototyping.
- ◆ Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. Requirements elicitation is a part of the requirements engineering process, usually followed by analysis and specification of the requirements.
- ◆ Commonly used elicitation processes are the stakeholder meetings or interviews.<sup>[21]</sup> For example, an important first meeting could be between software engineers and customers where they discuss their perspective of the requirements

## REQUIREMENT ELICITATION TECHNIQUES

BRAINSTORMING

DOCUMENT ANALYSIS

FOCUS GROUP

INTERFACE ANALYSIS

INTERVIEWS

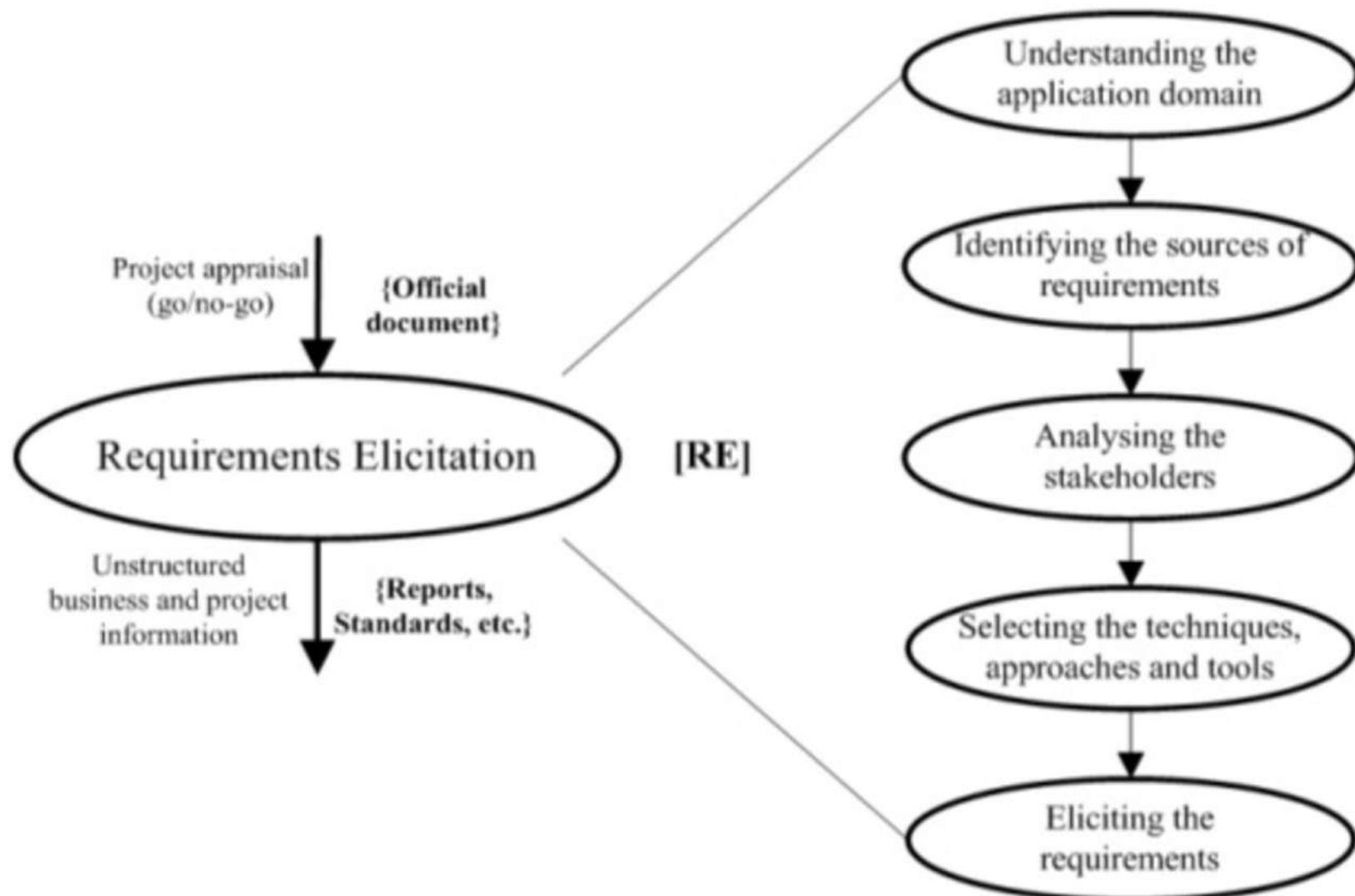
OBSERVATION

PROCESS MODELING

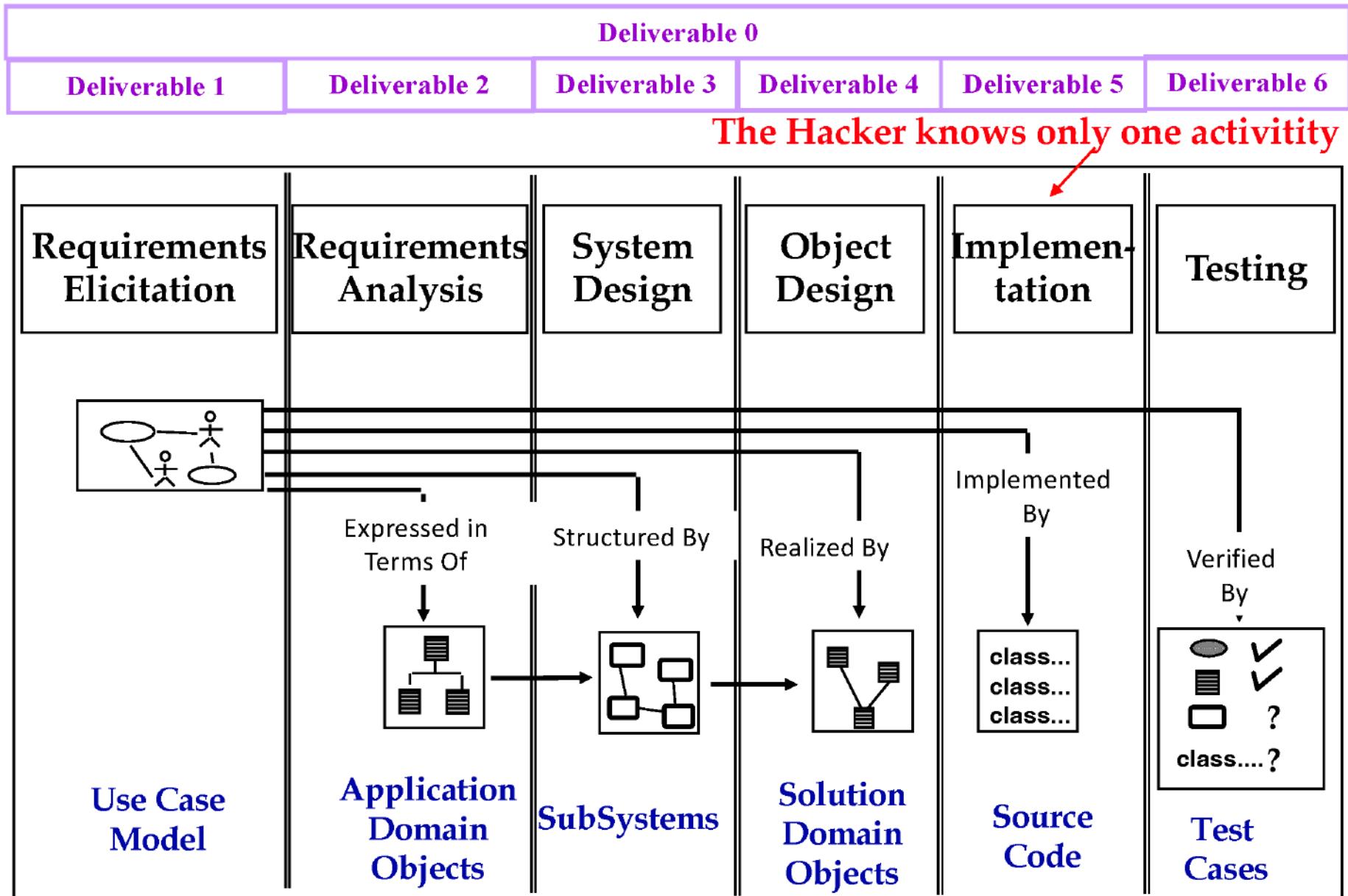
PROTOTYPE

REQUIREMENT WORKSHOPS

SURVEYS / QUESTIONNAIRE



# Software Lifecycle Activities



Each activity produces one or more models

# *First Step in Establishing the Requirements: System Identification*

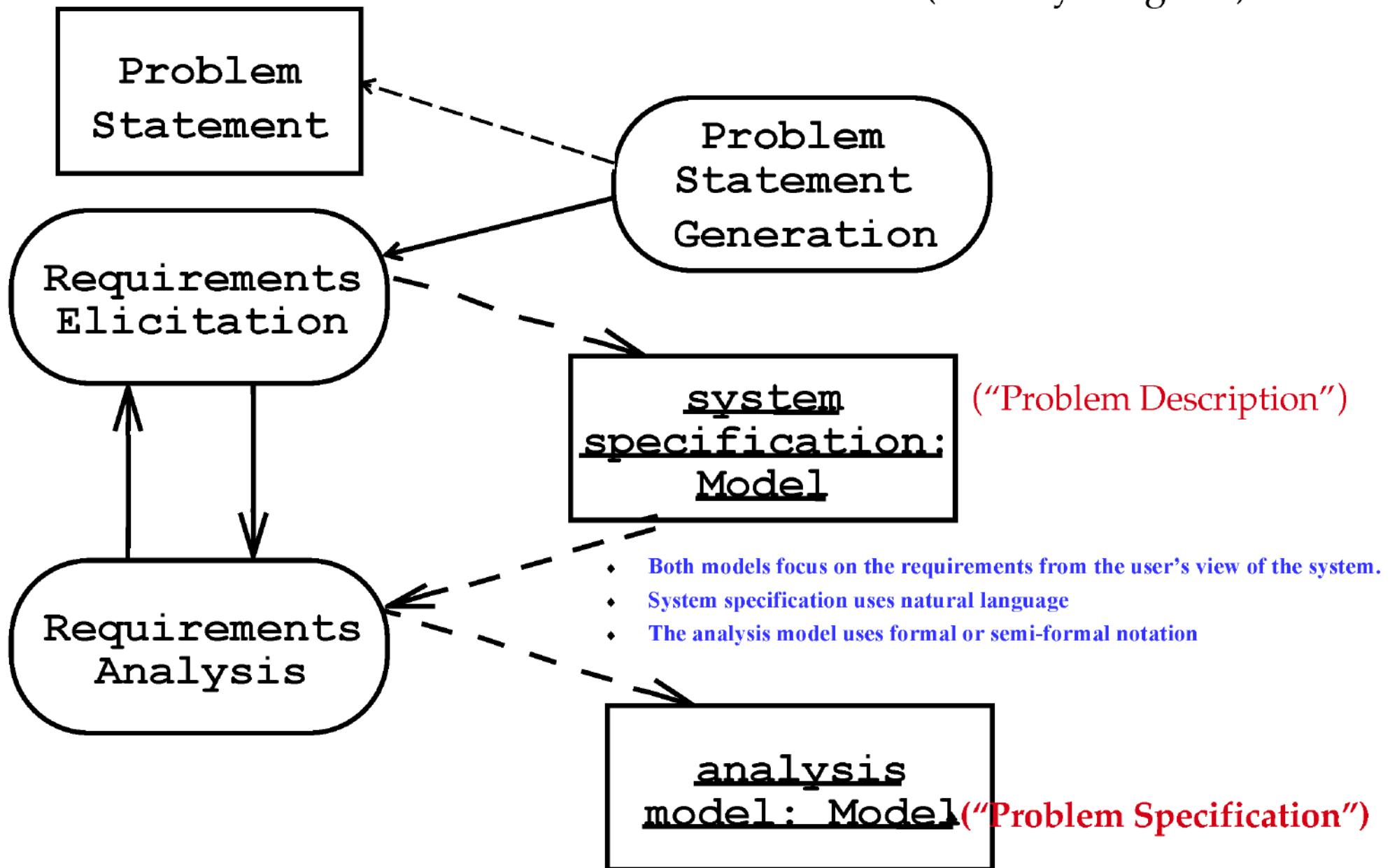
*In Brugge's methodology*

- ◆ The development of a system is not just done by taking a snapshot of a scene (domain)
- ◆ Two questions need to be answered:
  - ◆ How can we identify the *purpose* of a system?
  - ◆ Crucial is the definition of the *system boundary*: What is inside, what is outside the system?  
*Any example?*
- ◆ The requirements process consists of two activities:
  - ◆ Requirements *Elicitation*:
    - ◆ Definition of the system in terms understood by the *customer* (“Problem Description”)
  - ◆ Requirements *Analysis*:
    - ◆ *Technical* specification of the system in terms understood by the *developer* (“Problem Specification”)

*Any issues with this  
terminology?*

# *Products of Requirements Process*

(Activity Diagram)



An old school of thought mixing the domain model with the solution model,  
being design-oriented, and in a Waterfall fashion.

# *Requirements Elicitation*

- ◆ Very challenging activity
- ◆ Requires collaboration of people with different backgrounds
  - ◆ **Users with application domain knowledge**
  - ◆ **Developer with solution domain knowledge (design knowledge, implementation knowledge)**
- ◆ Bridging the gap between user and developer:
  - ◆ **Scenarios:** Example of the use of the system in terms of a series of interactions with between the user and the system
  - ◆ **Use cases:** Abstraction that describes a class of scenarios

## *Ingredients of a Problem Statement*

- ◆ Current situation: The Problem to be solved
- ◆ Description of one or more scenarios
- ◆ Requirements
  - ◆ **Functional and Nonfunctional requirements**
  - ◆ **Constraints (“pseudo requirements”)**
- ◆ Project Schedule
  - ◆ **Major milestones that involve interaction with the client including deadline for delivery of the system**
- ◆ Target environment
  - ◆ **The environment in which the delivered system has to perform a specified set of system tests**
- ◆ Client Acceptance Criteria
  - ◆ **Criteria for the system tests**

# *Challenges for requirements elicitation*

- ◆ In 1992, Christel and Kang identified problems that indicate the challenges for requirements elicitation:<sup>[3]</sup>
  1. **'Problems of scope'.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical details that may confuse, rather than clarify, overall system objectives.
  2. **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be “**obvious**,” specify requirements that conflict with the needs of other customers/users, or specific requirements that are ambiguous or untestable.
  3. **Problems of volatility.** The requirements change over time. The rate of change is sometimes referred to as the level of requirement volatility

# Requirements quality can be improved through these approaches:

1. **Visualization.** Using tools that promote better understanding of the desired end-product such as visualization and simulation.
2. **Consistent language.** Using simple, consistent definitions for requirements described in natural language and use the business terminology that is prevalent in the enterprise.
3. **Guidelines.** Following organizational guidelines that describe the collection techniques and the types of requirements to be collected. These guidelines are then used consistently across projects.
4. **Consistent use of templates.** Producing a consistent set of models and templates to document the requirements.
5. **Documenting dependencies.** Documenting dependencies and interrelationships among requirements.
6. **Analysis of changes.** Performing root cause analysis of changes to requirements and making corrective actions

# **Requirements Validation**

- ◆ Requirements validation is a critical step in the development process, usually after requirements engineering or requirements analysis. Also at delivery (client acceptance test).
- ◆ **Requirements validation criteria:**
  - ◆ **Correctness:**
    - ◆ The requirements represent the client's view.
  - ◆ **Completeness:**
    - ◆ All possible scenarios, in which the system can be used, are described, including exceptional behavior by the user or the system
  - ◆ **Consistency:**
    - ◆ There are functional or nonfunctional requirements that contradict each other
  - ◆ **Realism:**
    - ◆ Requirements can be implemented and delivered
  - ◆ **Traceability:**
    - ◆ Each system function can be traced to a corresponding set of functional requirements

**Anything**

RequisitePro from Rational  
*else?*

<http://www.rational.com/products/reqpro/docs/datasheet.html>

# Scenarios

- ◆ “A narrative description of what people do and experience as they try to make use of computer systems and applications” [M. Carroll, Scenario-based Design, Wiley, 1995]
- ◆ A concrete, focused, informal description of a single feature of the system used by a single actor.
- ◆ Scenarios can have many different uses during the software lifecycle
  - ◆ *Requirements Elicitation:* **As-is scenario, visionary scenario**
  - ◆ *Client Acceptance Test:* **Evaluation scenario**
  - ◆ *System Deployment:* **Training scenario.**

## *Types of Scenarios*

- ◆ **As-is scenario:**
  - ◆ Used in describing a current situation. Usually used in re-engineering projects. The user describes the system.
- ◆ **Visionary scenario:**
  - ◆ Used to describe a future system. Usually used in greenfield engineering and reengineering projects.
  - ◆ Can often not be done by the user or developer alone
- ◆ **Evaluation scenario:**
  - ◆ User tasks against which the system is to be evaluated.
    - ◆ .
- ◆ **Training scenario:**
  - ◆ Step by step instructions that guide a novice user through a system

## *How do we find scenarios?*

- ◆ Don't expect the client to be verbal if the system does not exist (greenfield engineering)
- ◆ Don't wait for information even if the system exists
- ◆ Engage in a *dialectic* approach (evolutionary, incremental engineering)
  - ◆ You help the client to formulate the requirements
  - ◆ The client helps you to understand the requirements
  - ◆ The requirements evolve while the scenarios are being developed

*Where did we see this word  
"dialectic"?*

## *Heuristics for finding Scenarios*

- ◆ Ask yourself or the client the following questions:
  - ◆ What are the primary **tasks** that the system needs to perform?
  - ◆ What **data** will the actor create, store, change, remove or add in the system?
  - ◆ What **external changes** does the system need to know about?
  - ◆ What changes or events will the actor of the system need to be informed about?
- ◆ However, don't rely on *questionnaires* alone. **What's questionnaire?**
- ◆ Insist on *task observation* if the system already exists (interface engineering or reengineering)
  - ◆ Ask to speak to the end user, not just to the software contractor
  - ◆ Expect resistance and try to overcome it

## *Example: Accident Management System*

- ◆ What needs to be done to report a “Cat in a Tree” incident?
- ◆ What do you need to do if a person reports “Warehouse on Fire?”
- ◆ Who is involved in reporting an incident?
- ◆ What does the system do, if no police cars are available? If the police car has an accident on the way to the “cat in a tree” incident?
- ◆ What do you need to do if the “Cat in the Tree” turns into a “Grandma has fallen from the Ladder”?
- ◆ Can the system cope with a simultaneous incident report “Warehouse on Fire?”

## *Scenario Example: Warehouse on Fire*

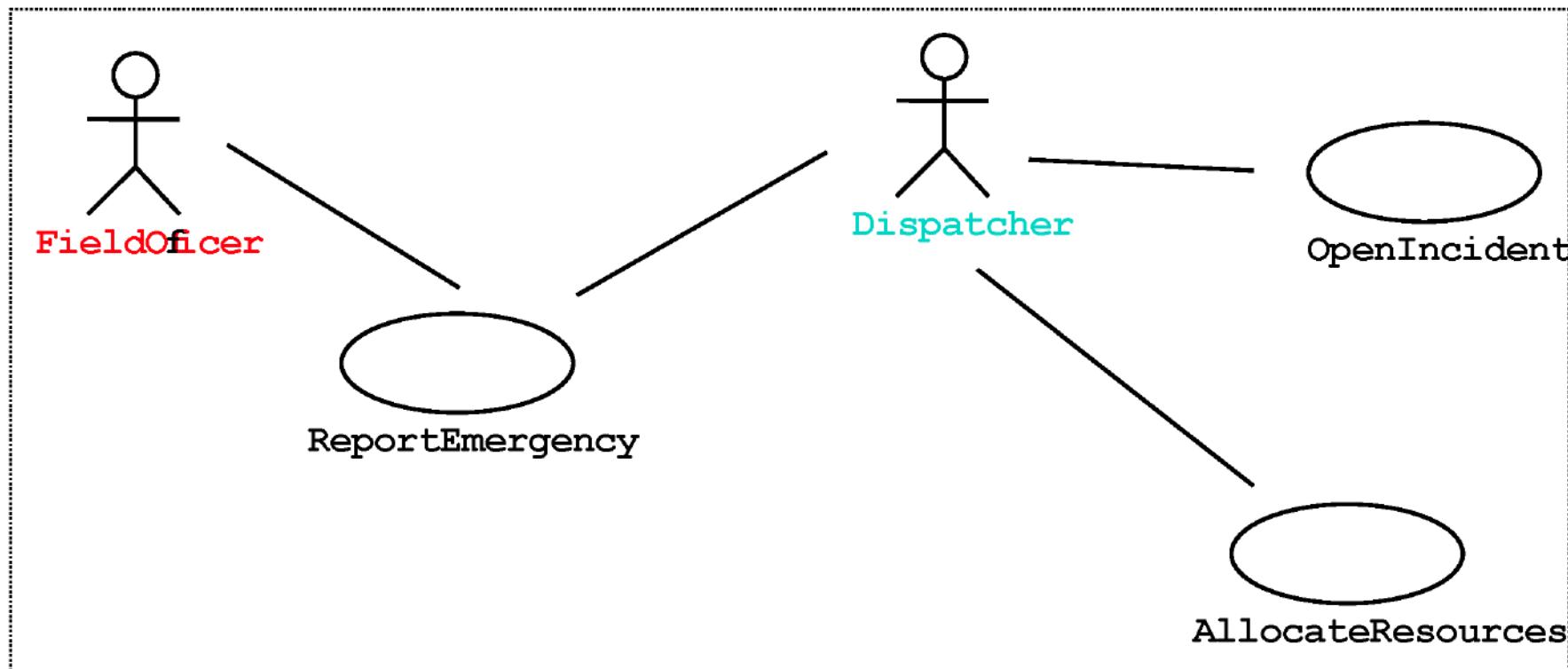
- ◆ Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, **reports the emergency** from her car.
- ◆ Alice enters the address of the building, a brief description of its location (i.e., north west corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appear to be relatively busy. She confirms her input and waits for an acknowledgment.
- ◆ John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and sends their estimated arrival time (ETA) to Alice.
- ◆ Alice received the acknowledgment and the ETA.

## *Observations about Warehouse on Fire Scenario*

- ◆ Concrete scenario
  - ◆ **Describes a single instance of reporting a fire incident.**
  - ◆ **Does not describe all possible situations in which a fire can be reported.**
- ◆ Participating actors
  - ◆ **Bob, Alice and John**

# *Example: Use Case Model for Incident Management*

*How do we go from scenarios to use cases?*



# *Use Case Example: ReportEmergency*

- ◆ Use case name: ReportEmergency
- ◆ Participating Actors:
  - ◆ **Field Officer (Bob and Alice in the Scenario)**
  - ◆ **Dispatcher (John in the Scenario)**
- ◆ Exceptions:
  - ◆ **The FieldOfficer is notified immediately if the connection between her terminal and the central is lost.**
  - ◆ **The Dispatcher is notified immediately if the connection between any logged in FieldOfficer and the central is lost.**
- ◆ Flow of Events: **on next slide.**
- ◆ Special Requirements:
  - ◆ **The FieldOfficer's report is acknowledged within 30 seconds. The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.**

Name of Use Case

Actors: Description of Actors involved in use case)

Entry condition: "This use case starts when..."

Flow of Events: Free form, informal natural language

Exit condition: "This use cases terminates when..."

Exceptions: Describe what happens if things go wrong

Special Requirements: NFR, Constraints

## *Use Case Example: ReportEmergency Flow of Events*

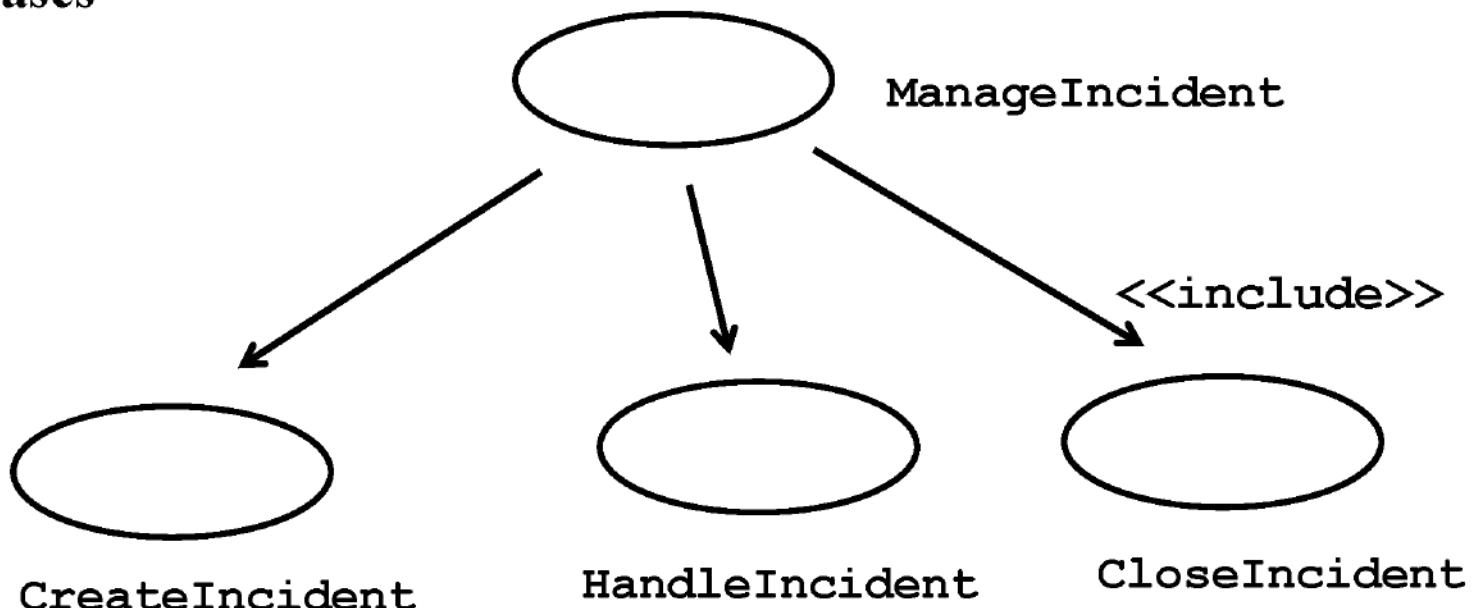
- ◆ The **FieldOfficer** activates the “Report Emergency” function of her terminal. FRIEND responds by presenting a form to the officer.
- ◆ The **FieldOfficer** fills the form, by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form, at which point, the **Dispatcher** is notified.
- ◆ The **Dispatcher** reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the emergency report.
- ◆ The **FieldOfficer** receives the acknowledgment and the selected response.

## *Use Case Associations*

- ◆ A use case model consists of use cases and use case associations
  - ◆ **A use case association is a relationship between use cases**
- ◆ Important types of use case associations: Include, Extends, Generalization
  - ◆ **Include**
    - ◆ **A use case uses another use case (“functional decomposition”)**
  - ◆ **Extends**
    - ◆ **A use case extends another use case**
  - ◆ **Generalization**
    - **An abstract use case has different specializations**

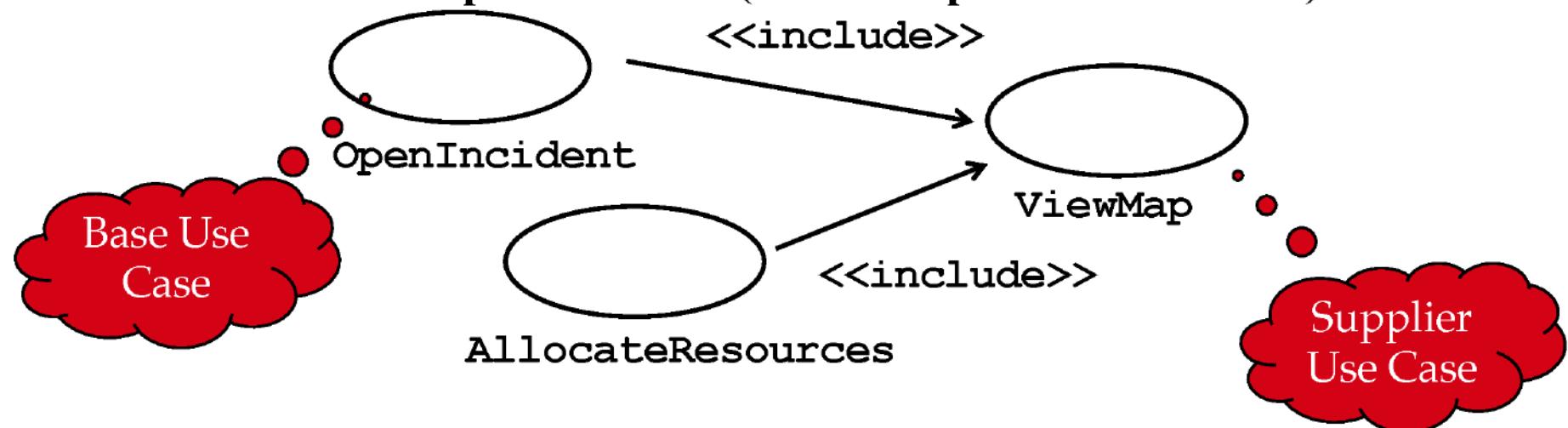
## *<<Include>>: Functional Decomposition*

- ◆ Problem:
  - ◆ A function in the original problem statement is too complex to be solvable immediately
- ◆ Solution:
  - ◆ Describe the function as the aggregation of a set of simpler functions. The associated use case is decomposed into smaller use cases



## *<<Include>>: Reuse of Existing Functionality*

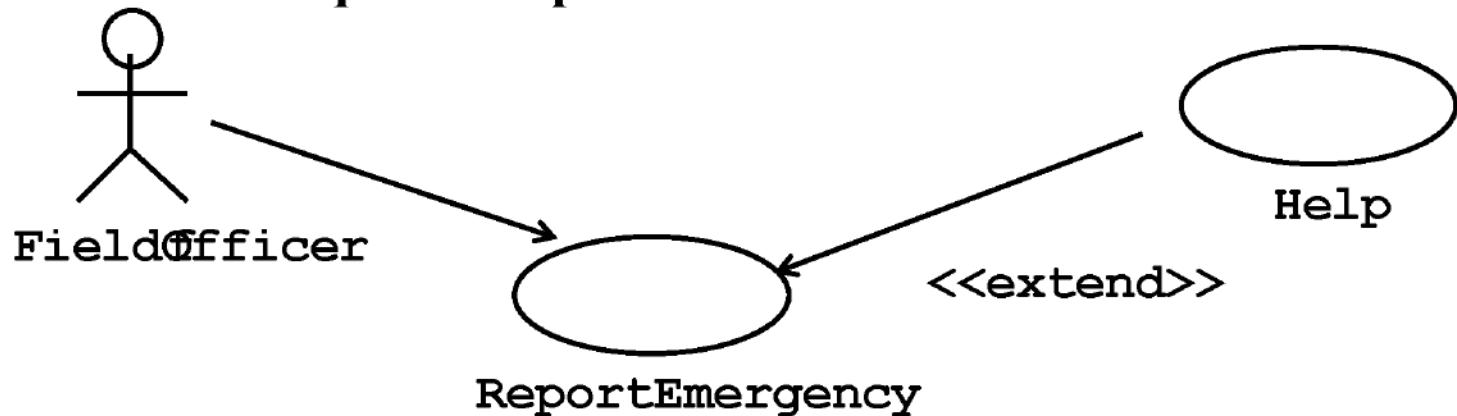
- ◆ Problem:
  - ◆ There are already existing functions. How can we *reuse* them?
- ◆ Solution:
  - ◆ The *include association* from a use case A to a use case B indicates that an instance of the use case A performs all the behavior described in the use case B (“A delegates to B”)
- ◆ Example:
  - ◆ The use case “ViewMap” describes behavior that can be used by the use case “OpenIncident” (“ViewMap” is factored out)



**Note: The base case cannot exist alone. It is always called with the supplier use case**

## *<Extend>> Association for Use Cases*

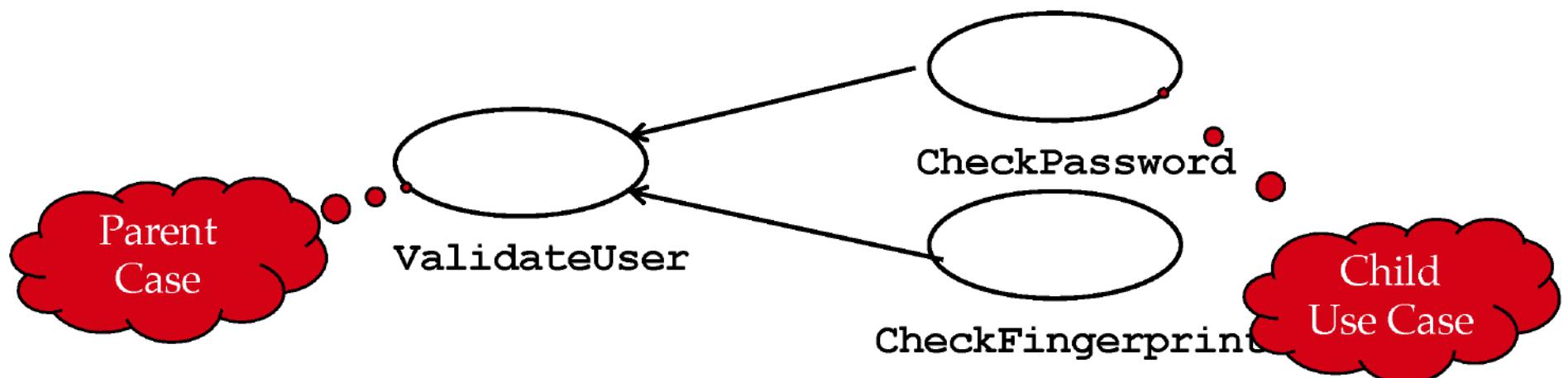
- ◆ Problem:
  - ◆ The functionality in the original problem statement needs to be extended.
- ◆ Solution:
  - ◆ An *extend association* from a use case A to a use case B indicates that use case B is an extension of use case A.
- ◆ Example:
  - ◆ The use case “ReportEmergency” is complete by itself , but can be extended by the use case “Help” for a specific scenario in which the user requires help



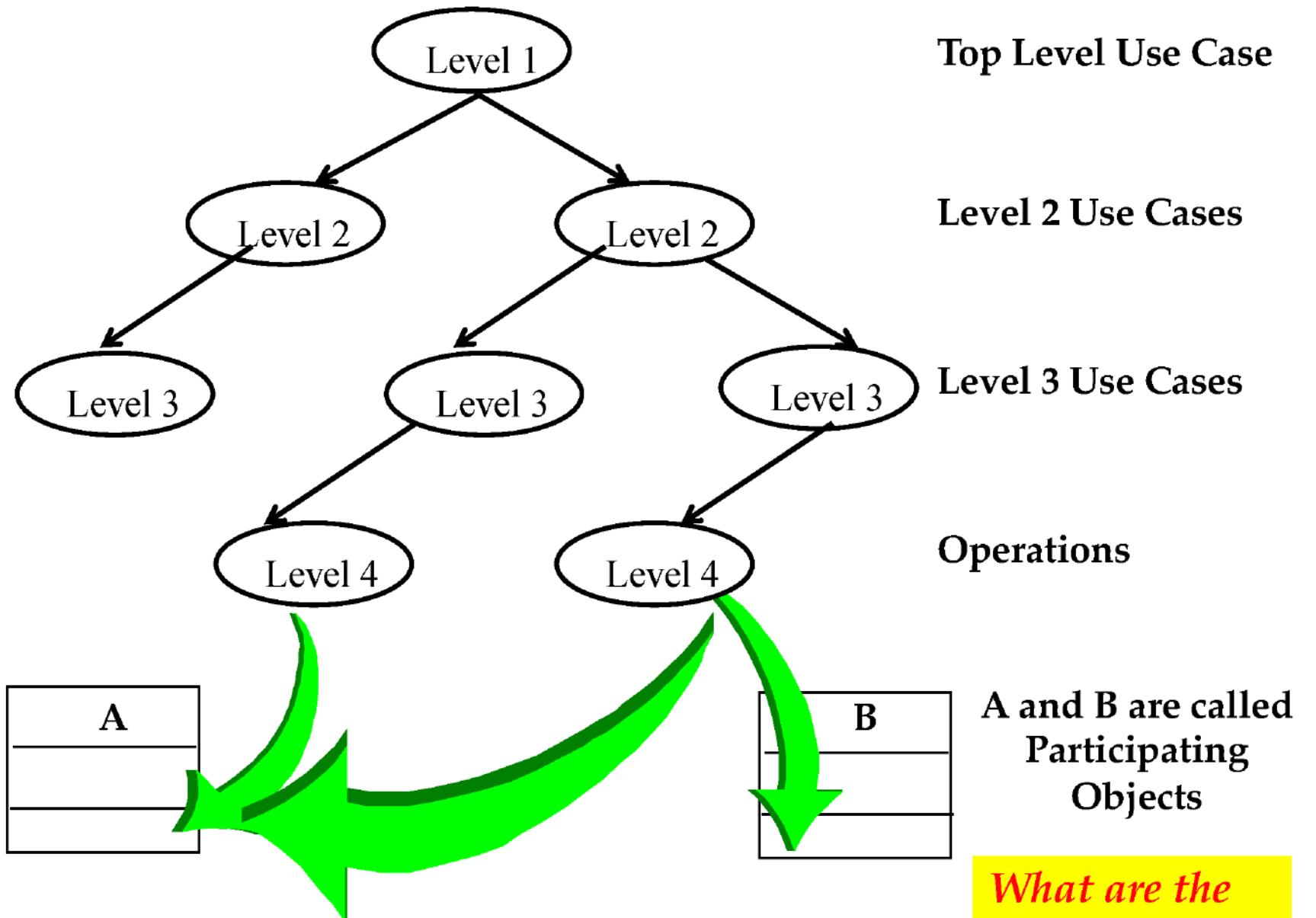
**Note: The base use case can be executed without the use case extension in extend associations.**

## *Generalization association in use cases*

- ◆ Problem:
  - ◆ You have common behavior among use cases and want to factor this out.
- ◆ Solution:
  - ◆ The generalization association among use cases factors out common behavior. The child use cases inherit the behavior and meaning of the parent use case and add or override some behavior.
- ◆ Example:
  - ◆ Consider the use case “ValidateUser”, responsible for verifying the identity of the user. The customer might require two realizations: “CheckPassword” and “CheckFingerprint”



## *From Use Cases to Objects*



# **Requirements Analysis Document Template**

1. Introduction
2. Current system
3. Proposed system
  - 3.1 Overview
  - 3.2 Functional requirements
  - 3.3 Nonfunctional requirements
  - 3.4 Constraints (“Pseudo requirements”)
  - 3.5 System models
    - 3.5.1 Scenarios
    - 3.5.2 Use case model
    - 3.5.3 Object model**
      - 3.5.3.1 Data dictionary**
      - 3.5.3.2 Class diagrams**
    - 3.5.4 Dynamic models**
    - 3.5.5 User interface**
  4. Glossary

# *Summary*

- ◆ The requirements process consists of requirements elicitation and analysis.
- ◆ The requirements elicitation activity is different for:
  - ◆ **Greenfield Engineering, Reengineering, Interface Engineering**
- ◆ Scenarios:
  - ◆ **Great way to establish communication with client**
  - ◆ **Different types of scenarios: As-Is, visionary, evaluation and training**
  - ◆ **Use cases: Abstraction of scenarios**
- ◆ Pure functional decomposition is bad:
  - ◆ **Leads to unmaintainable code**
- ◆ Pure object identification is bad:
  - ◆ **May lead to wrong objects, wrong attributes, wrong methods**
- ◆ The key to successful analysis:
  - ◆ **Start with use cases and then find the participating objects**
  - ◆ **If somebody asks “What is this?”, do not answer right away. Return the question or observe the end user: “What is it used for?”**

## *Appendix: Additional Slides*

# *Where are we right now?*

- ♦ Three ways to deal with complexity:
  - ♦ Abstraction
  - ♦ Decomposition (Technique: Divide and conquer)
  - ♦ Hierarchy (Technique: Layering)
- ♦ Two ways to deal with decomposition:
  - ♦ Object-orientation and functional decomposition
  - ♦ Functional decomposition leads to unmaintainable code
  - ♦ Depending on the purpose of the system, different objects can be found  
*Brugge's*
- ♦ What is the right way?
  - ♦ Start with a description of the functionality (Use case model). Then proceed by finding objects (object model).
- ♦ What activities and models are needed?
  - ♦ This leads us to the software lifecycle we use in this class

# *Software Lifecycle Definition*

- ◆ Software lifecycle:
  - ◆ Set of activities and their relationships to each other to support the development of a software system
- ◆ Typical Lifecycle questions:
  - ◆ Which activities should I select for the software project?
  - ◆ What are the dependencies between activities?
  - ◆ How should I schedule the activities?
  - ◆ What is the result of an activity

## *Problem Statement*

- ◆ The problem statement is developed by the client as a description of the problem addressed by the system
- ◆ Other words for problem statement:
  - ◆ **Statement of Work**
- ◆ A good problem statement describes
  - ◆ **The current situation**
  - ◆ **The functionality the new system should support**
  - ◆ **The environment in which the system will be deployed**
  - ◆ **Deliverables expected by the client**
  - ◆ **Delivery dates**
  - ◆ **A set of acceptance criteria**

## *What is usually not in the requirements?*

- ◆ System structure, implementation technology
  - ◆ Development methodology
  - ◆ Development environment
  - ◆ Implementation language
  - ◆ Reusability
- 
- ◆ It is desirable that none of these above are constrained by the client. Fight for it!

## *ARENA: The Problem*

- ◆ The Internet has enabled virtual communities
  - ◆ Groups of people sharing common interests but who have never met each other in person. Such virtual communities can be short lived (e.g. people in a chat room or playing a multi player game) or long lived (e.g., subscribers to a mailing list).
- ◆ Many multi-player computer games now include support for virtual communities.
  - ◆ Players can receive news about game upgrades, new game levels, announce and organize matches, and compare scores.
- ◆ Currently each game company develops such community support in each individual game.
  - ◆ Each company uses a different infrastructure, different concepts, and provides different levels of support.
- ◆ This redundancy and inconsistency leads to problems:
  - ◆ High learning curve for players joining a new community,
  - ◆ Game companies need to develop the support from scratch
  - ◆ Advertisers need to contact each individual community separately.

## *ARENA: The Objectives*

- ◆ Provide a generic infrastructure for operating an arena to
  - ◆ Support virtual game communities.
  - ◆ Register new games
  - ◆ Register new players
  - ◆ Organize tournaments
  - ◆ Keeping track of the players scores.
- ◆ Provide a framework for tournament organizers
  - ◆ to customize the number and sequence of matchers and the accumulation of expert rating points.
- ◆ Provide a framework for game developers
  - ◆ for developing new games, or for adapting existing games into the ARENA framework.
- ◆ Provide an infrastructure for advertisers.

# *Types of Requirements*

- ◆ Functional requirements:
  - ◆ **Describe the interactions between the system and its environment independent from implementation**
  - ◆ **Examples:**
    - ◆ An ARENA operator should be able to define a new game.
- ◆ Nonfunctional requirements:
  - ◆ **User visible aspects of the system not directly related to functional behavior.**
  - ◆ **Examples:**
    - ◆ The response time must be less than 1 second
    - ◆ The ARENA server must be available 24 hours a day
- ◆ Constraints (“Pseudo requirements”):
  - ◆ **Imposed by the client or the environment in which the system operates**
    - ◆ The implementation language must be Java
    - ◆ ARENA must be able to dynamically interface to existing games provided by other game developers.

# *Types of Requirements Elicitation*

- ◆ Greenfield Engineering
  - ◆ Development starts from scratch, no prior system exists, the requirements are extracted from the end users and the client
  - ◆ Triggered by user needs
  - ◆ Example: Develop a game from scratch: Asteroids
- ◆ Re-engineering
  - ◆ Re-design and/or re-implementation of an existing system using newer technology
  - ◆ Triggered by technology enabler
  - ◆ Example: Reengineering an existing game
- ◆ Interface Engineering
  - ◆ Provide the services of an existing system in a new environment
  - ◆ Triggered by technology enabler or new market needs
  - ◆ Example: Interface to an existing game (Bumpers)

## *Current Situation: The Problem To Be Solved*

- ◆ There is a problem in the current situation
  - ◆ Examples:
    - ◆ The response time when playing letter-chess is far too slow.
    - ◆ I want to play Go, but cannot find players on my level.
- ◆ What has changed? Why can address the problem now?
  - ◆ There has been a change, either in the application domain or in the solution domain
  - ◆ *Change in the application domain*
    - ◆ A new function (business process) is introduced into the business
    - ◆ Example: We can play highly interactive games with remote people
  - ◆ *Change in the solution domain*
    - ◆ A new solution (technology enabler) has appeared
    - ◆ Example: The internet allows the creation of virtual communities.

## *Heuristics: How do I find use cases?*

- ◆ Select a narrow vertical slice of the system (i.e. one scenario)
  - ◆ **Discuss it in detail with the user to understand the user's preferred style of interaction**
- ◆ Select a horizontal slice (i.e. many scenarios) to define the scope of the system.
  - ◆ **Discuss the scope with the user**
- ◆ Use illustrative prototypes (mock-ups) as visual support
- ◆ Find out what the user does
  - ◆ **Task observation (Good)**
  - ◆ **Questionnaires (Bad)**

## *Next goal, after the scenarios are formulated:*

- ◆ Find all the use cases in the scenario that specifies all possible instances of how to report a fire
  - ◆ Example: “Report Emergency “ in the first paragraph of the scenario is a candidate for a use case
- ◆ Describe each of these use cases in more detail
  - ◆ Participating actors
  - ◆ Describe the Entry Condition
  - ◆ Describe the Flow of Events
  - ◆ Describe the Exit Condition
  - ◆ Describe Exceptions
  - ◆ Describe Special Requirements (Constraints, Nonfunctional Requirements)

# *Use Cases*

- ◆ A use case is a flow of events in the system, including interaction with actors
- ◆ It is initiated by an actor
- ◆ Each use case has a name
- ◆ Each use case has a termination condition
- ◆ Graphical Notation: An oval with the name of the use case



Name of Use Case

Actors: Description of Actors involved in use case)

Entry condition: "This use case starts when..."

Flow of Events: Free form, informal natural language

Exit condition: "This use cases terminates when..."

Exceptions: Describe what happens if things go wrong

Special Requirements: NFR, Constraints

*Use Case Model: The set of all use cases specifying the complete functionality of the system*

## *Another Use Case Example: Allocate a Resource*

- ◆ Actors:

- ◆ *Field Supervisor:* This is the official at the emergency site....
- ◆ *Resource Allocator:* The Resource Allocator is responsible for the commitment and decommitment of the Resources managed by the FRIEND system. ...
- ◆ *Dispatcher:* A Dispatcher enters, updates, and removes Emergency Incidents, Actions, and Requests in the system. The Dispatcher also closes Emergency Incidents.
- ◆ *Field Officer:* Reports accidents from the Field

## *Another Use Case Example: Allocate a Resource*

- ◆ *Use case name:* AllocateResources
- ◆ *Participating Actors:*
  - ◆ **Field Officer (Bob and Alice in the Scenario)**
  - ◆ **Dispatcher (John in the Scenario)**
  - ◆ **Resource Allocator**
  - ◆ **Field Supervisor**
- ◆ *Entry Condition*
  - ◆ **The Resource Allocator has selected an available resource.**
  - ◆ **The resource is currently not allocated**
- ◆ *Flow of Events*
  - ◆ **The Resource Allocator selects an Emergency Incident.**
  - ◆ **The Resource is committed to the Emergency Incident.**
- ◆ *Exit Condition*
  - ◆ **The use case terminates when the resource is committed.**
  - ◆ **The selected Resource is now unavailable to any other Emergency Incidents or Resource Requests.**
- ◆ *Special Requirements*
  - ◆ **The Field Supervisor is responsible for managing the Resources**

## *Order of steps when formulating use cases*

- ◆ First step: name the use case
  - ◆ Use case name: ReportEmergency
- ◆ Second step: Find the actors
  - ◆ Generalize the concrete names (“Bob”) to participating actors (“Field officer”)
  - ◆ Participating Actors:
    - ◆ Field Officer (Bob and Alice in the Scenario)
    - ◆ Dispatcher (John in the Scenario)
- ◆ Third step: Then concentrate on the flow of events
  - ◆ Use informal natural language

Problematic???

Problematic!

# UML IN SOFTWARE ENGG

DR TRIPTY SINGH

SHREEVIDYA

# Overview

- What is modeling?
- What is UML?
- Use case diagrams
- Class diagrams
- Sequence diagrams
- Activity diagrams
- Summary

## The Origin of UML

- The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations. UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design and deployment).

# Why UML

- As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs. The primary goals in the design of the UML summarize by Page-Jones in Fundamental Object-Oriented Design in UML as follows:
  1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
  2. Provide extensibility and specialization mechanisms to extend the core concepts.
  3. Be independent of particular programming languages and development processes.
  4. Provide a formal basis for understanding the modeling language.
  5. Encourage the growth of the OO tools market.
  6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
  7. Integrate best practices.

# What is UML and Why we use UML?

## ■ Why we use UML?

- Use graphical notation: more clearly than natural language (imprecise) and code (too detailed).
- Help acquire an overall view of a system.
- UML is *not* dependent on any one language or technology.
- UML moves us from fragmentation to standardization.

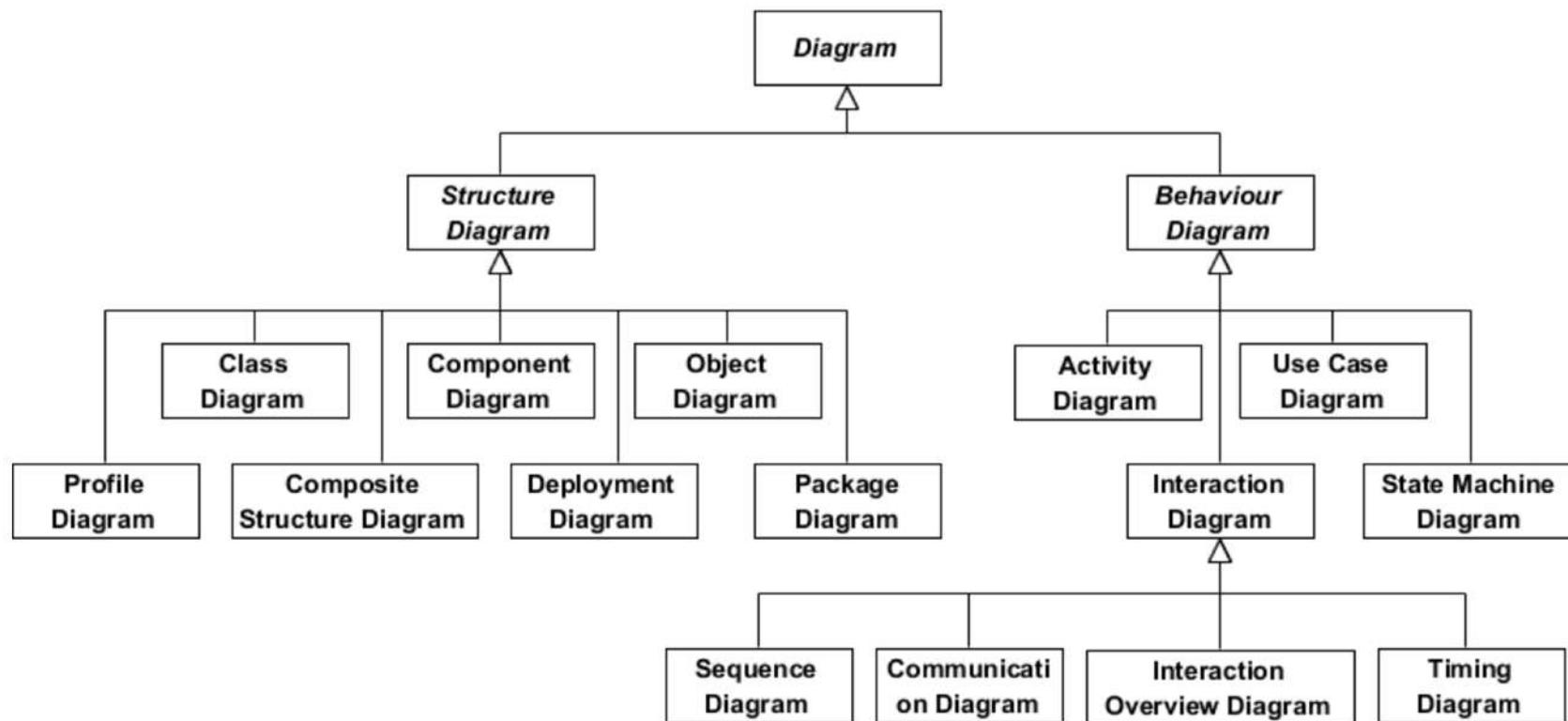
## UML-An Overview

- UML - Before we begin to look at the theory of the UML, we are going to take a very brief run through some of the major concepts of the UML.
- The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part. For Example:
  - Analysts
  - Designers
  - Coders
  - Testers
  - QA
  - The Customer
  - Technical Authors
- All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code. By contrast, a technical writer is interested in the behavior of the system as a whole, and needs to understand how the product functions. The UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram

# UML-DIAGRAMS

- Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts, there are seven types of structure diagram as follows:
  - Class Diagram
  - Component Diagram
  - Deployment Diagram
  - Object Diagram
  - Package Diagram
  - Composite Structure Diagram
  - Profile Diagram
- Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time, there are seven types of behavior diagrams as follows:
  - Use Case Diagram
  - Activity Diagram
  - State Machine Diagram
  - Sequence Diagram
  - Communication Diagram
  - Interaction Overview Diagram
  - Timing Diagram

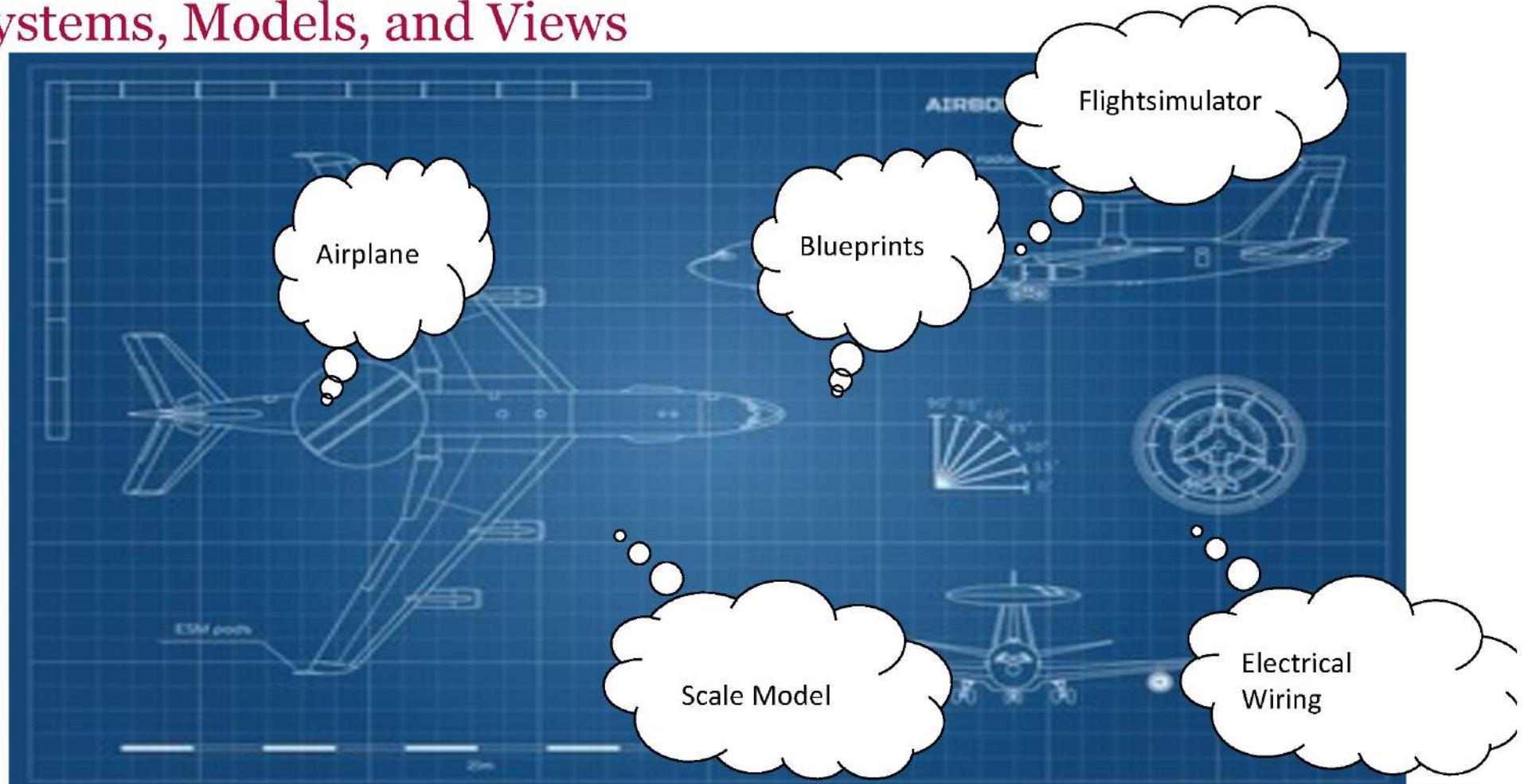
# UML-DIAGRAMS



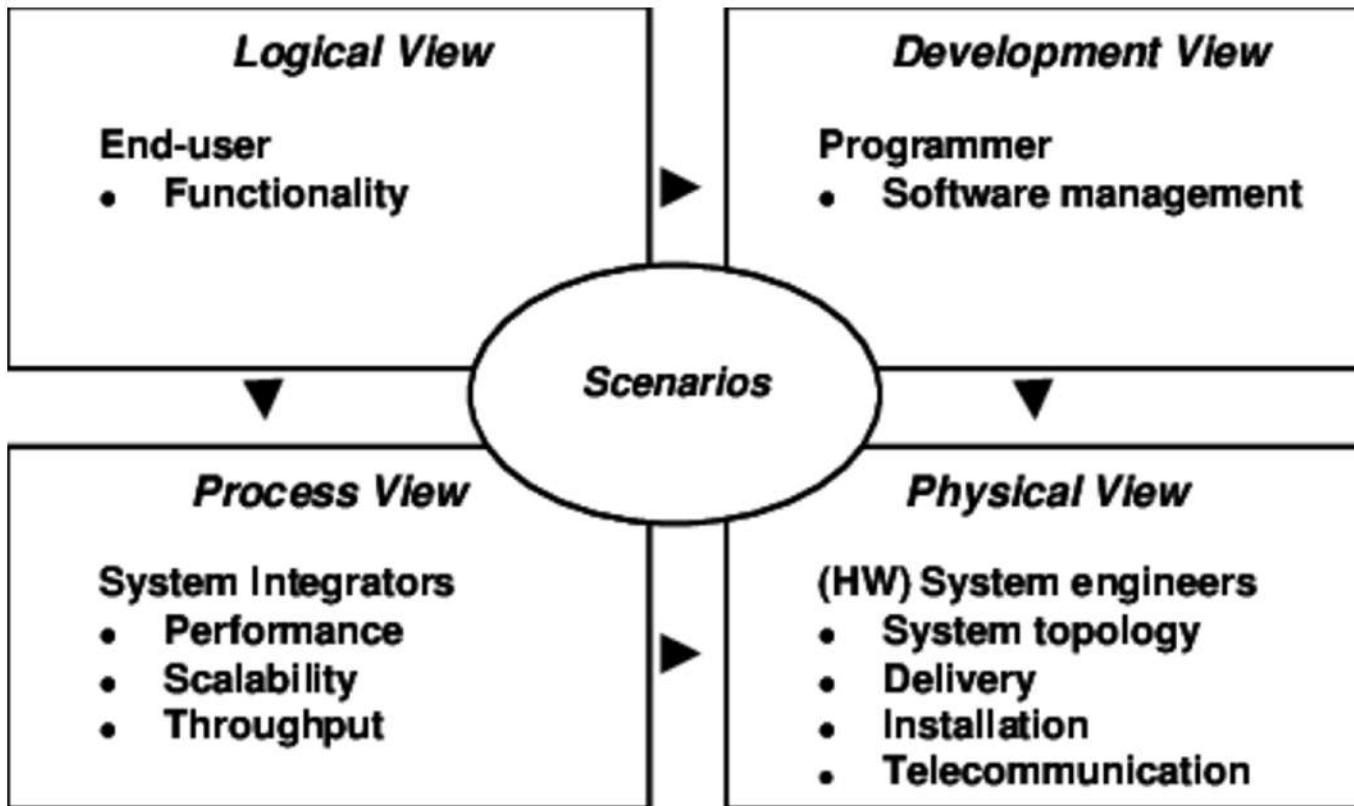
## Systems, Models, and Views

- A *model* is an abstraction describing system or a subset of a system
- A *view* depicts selected aspects of a model
- A *notation* is a set of graphical or textual rules for representing views
- Views and models of a single system may overlap each other

# Systems, Models, and Views



# Systems, Models, and Views



# How to use UML diagrams to design software system?

- Types of UML Diagrams:

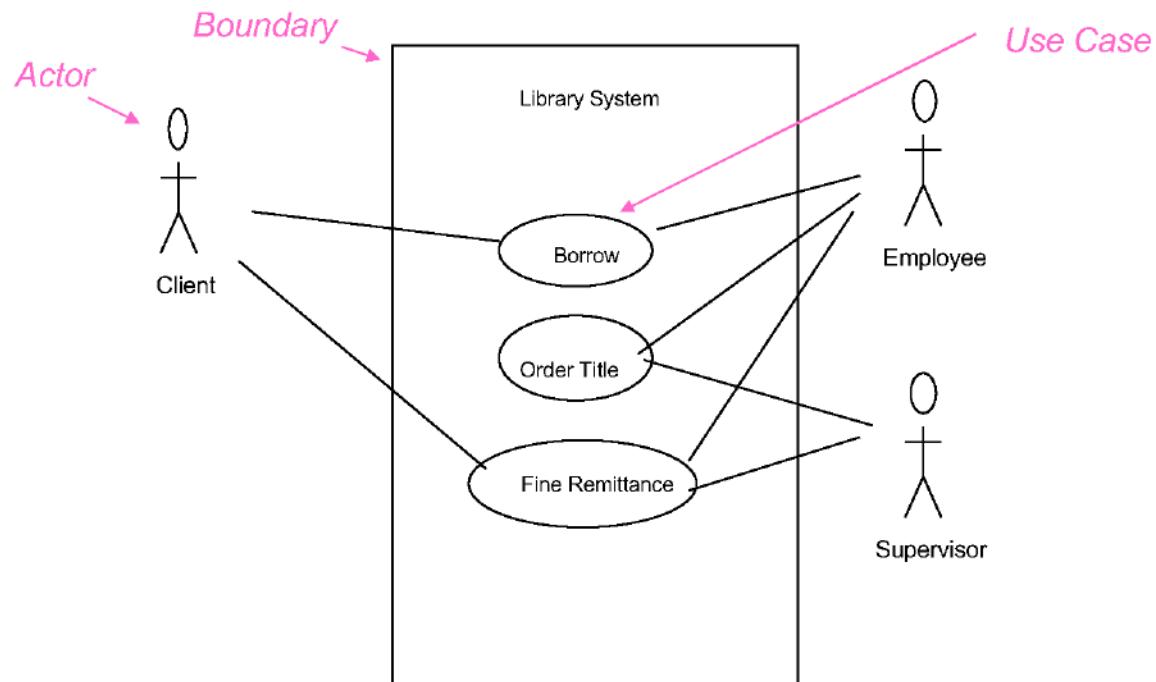
- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- State Diagram

This is only a subset of diagrams ... but are most widely used

# Use-Case Diagrams

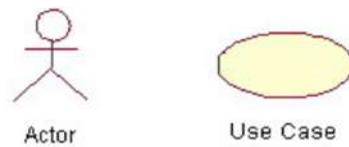
- A use-case diagram is a set of use cases
- A use case is a model of the interaction between
  - External users of a software product (actors) and
  - The software product itself
  - More precisely, an actor is a user playing a specific role
- describing a set of user **scenarios**
- capturing user requirements
- **contract** between end user and software developers

# Use-Case Diagrams



# Use-Case Diagrams

- **Actors:** A role that a user plays with respect to the system, including human users and other systems. e.g., inanimate physical objects (e.g. robot); an external system that needs some information from the current system.
- **Use case:** A set of scenarios that describing an interaction between a user and a system, including alternatives.
- **System boundary:** rectangle diagram representing the boundary between the actors and the system.



# Use-Case Diagrams

- Association:

communication between an actor and a use case; Represented by a solid line.

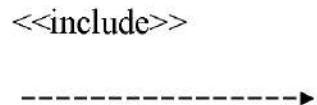
---

- Generalization: relationship between one general use case and a special use case (used for defining special alternatives)  
Represented by a line with a triangular arrow head toward the parent use case.

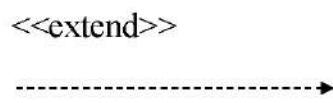


# Use-Case Diagrams

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrow pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use “include” in stead of copying the description of that behavior.



Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares “extension points”.



# Use-Case Diagrams

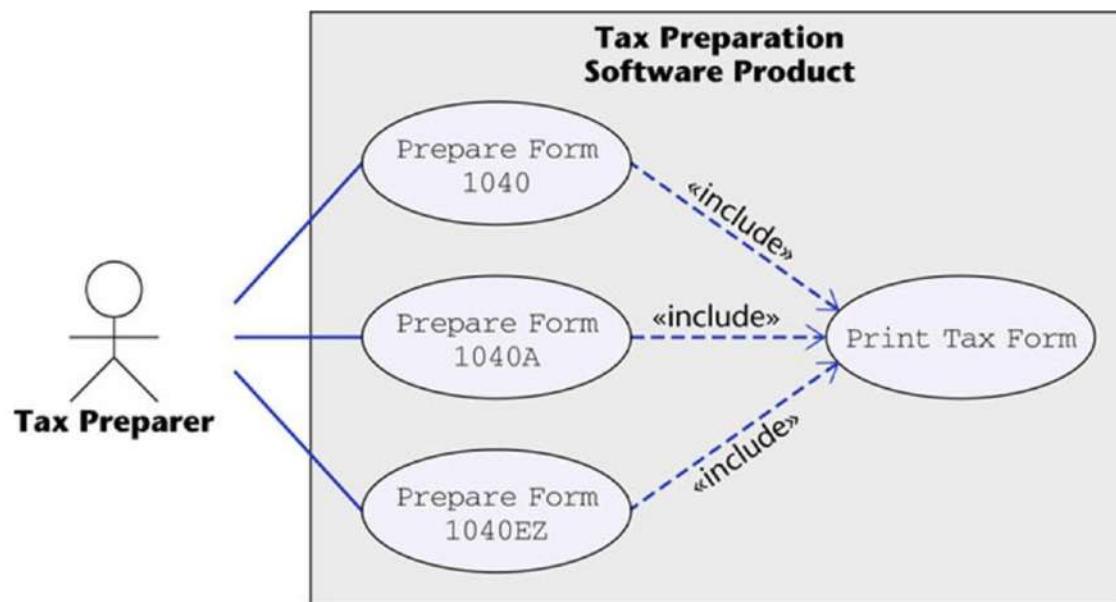
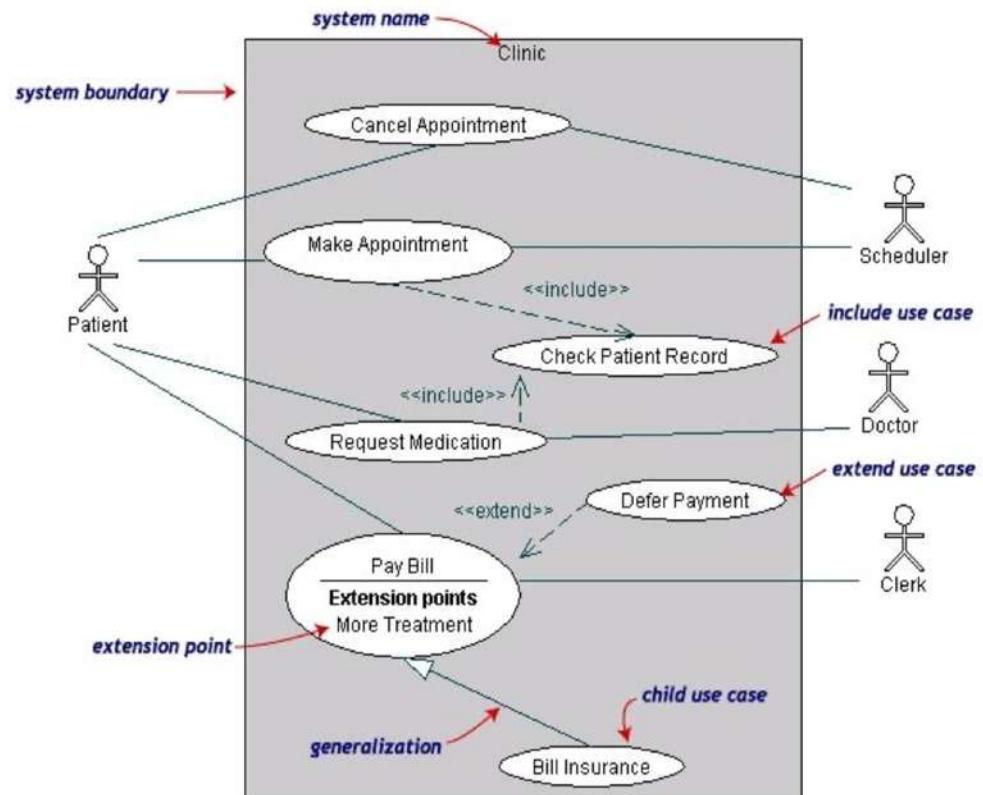


Figure 16.12

# Use-Case Diagrams

- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)
- The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)
- Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)



(TogetherSoft, Inc)

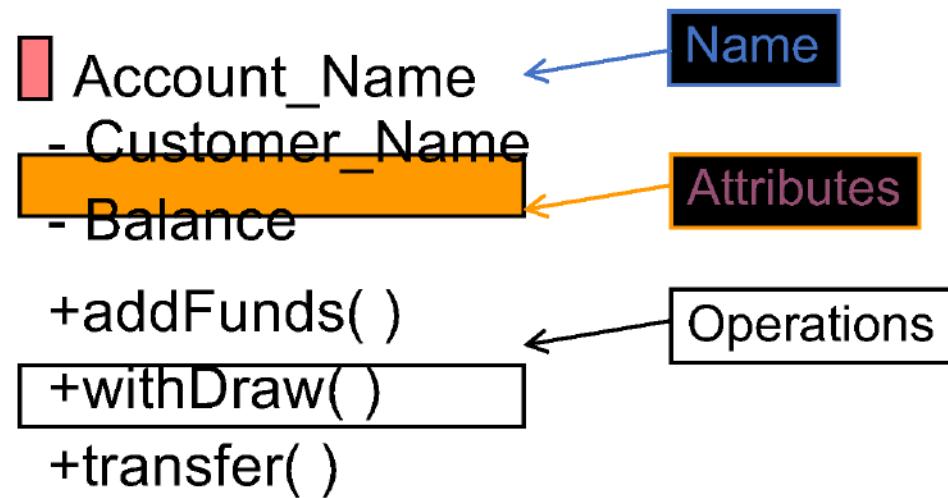
# Class diagram

- A class diagram depicts classes and their interrelationships
- Used for describing **structure and behavior** in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

# Class diagram

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations
- Modifiers are used to indicate visibility of attributes and operations.
  - ‘+’ is used to denote *Public* visibility (everyone)
  - ‘#’ is used to denote *Protected* visibility (friends and derived)
  - ‘-’ is used to denote *Private* visibility (no one)
- By default, attributes are hidden and operations are visible.

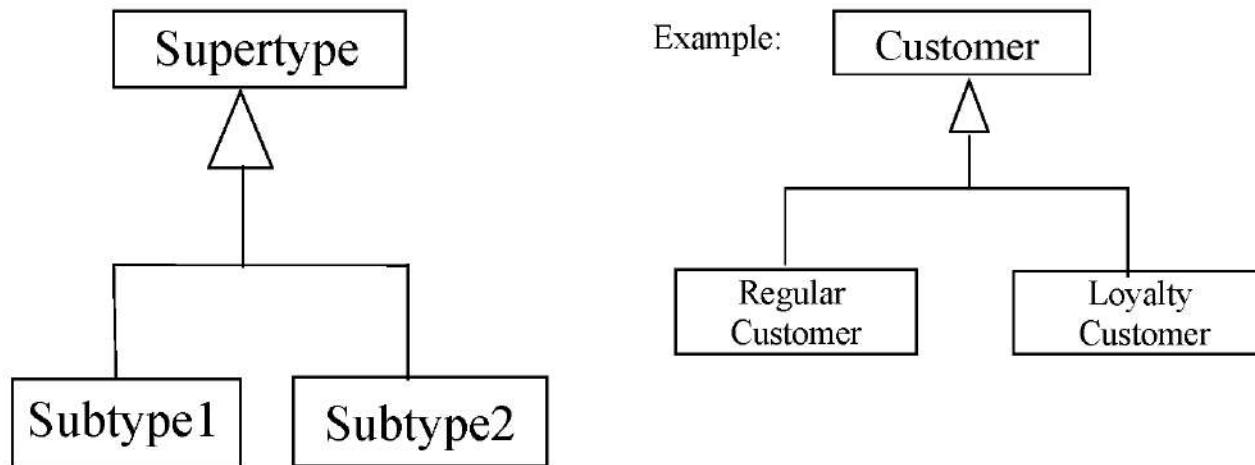
# Class diagram



# OO Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)
- Associations can be further classified as
  - Aggregation
  - Composition

## OO Relationships: **Generalization**

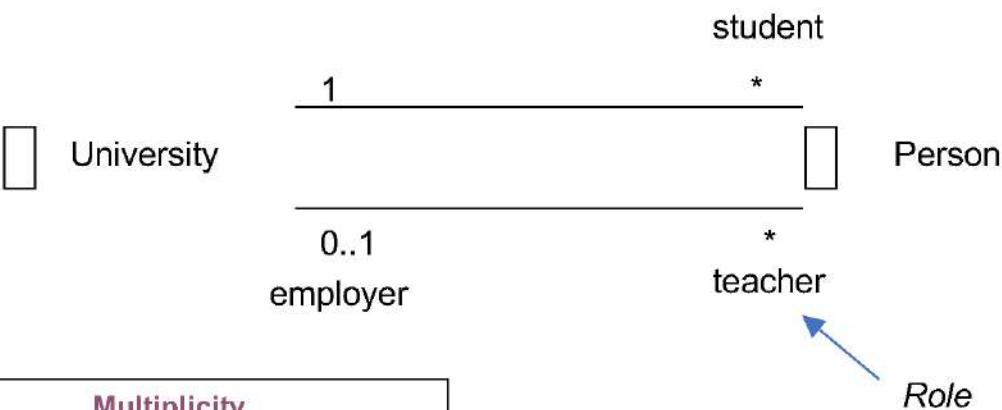


- Inheritance is a required feature of object orientation
- Generalization expresses a parent/child relationship among related classes.
- Used for abstracting details in several layers

# OO Relationships: Association

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)

# Association: Multiplicity and Roles

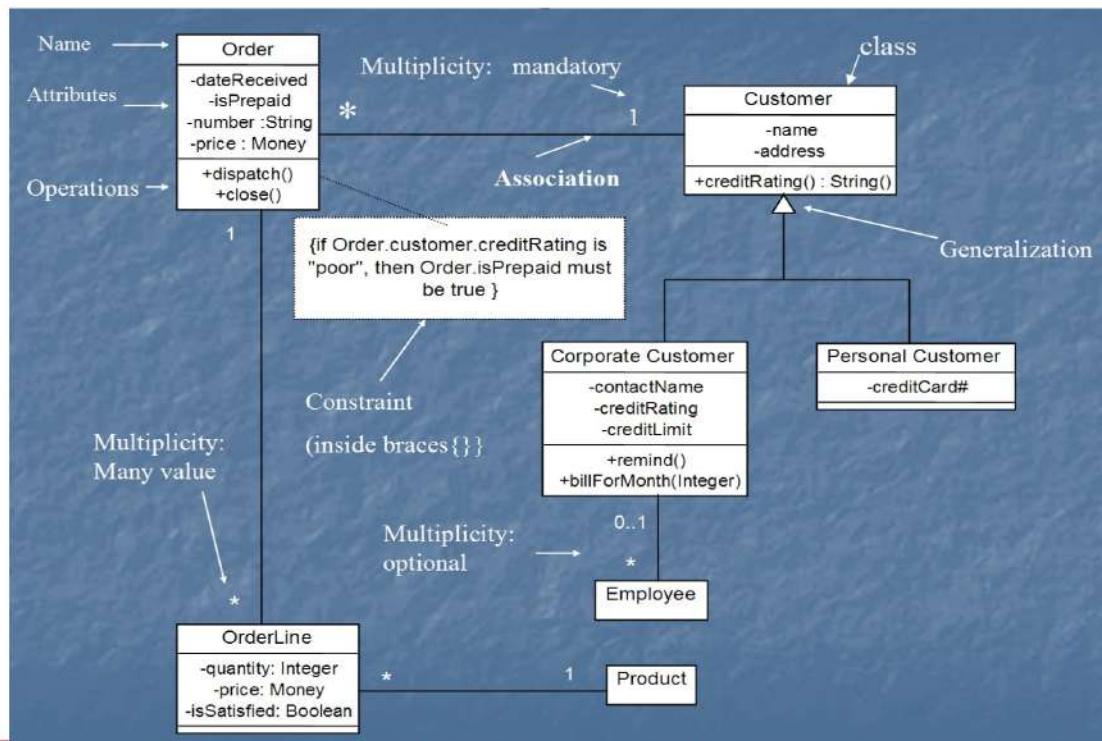


Multiplicity	
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

**Role**

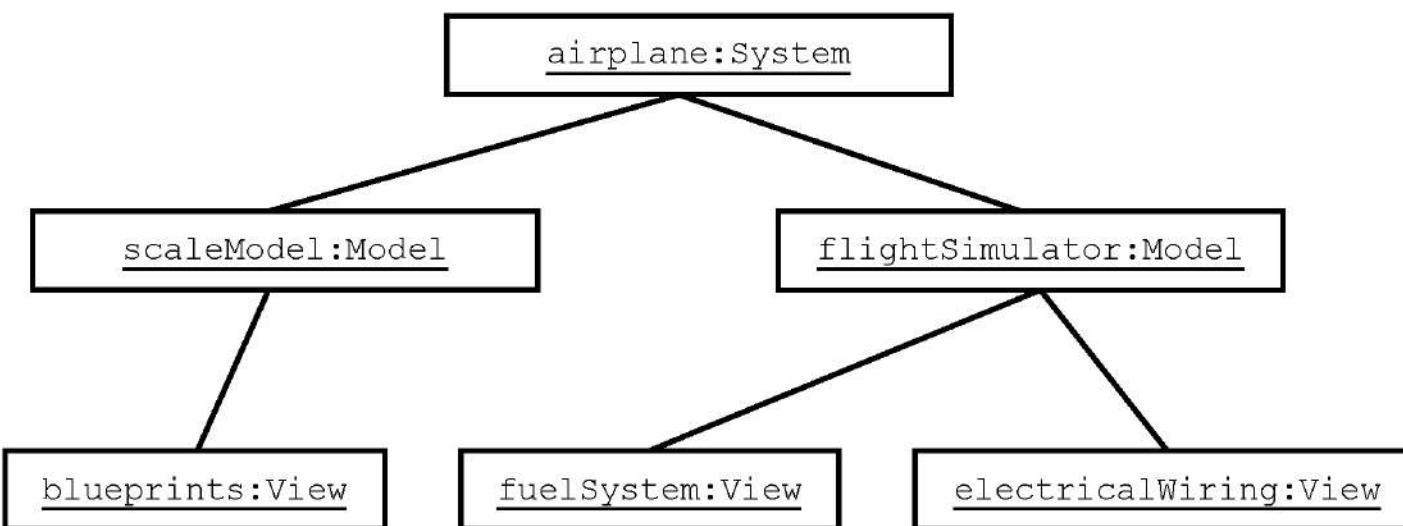
*"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."*

# Class diagram



[from UML Distilled Third Edition]

# Models, Views, and Systems (UML)

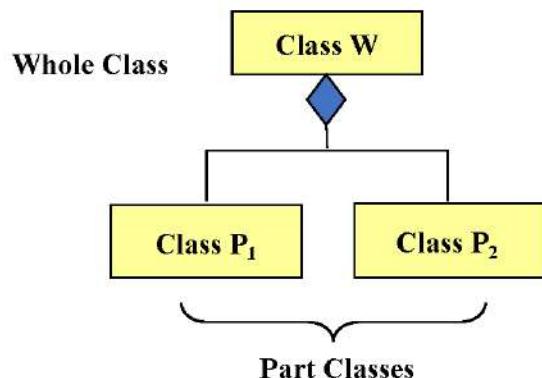


## Association: Model to Implementation



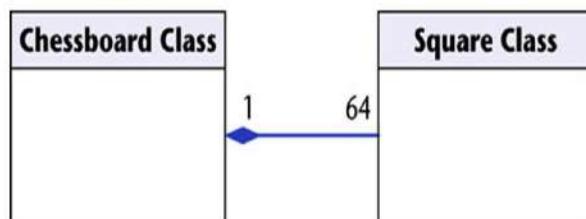
```
Class Student {  
    Course enrolls[4];  
}  
  
Class Course {  
    Student have[];  
}
```

# OO Relationships: **Composition**



[From Dr.David A. Workman]

## Example



## Association

Models the part–whole relationship

## Composition

Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

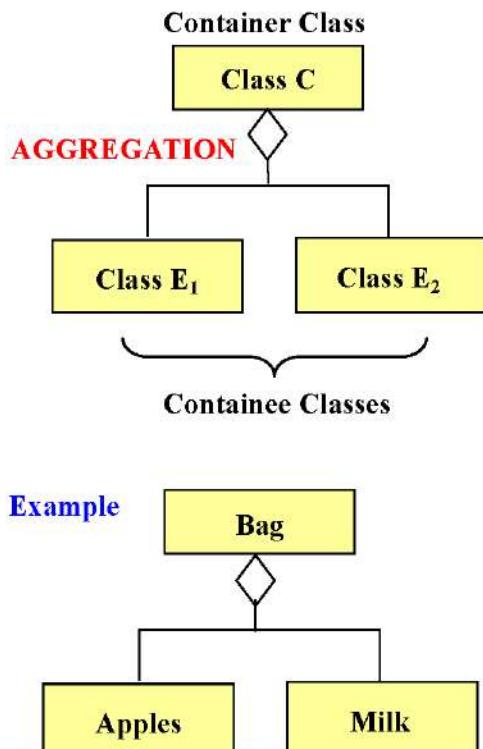
## Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

Figure 16.7

The McGraw-Hill Companies, 2005

# OO Relationships: Aggregation



## Aggregation:

expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

[From Dr.David A. Workman]

# Aggregation vs. Composition

## ■ **Composition** is really a strong form of **association**

- components have only one owner
- components cannot exist independent of their owner
- components live or die with their owner
- e.g. Each car has an engine that can not be shared with other cars.

## ■ **Aggregations**

may form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

# Good Practice: CRC Card

## Class Responsibility Collaborator

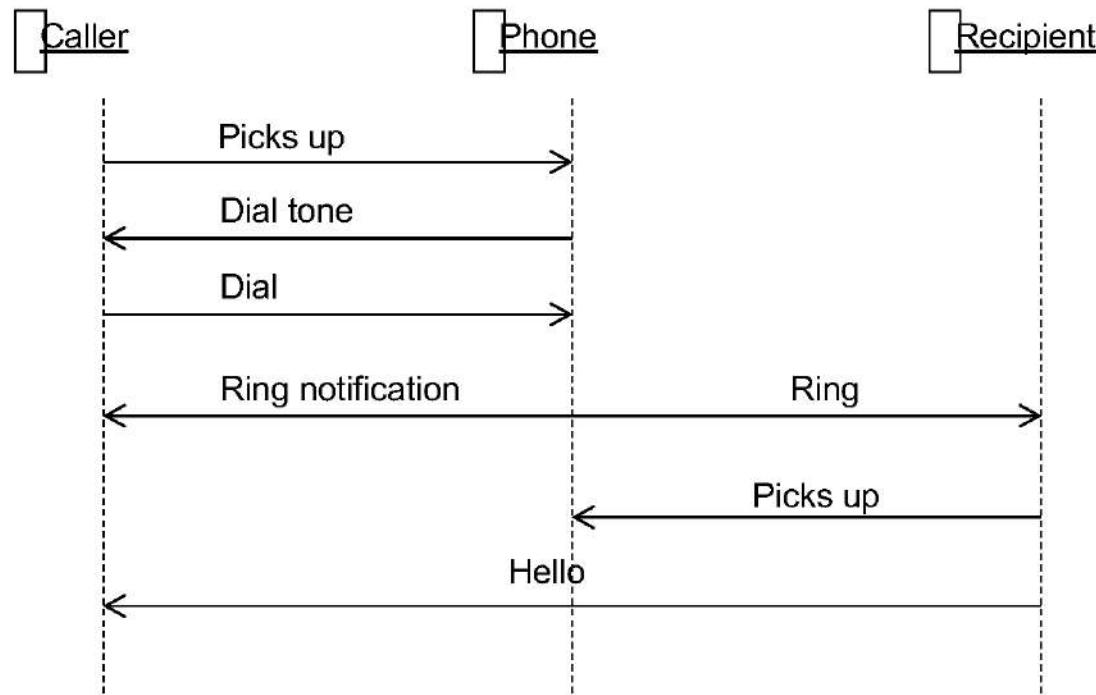
- easy to describe how classes work by moving cards around; allows to quickly consider alternatives.

<b>Class</b> Reservations	<b>Collaborators</b> <ul style="list-style-type: none"><li>▪ Catalog</li><li>▪ User session</li></ul>
<b>Responsibility</b> <ul style="list-style-type: none"><li>▪ Keep list of reserved titles</li><li>▪ Handle reservation</li></ul>	

## Interaction Diagrams

- show how objects interact with one another
- UML supports two types of interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams

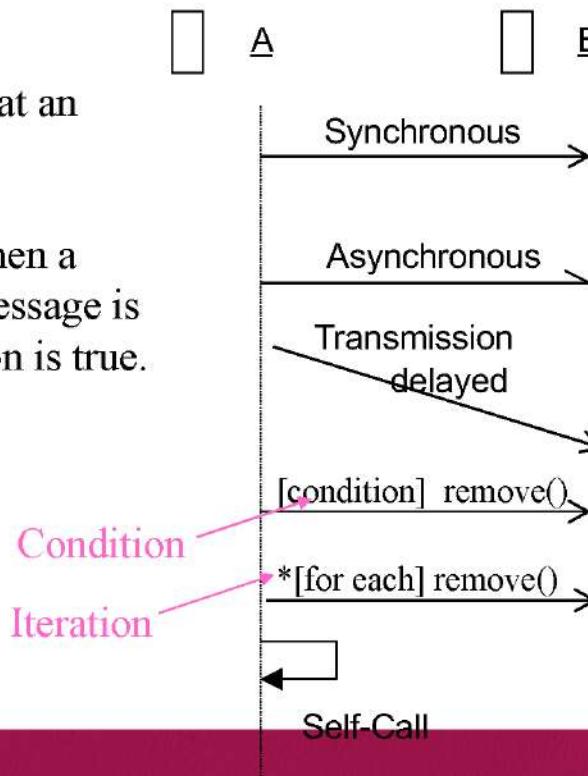
## Sequence Diagram(make a phone call)



# Sequence Diagram: Object interaction

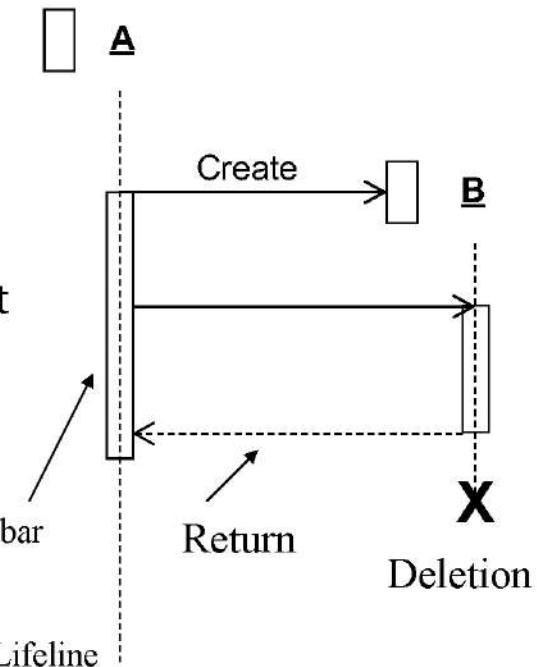
*Self-Call:* A message that an Object sends to itself.

*Condition:* indicates when a message is sent. The message is sent only if the condition is true.



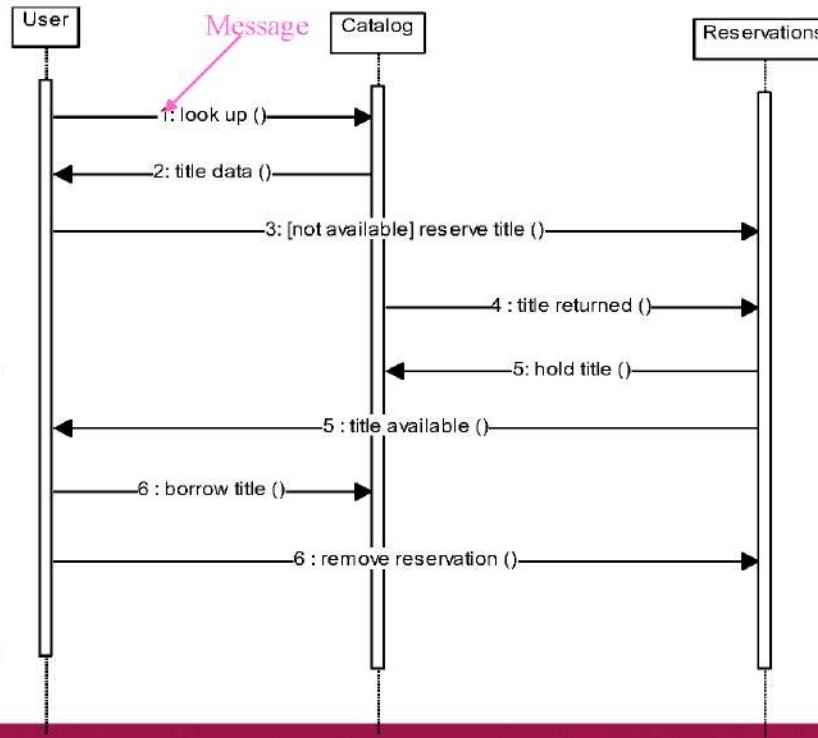
## Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point

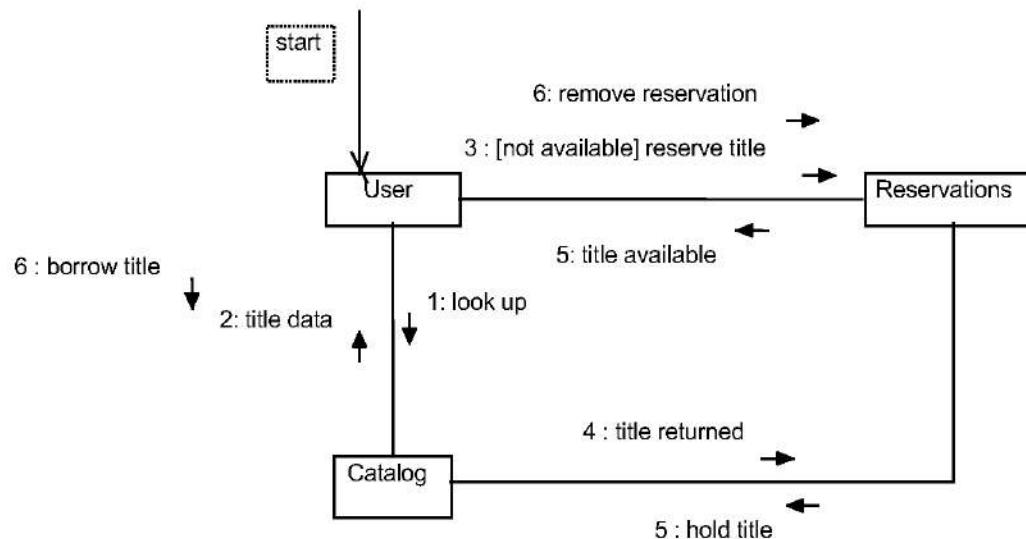


# Sequence Diagram

- Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.
- The horizontal dimension shows the objects participating in the interaction.
- The vertical arrangement of messages indicates their order.
- The labels may contain the seq. # to indicate concurrency.



## Interaction Diagrams: Collaboration diagrams

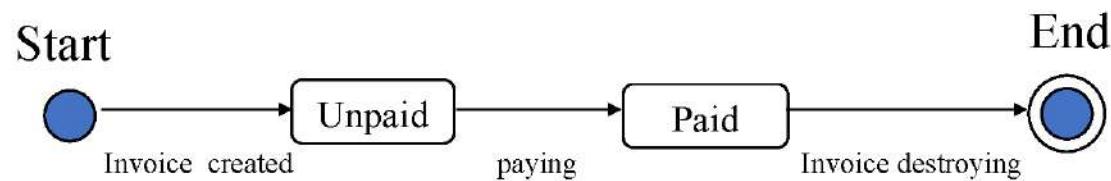


- Collaboration diagrams are equivalent to sequence diagrams. All the features of sequence diagrams are equally applicable to collaboration diagrams
- Use a sequence diagram when the transfer of information is the focus of attention

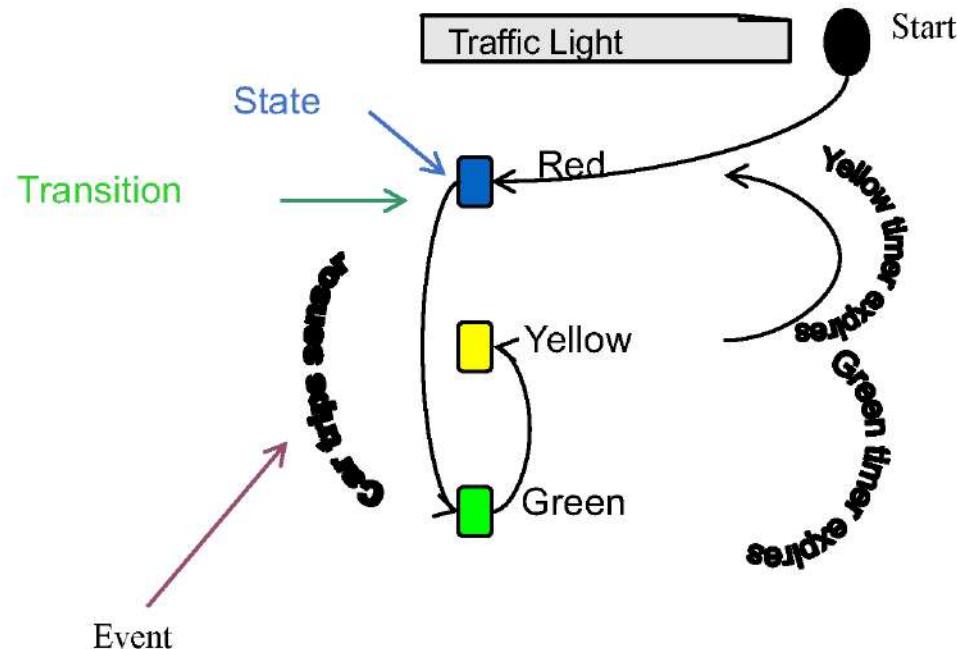
- Use a collaboration diagram when concentrating on the classes

## State Diagrams (Billing Example)

State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



## State Diagrams (Traffic light example)

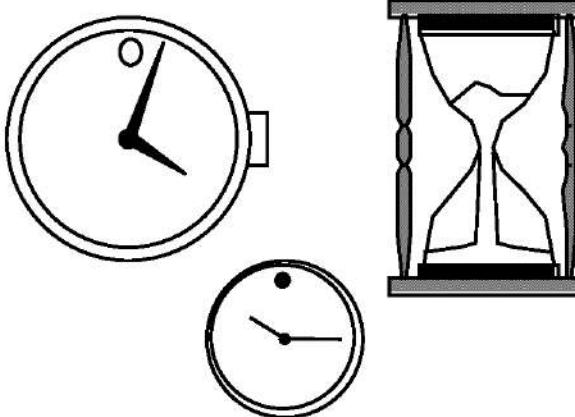


## Why model software?

Software is already an abstraction: why model software?

- Software is getting larger, not smaller
  - NT 5.0 ~ 40 million lines of code
  - A single programmer cannot manage this amount of code in its entirety.
- Code is often not directly understandable by developers who did not participate in the development
- We need simpler representations for complex systems
  - Modeling is a mean for dealing with complexity

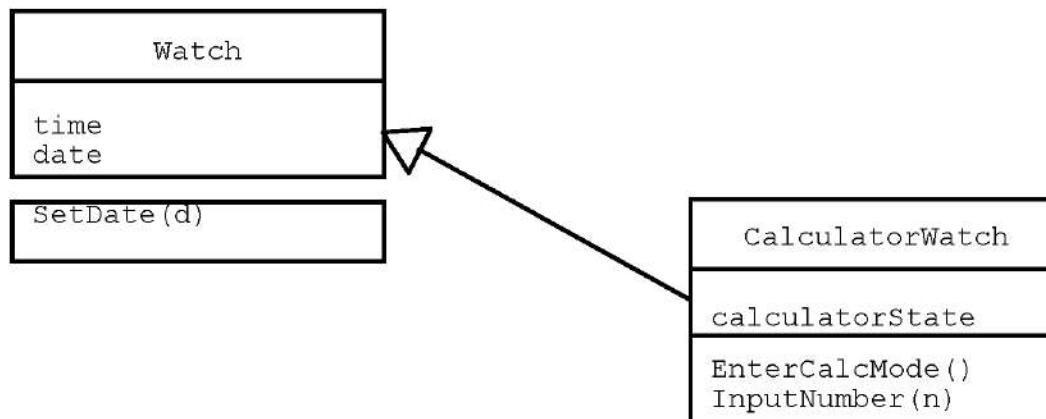
# Concepts and Phenomena

Name	Purpose	Members
Clock	A device that measures time.	 A diagram showing a round wall clock and a sand timer (hourglass) side-by-side, representing different devices used for measuring time.

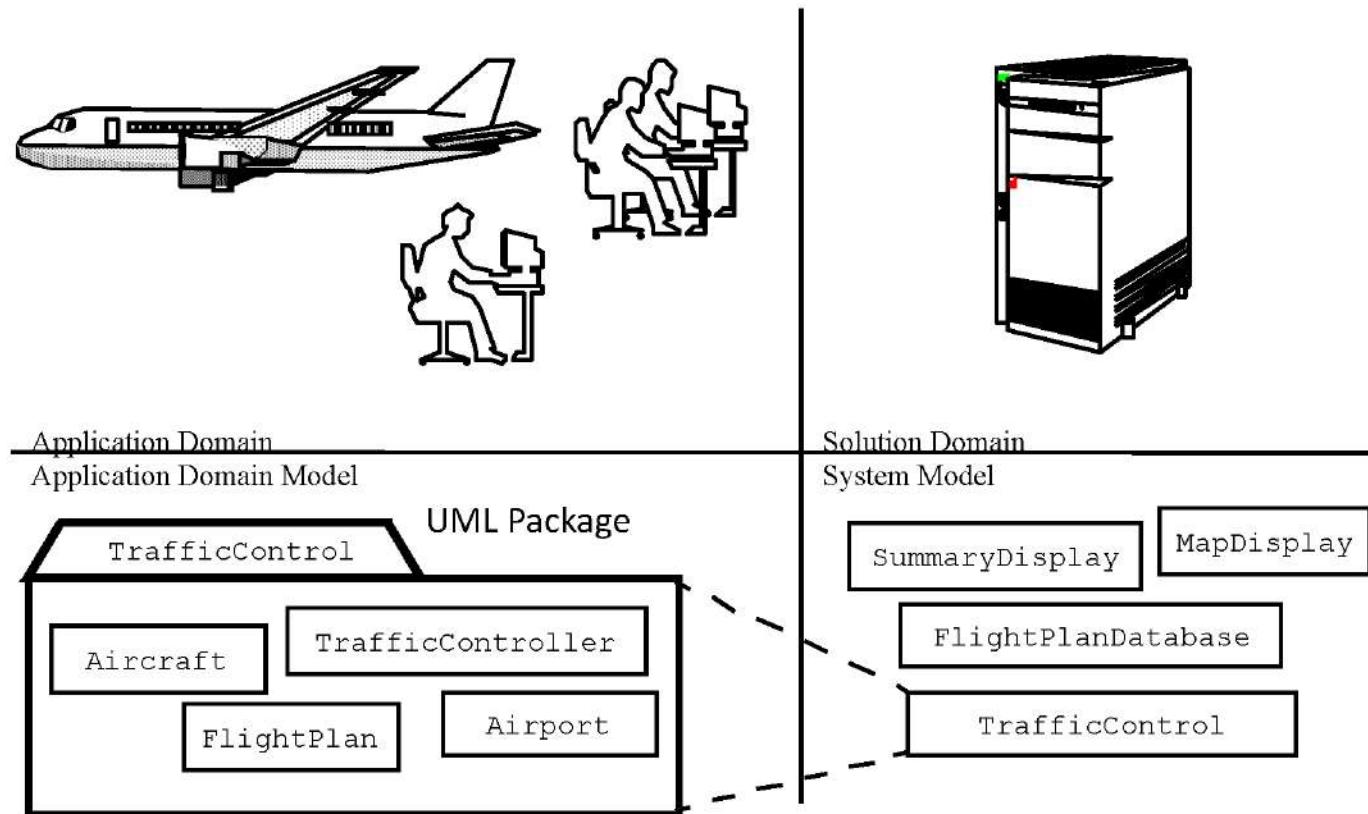
- Abstraction: Classification of phenomena into concepts
- Modeling: Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

# Class

- Class:
  - An abstraction in the context of object-oriented languages
  - Like an abstract data type, a class encapsulates both state (variables) and behavior (methods)
  - Unlike abstract data types, classes can be defined in terms of other classes using inheritance



# Object-Oriented Modeling



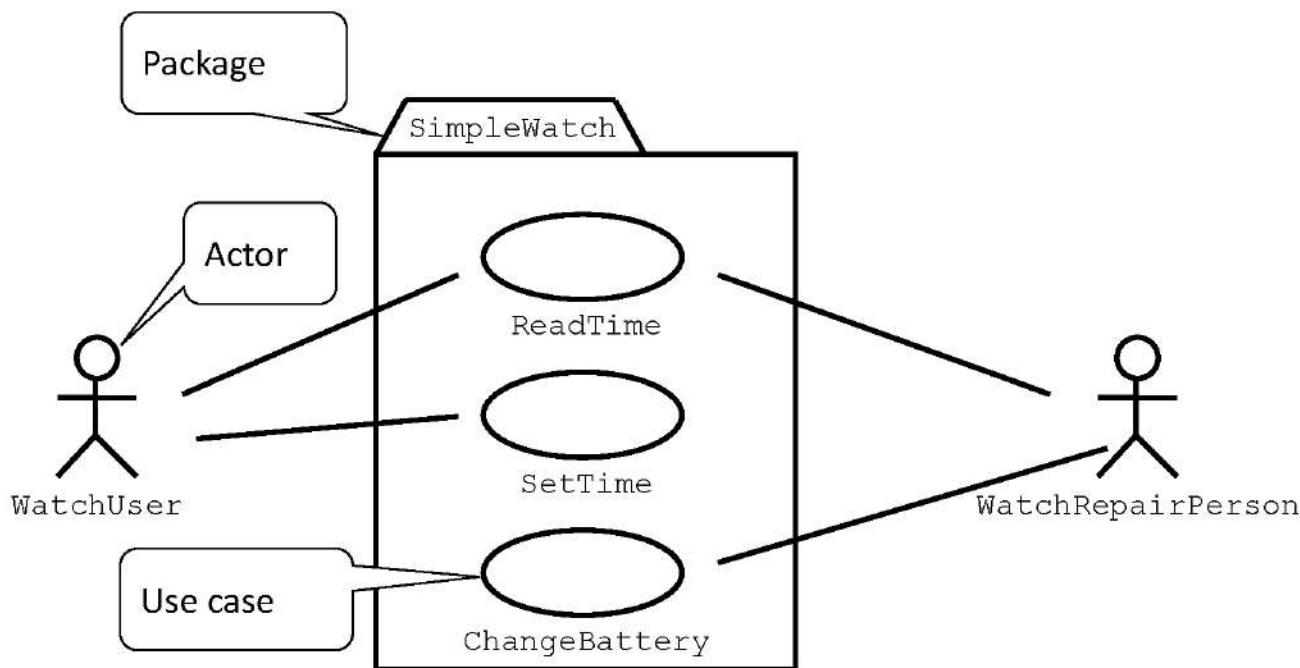
## **Application and Solution Domain**

- Application Domain (Requirements Analysis):
  - The environment in which the system is operating
- Solution Domain (System Design, Object Design):
  - The available technologies to build the system

## UML First Pass

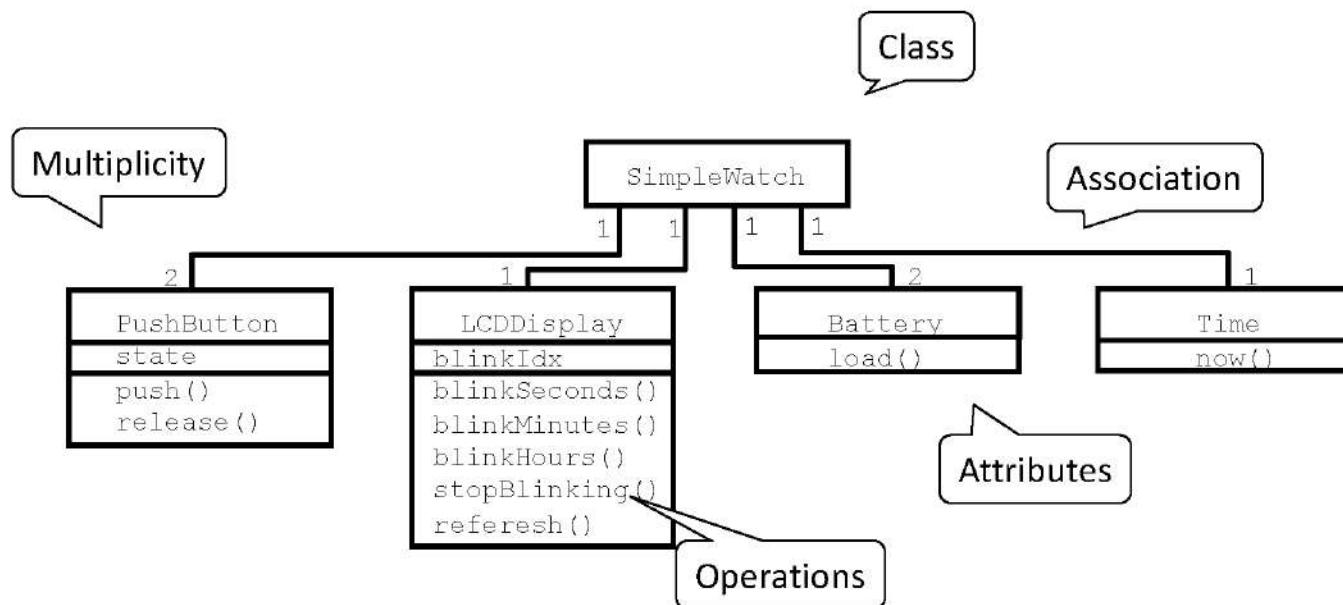
- Use case diagrams
  - Describe the functional behavior of the system as seen by the user.
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, and Associations.
- Sequence diagrams
  - Describe the dynamic behavior between actors and the system and between objects of the system.
- Statechart diagrams
  - Describe the dynamic behavior of an individual object as a finite state machine.
- Activity diagrams
  - Model the dynamic behavior of a system, in particular the workflow, i.e. a flowchart.

# UML First Pass: Use Case Diagrams



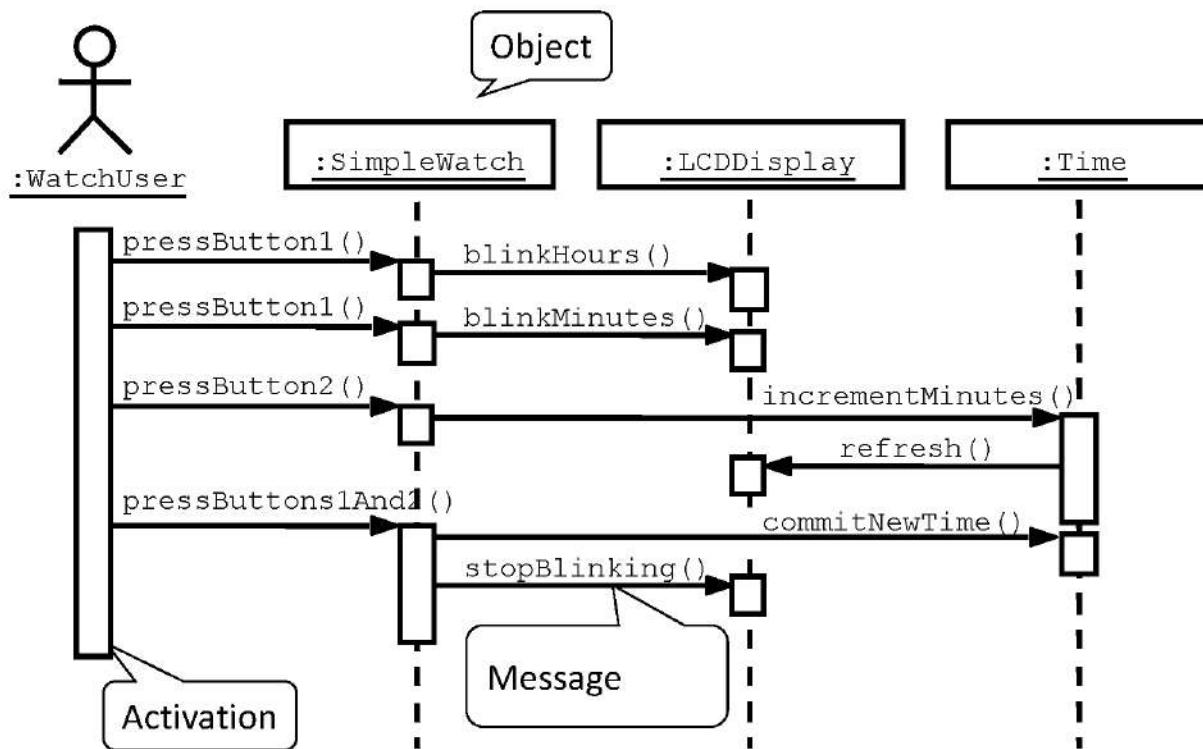
Use case diagrams represent the functionality of the system from user's point of view

# UML First Pass: Class Diagrams



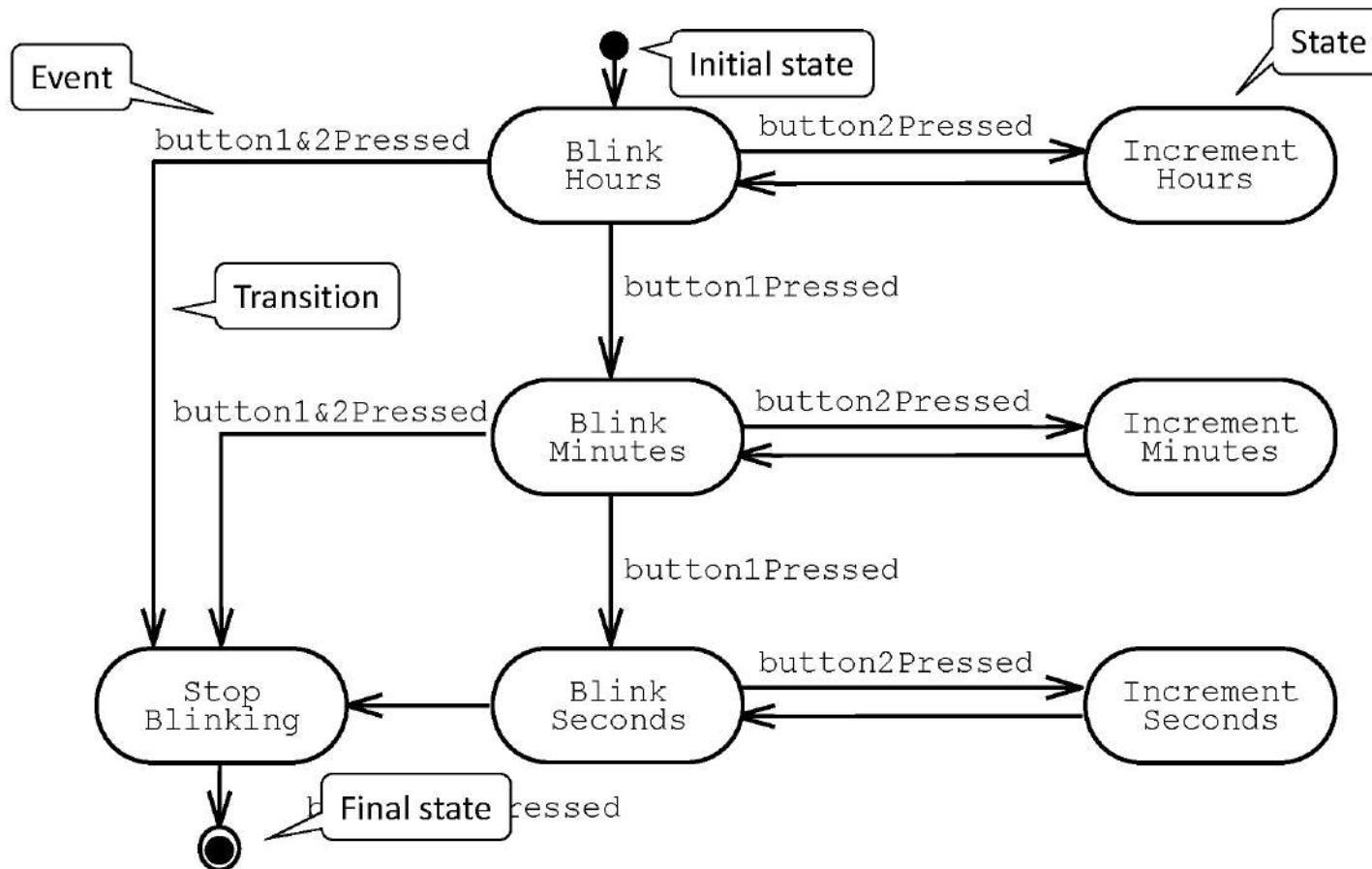
Class diagrams represent the structure of the system

# UML First Pass: Sequence Diagram



Sequence diagrams represent the behavior as interactions

# UML First Pass: Statechart Diagrams



## Other UML Notations

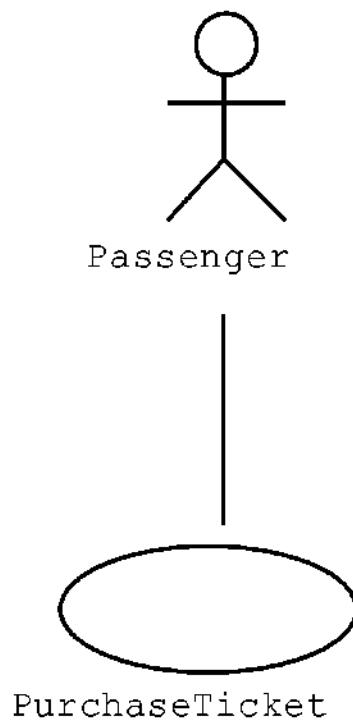
UML provide other notations that we will be introduced in subsequent lectures, as needed.

- Implementation diagrams
  - Component diagrams
  - Deployment diagrams
  - Introduced in lecture on System Design
- Object Constraint Language (OCL)
  - Introduced in lecture on Object Design

## UML Core Conventions

- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - myWatch:SimpleWatch
  - Joe:Firefighter
- Types are denoted with nonunderlined names
  - SimpleWatch
  - Firefighter
- Diagrams are graphs
  - Nodes are entities
  - Arcs are relationships between entities

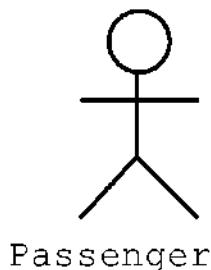
# UML Second Pass: Use Case Diagrams



Used during requirements elicitation  
to represent external behavior

- **Actors** represent roles, that is, a type of user of the system
- **Use cases** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

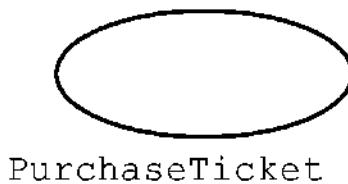
# Actors



- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description.
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

# Use Case

A use case represents a class of functionality provided by the system as an event flow.



A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

# Use Case Example

*Name:* Purchase ticket

*Participating actor:* Passenger

*Entry condition:*

- Passenger standing in front of ticket distributor.
- Passenger has sufficient money to purchase ticket.

*Exit condition:*

- Passenger has ticket.

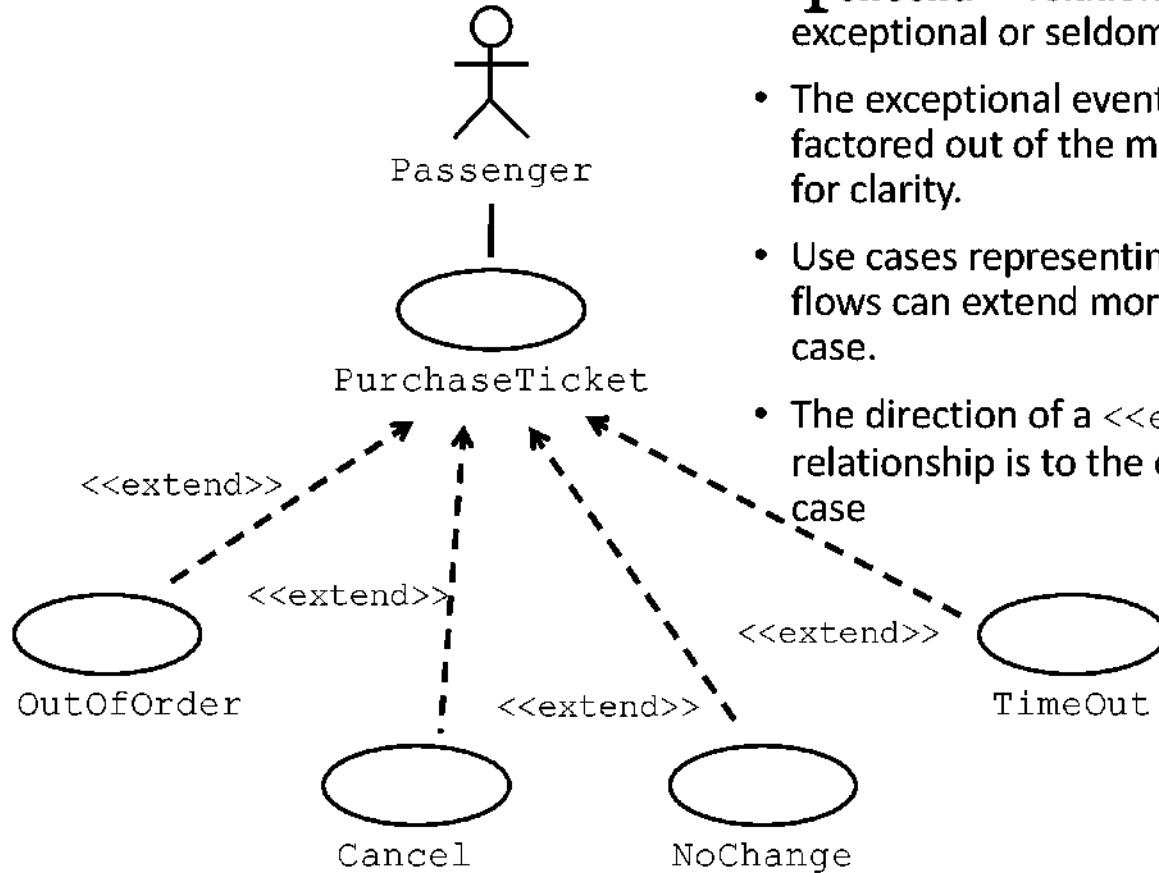
*Event flow:*

1. Passenger selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. Passenger inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

Exceptional cases!

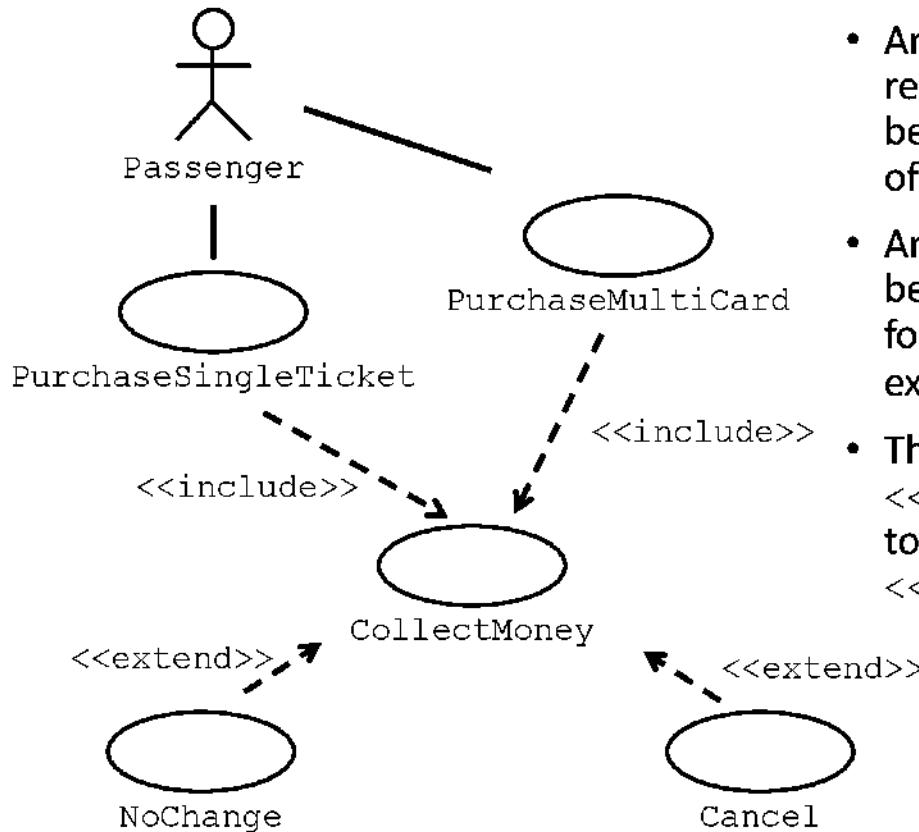
# The <<extend>> Relationship



<<extend>> relationships represent exceptional or seldom invoked cases.

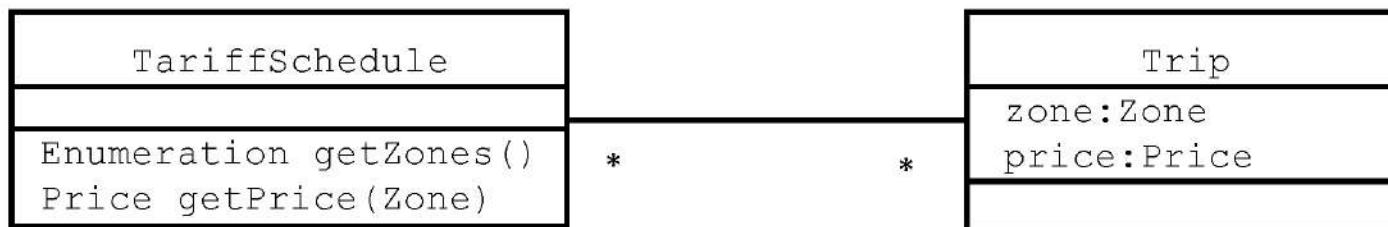
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extend>> relationship is to the extended use case

# The <<include>> Relationship



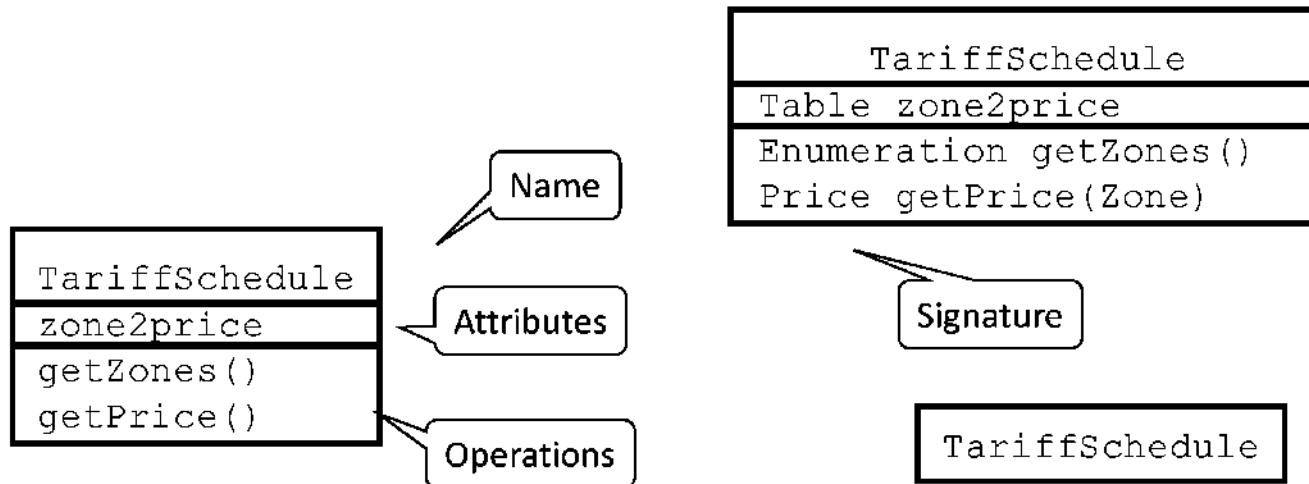
- An <<include>> relationship represents behavior that is factored out of the use case.
- An <<include>> represents behavior that is factored out for reuse, not because it is an exception.
- The direction of a <<include>> relationship is to the using use case (unlike <<extend>> relationships).

# Class Diagrams



- Class diagrams represent the structure of the system.
- Class diagrams are used
  - during requirements analysis to model problem domain concepts
  - during system design to model subsystems and interfaces
  - during object design to model classes.

# Classes



- A **class** represent a concept.
- A class encapsulates state (**attributes**) and behavior (**operations**).
- Each attribute has a **type**.
- Each operation has a **signature**.
- The class name is the only mandatory information.

# Instances

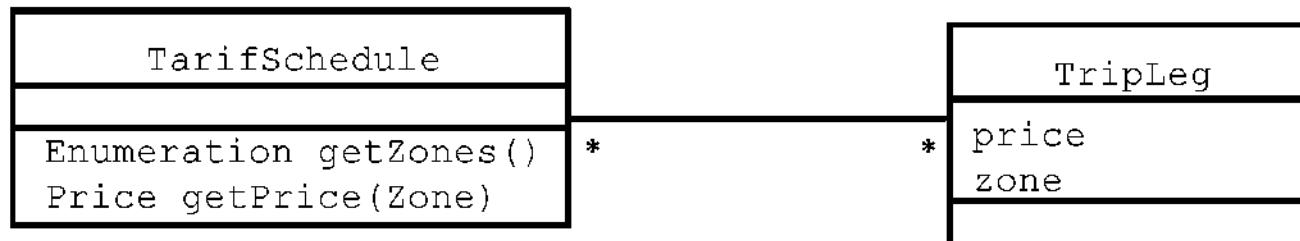
```
tariff 1974:TarifSchedule
zone2price = {
  {'1', .20},
  {'2', .40},
  {'3', .60}}
```

- An *instance* represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their *values*.

## Actor vs. Instances

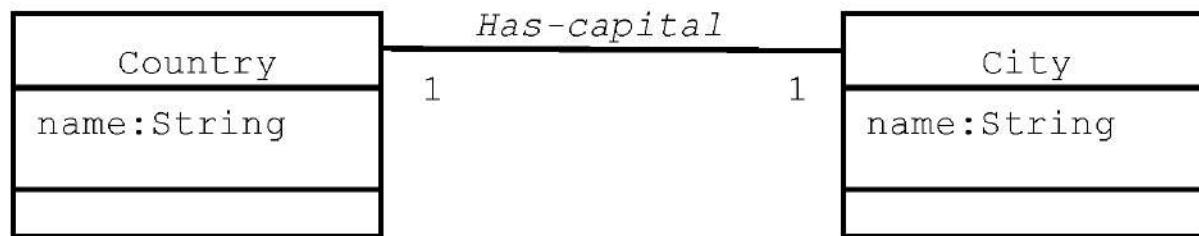
- What is the difference between an actor and a class and an instance?
- Actor:
  - An entity outside the system to be modeled, interacting with the system (“Pilot”)
- Class:
  - An abstraction modeling an entity in the problem domain, inside the system to be modeled (“Cockpit”)
- Object:
  - A specific instance of a class (“Joe, the inspector”).

# Associations

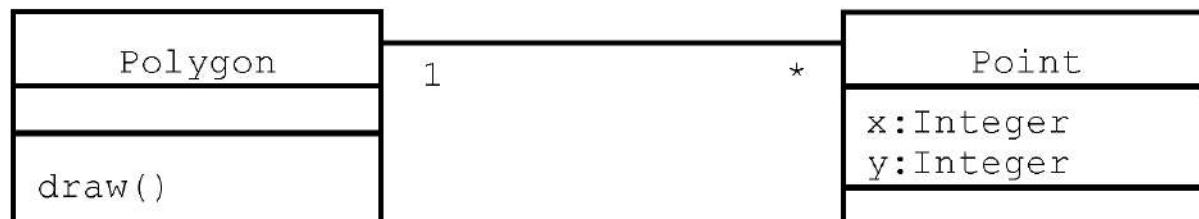


- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

## 1-to-1 and 1-to-Many Associations



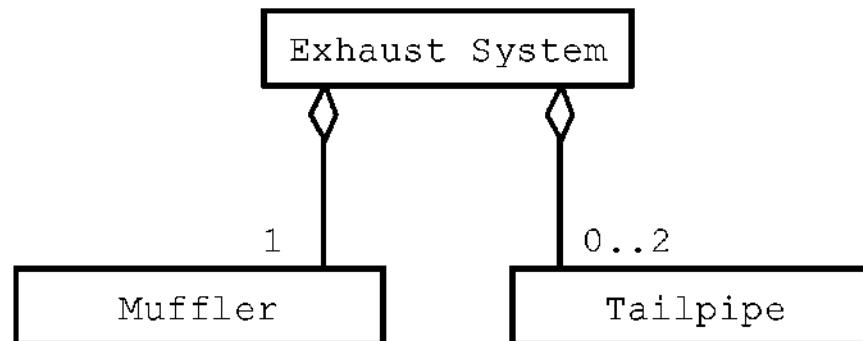
1-to-1 association



1-to-many association

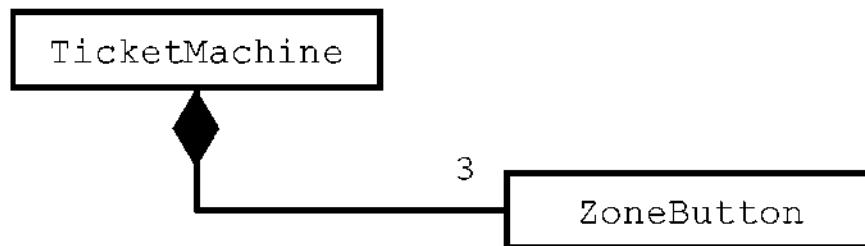
# Aggregation

- An ***aggregation*** is a special case of association denoting a “consists of” hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.

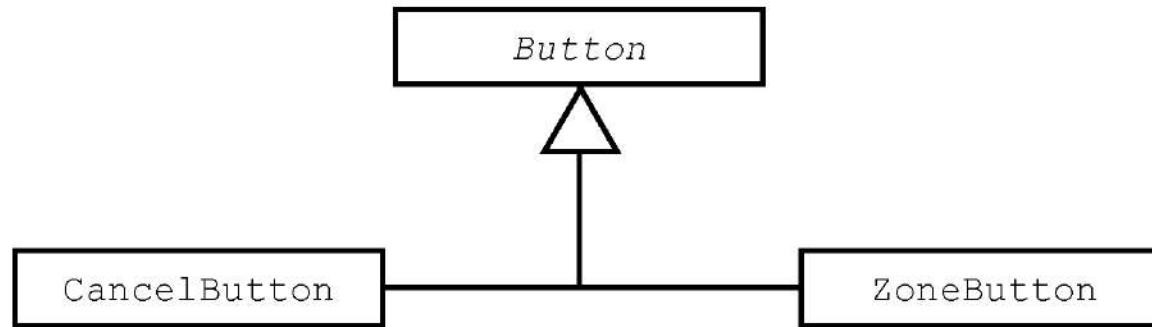


# Composition

- A solid diamond denote ***composition***, a strong form of aggregation where components cannot exist without the aggregate.



# Generalization



- Generalization relationships denote inheritance between classes.
- The children classes inherit the attributes and operations of the parent class.
- Generalization simplifies the model by eliminating redundancy.

# From Problem Statement to Code

## *Problem Statement*

A stock exchange lists many companies. Each company is identified by a ticker symbol

## *Class Diagram*

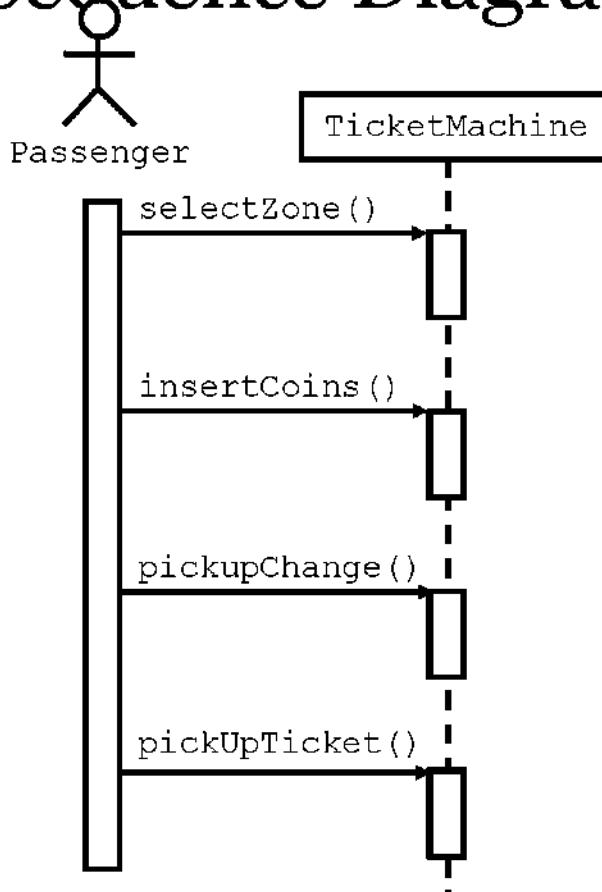


## *Java Code*

```
public class StockExchange {
    public Vector m_Company = new Vector();
};

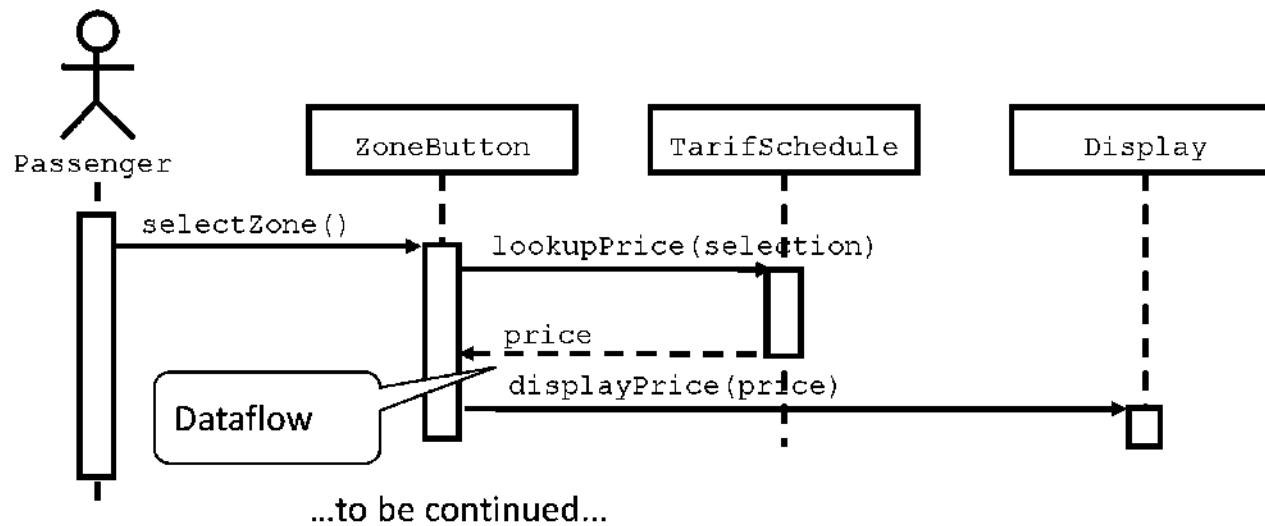
public class Company {
    public int m_tickerSymbol;
    public Vector m_StockExchange = new Vector();
};
```

# UML Sequence Diagrams



- Used during requirements analysis
  - To refine use case descriptions
  - to find additional objects (“participating objects”)
- Used during system design
  - to refine subsystem interfaces
- **Classes** are represented by columns
- **Messages** are represented by arrows
- **Activations** are represented by narrow rectangles
- **Lifelines** are represented by dashed lines

# UML Sequence Diagrams: Nested Messages

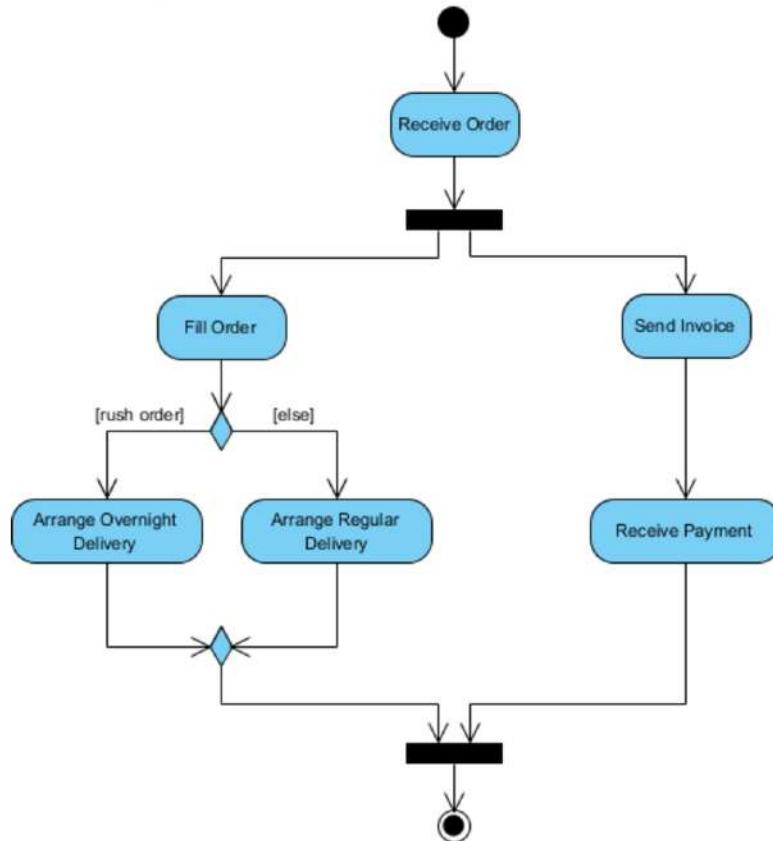


- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

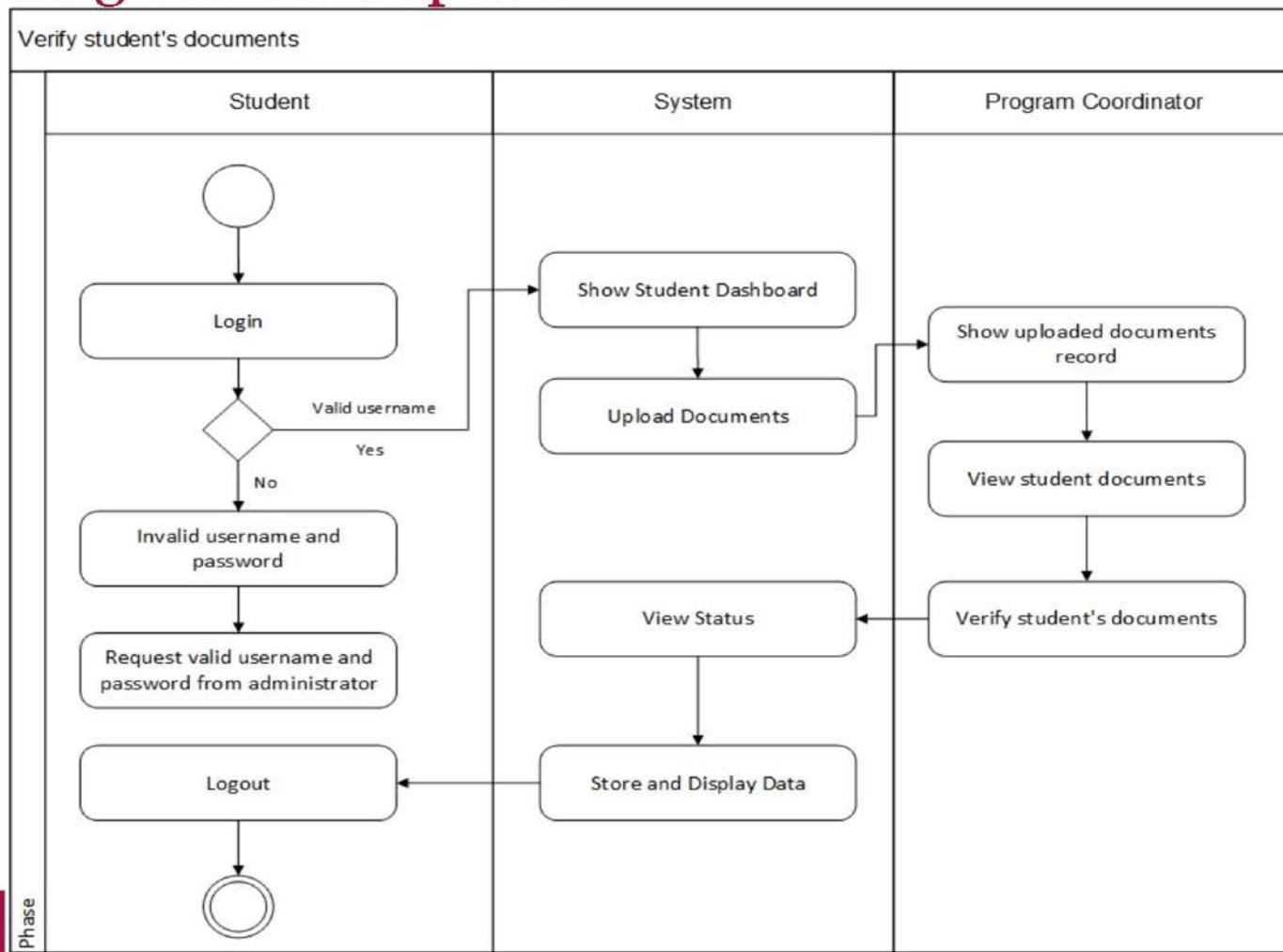
## Sequence Diagram Observations

- UML sequence diagram represent behavior in terms of interactions.
- Complement the class diagrams which represent structure.
- Useful to find participating objects.
- Time consuming to build but worth the investment.

## Activity Diagrams- simple



# Activity Diagrams- complex

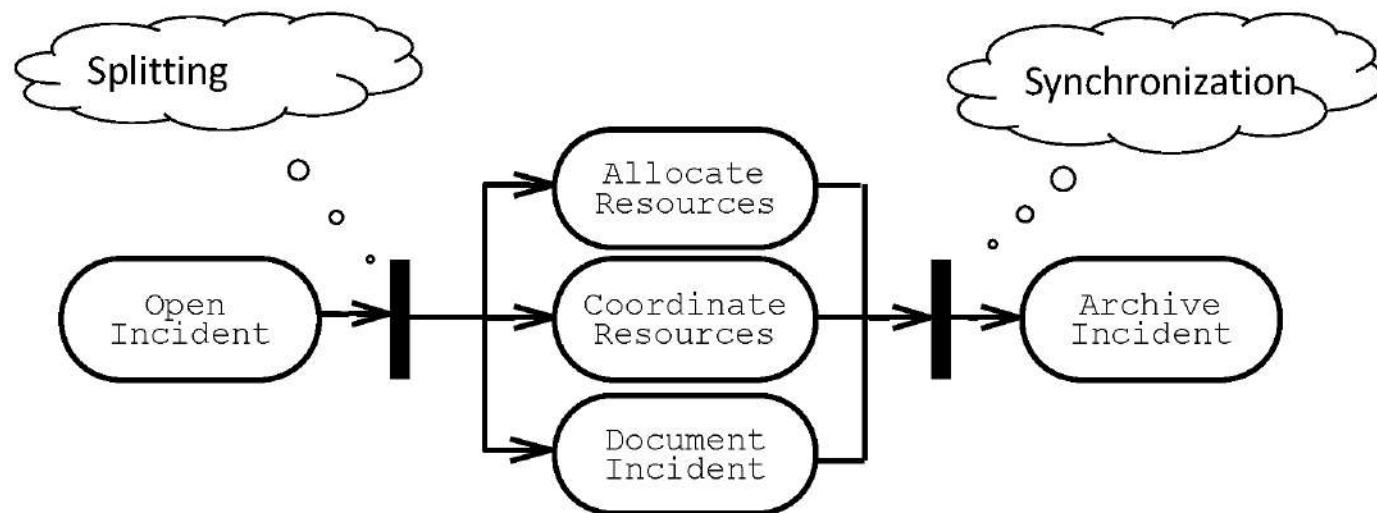


# Activity Diagrams

- An activity diagram shows flow control within a system
- An activity diagram is a special case of a state chart diagram in which states are activities (“functions”)
- Two types of states:
  - *Action state*:
    - Cannot be decomposed any further
    - Happens “instantaneously” with respect to the level of abstraction used in the model
  - *Activity state*:
    - Can be decomposed further
    - The activity is modeled by another activity diagram

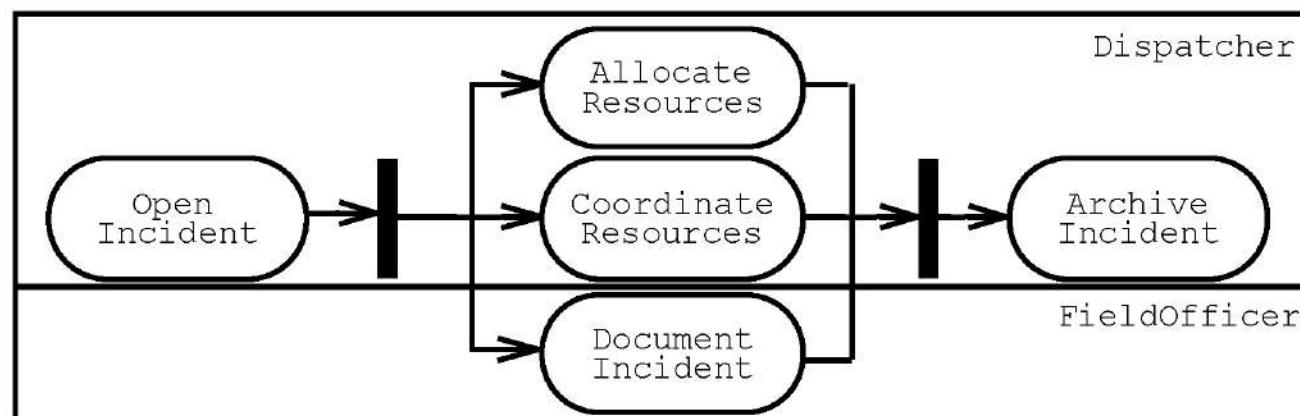
# Activity Diagrams: Modeling Concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



## Activity Diagrams: Swimlanes

- Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.



## Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features
- We concentrate only on a few notations:
  - Functional model: use case diagram
  - Object model: class diagram
  - Dynamic model: sequence diagrams, statechart and activity diagrams

# Last Lecture

- Definition of Done
- Definition of Ready
- Tracking Iteration Progress-BVIR

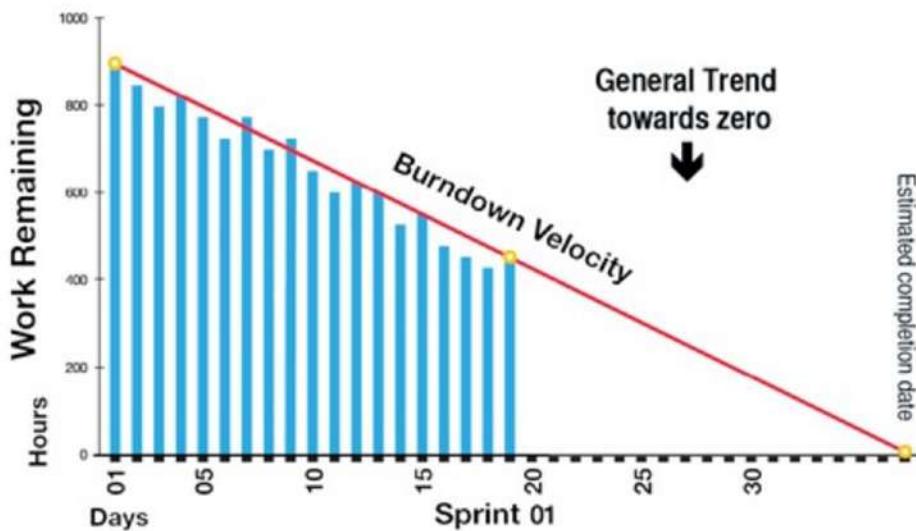
# Today's Session

- Various Release Reports
- Forecast and Management type of reports.

## Burndown Chart

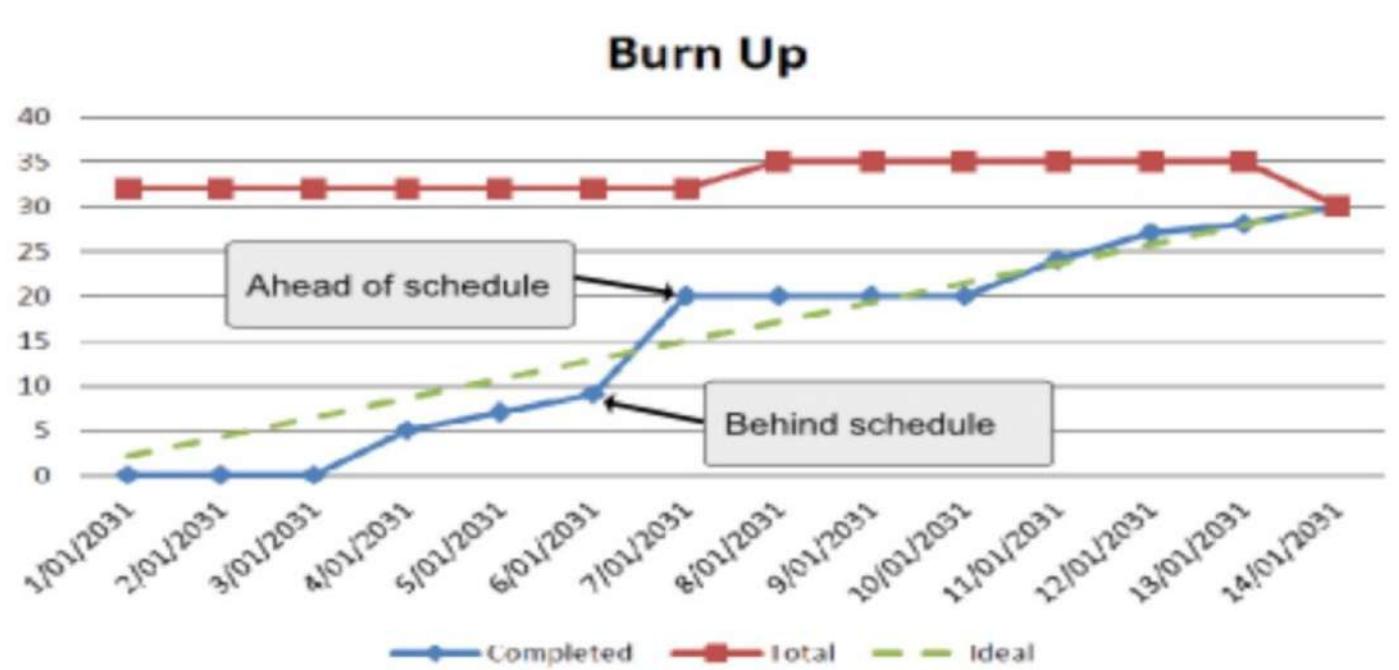
- A burndown chart shows the amount of work that has been completed in an epic or sprint, and the total work remaining. Burndown charts are used **to predict your team's likelihood of completing their work in the time available.**
- The burndown chart provides a day-by-day measure of the work that remains in a given sprint or release.
- The slope of the graph, or burndown velocity, is calculated by **comparing the number of hours worked to the original project estimation** and shows the average rate of productivity for each day.

- The progress of the team is tracked using a **burndown chart**, one of the best project visibility tools, and visually represents a project's progress. The burndown chart provides a day-by-day measure of the work that remains in a given sprint or release.
- The slope of the graph, or **burndown velocity**, is calculated by comparing the number of hours worked to the original project estimation and shows the average rate of productivity for each day.



# Burn Up Chart

- A Burn Up Chart is a tool used to track how much work has been completed, and **show the total amount of work for a project or iteration.**
- It's used by multiple software engineering methods but these charts are particularly popular in Agile and Scrum software project management.



## Sprint Report

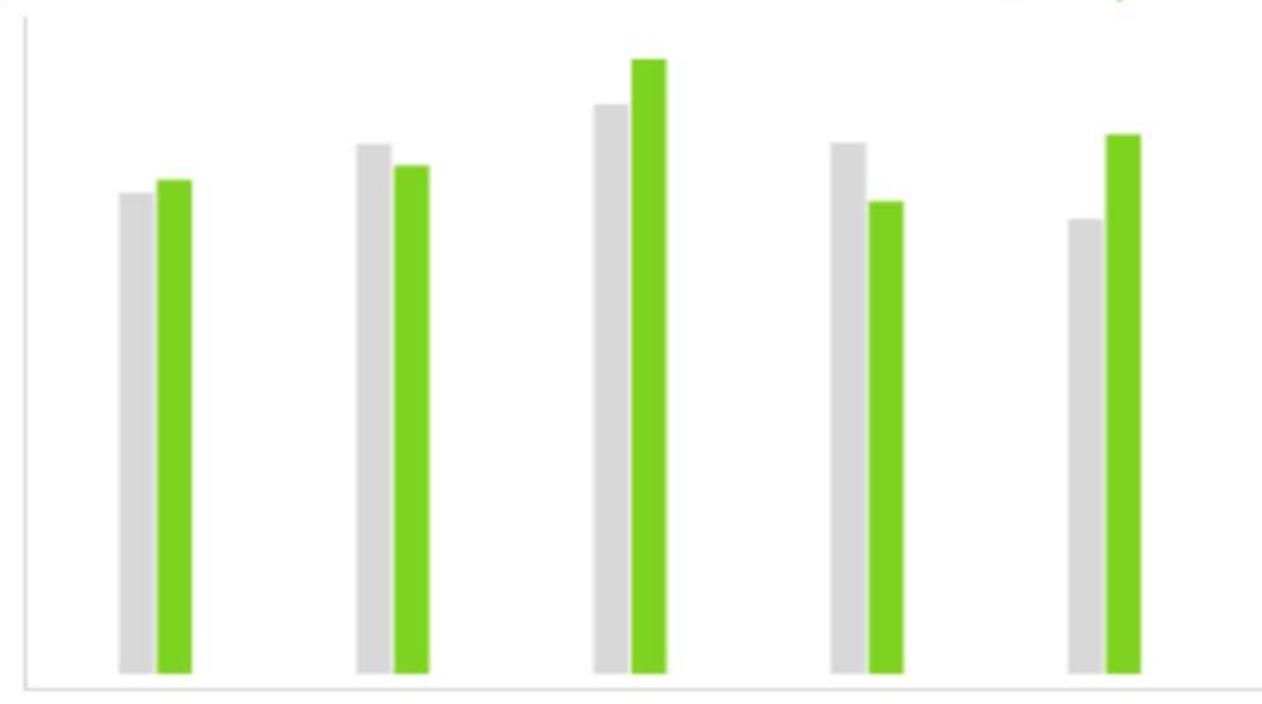
The Sprint Report shows the list of issues in each sprint. It is useful for your Sprint Retrospective meetings, and also for mid-sprint progress checks.

Understand the work completed or pushed back to the backlog in each sprint. This helps you determine if your team is overcommitting or if there is excessive scope creep.

## Velocity Chart

- The velocity chart displays **the average amount of work a scrum team completes during a sprint**. Teams can use velocity to predict how quickly they can work through the backlog because the report tracks the forecasted and completed work over several sprints. The more sprints, the more accurate the forecast.

#

**STORY  
POINTS**Estimated  
Completed100  
90  
80  
70  
60  
50  
40  
30  
20  
10**Sprint 4****Sprint 5****Sprint 6****Sprint 7****Sprint 8**

# Cumulative Flow Diagram

- The Cumulative Flow Diagram is a tool that **lets teams visualize the progress of their projects**. Teams can monitor the flow of work through its stages and gives the user the ability to predict blockers or disruptions in the progress of work.
- The Cumulative Flow Diagram (CFD) provides **typical information about status of your Scrum project**: how much work is done, ongoing and in backlog, what is the pace of progress, etc. ... The Cumulative Flow Diagram allows to detect the changes of scope and the work in progress.

## Reports: Cumulative Flow Diagram -

Board ▾



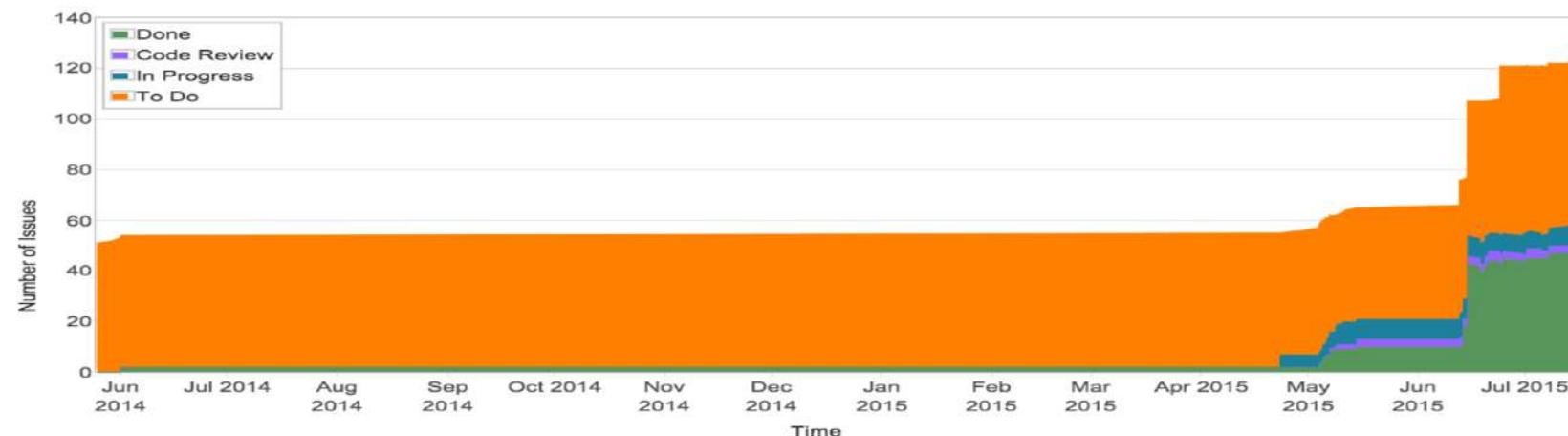
### ⓘ How to read this chart

Shows the statuses of issues over time. This helps you identify potential bottlenecks that need to be investigated.

[Hide this information](#)

### Cumulative Flow Diagram

25/May/14 to 16/Jul/15 (All Time) ▾ [Refine report](#) ▾



### Overview

Click and drag cursor across chart or chart overview to select date range (double-click overview to reset).



# Version Report

- The Version Report shows your team's progress towards the completion of a version.
- The Version Report also shows you the predicted Release Date, based on your team's average rate of progress (velocity) since the start of the version, and the estimated amount of work remaining.

- **Burn down Chart** – Track the total work remaining, also whether sprint is achieving the project goal or not.
- **Sprint Chart** – Track the work completed or pushed back to the backlog in each sprint.
- **Velocity Chart** – Track the amount of work completed from sprint to sprint.
- **Cumulative Flow Diagram** – Shows the statuses of issues over time. It helps to identify high-risk issues or unresolved important issues.
- **Version Report** – Track the projected release date for a version.
- **Epic Report** – Shows the progress towards completing an epic over a given time.
- **Control Chart** – Shows the cycle time for the product, its version or the sprint. It helps to identify whether data from the current process can be used to determine future performance.
- **Epic Burn Down** – Track the projected number of sprints required to complete the epic.
- **Release Burn Down** – Track the projected release date for a version. It helps to monitor whether the version will release on time, so mandatory action can be taken if work is falling behind.

ABC - Online Ho...  
Software project

Back to project

Reports

All reports

AGILE

ABC board  
Board

Burndown Chart

Burnup Chart

Sprint Report

Velocity Chart

Cumulative Flow Diagram

Version Report

Epic Report

Projects / ABC - Online Home Delivery Service



**Burndown Chart**

Track the total work remaining and project the likelihood of achieving the sprint goal. This helps your team manage its progress and respond accordingly.



**Burnup Chart**

Track the total scope independently from the total work done. This helps your team manage its progress and better understand the effect of scope change.



**Sprint Report**

Understand the work completed or pushed back to the backlog in each sprint. This helps you determine if your team is overcommitting or if there is excessive scope creep.



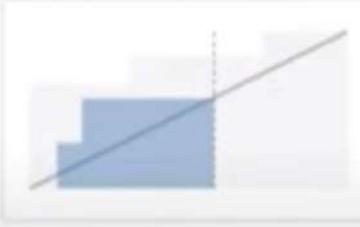
**Velocity Chart**

Track the amount of work completed from sprint to sprint. This helps you determine your team's velocity and estimate the work your team can realistically achieve in future sprints.



**Cumulative Flow Diagram**

Show the statuses of issues over time. This helps you identify potential bottlenecks that need to be investigated.



**Version Report**

Track the projected release date for a version. This helps you monitor whether the version will release on time, so you can take action if work is falling behind.

## Forecast & Management

- Following are the list of features of Forecast and Management type of reports.
- **Time Tracking Report** – Shows the original and current time estimates for issues in the current project. It can help to determine whether work is on track for those issues.
- **User Workload Report** – Shows the time estimates for all unresolved issues assigned to a user across projects. It helps to understand how much a user is occupied, whether overburdened or has less work.
- **Version Workload Report** – Displays how much outstanding work is remaining per user and per issue. It helps to understand the remaining work of a version.

## Even more metrics

- Good metrics aren't limited to the reports discussed above. For example, quality is an important metric for agile teams and there are a number of traditional metrics that can be applied to agile development:
- How many defects are found...
  - during development?
  - after release to customers?
  - by people outside of the team?
- How many defects are deferred to a future release?
- How many customer support requests are coming in?
- What is the percentage of automated test coverage?

## Even more metrics

- Agile teams should also look at release frequency and delivery speed.
- At the end of each sprint, the team should release software out to production.
- How often is that actually happening?
- Are most release builds getting shipped?
- In the same vein, how long does it take for the team to release an emergency fix out to production?
- Is release easy for the team or does it require heroics?

- Metrics are just one part in building a team's culture.
- They give quantitative insight into the team's performance and provide measurable goals for the team.
- While they're important, don't get obsessed.
- Listening to the team's feedback during [retrospectives](#) is equally important in growing trust across the team, quality in the product, and development speed through the release process.
- Use both the quantitative and qualitative feedback to drive change.

## **Chapter 8**

### **■ Design Concepts**

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 7/e*  
by Roger S. Pressman

**Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman**

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student

use.  
These Slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e*  
(McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

# Design

---

- Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Concept->Practice->software representations.
- Design is where you stand with a foot in two worlds – the world of technology and the world of people and human purpose – and you try to bring the two together.
- Design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system. It can be assessed for quality and improved before code is generated.

# Design

---

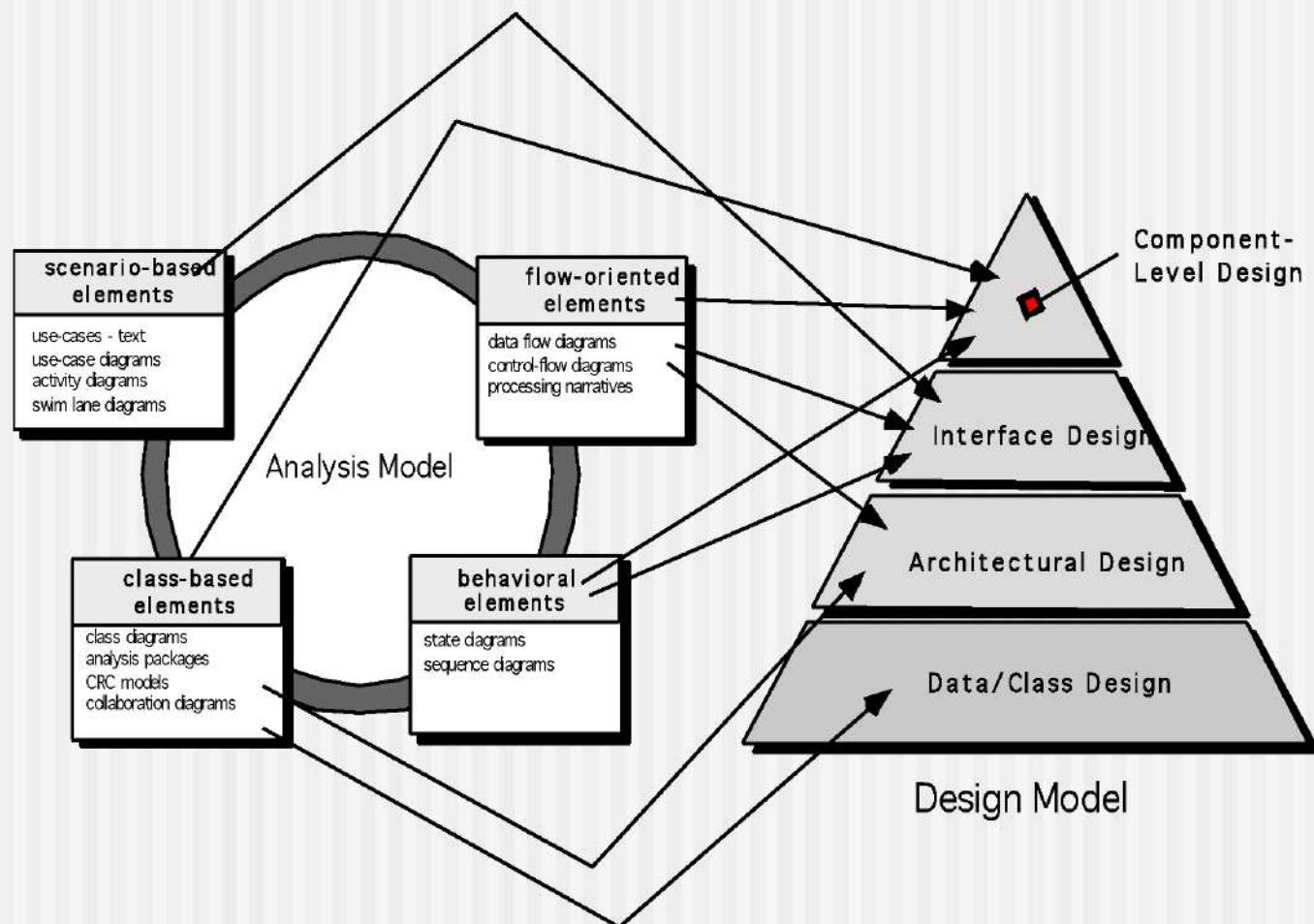
- Mitch Kapor, the creator of Lotus 1-2-3, presented a “software design manifesto” in *Dr. Dobbs Journal*. He said:
  - Good software design should exhibit:
  - *Firmness*: A program should not have any bugs that inhibit its function.
  - *Commodity*: A program should be suitable for the purposes for which it was intended.
  - *Delight*: The experience of using the program should be pleasurable one.

# Design

---

- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used.
- After requirement modeling, it is the last action within the modeling activity and sets the stage for construction (code generation and testing).
- Elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design.

# Requirements Model -> Design Model



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e*  
(McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

# Design

---

- The **data/class design** transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture.
- The **architectural design** (framework of a computer-based system) defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

# Design

---

- The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (data / control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information needed.
- The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

# Design and Quality

---

- The importance of design can be stated with a single word – **quality**. It is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software products or system.
- Software design serves as the **foundation** for all the software engineering and support activities that follow.
- Without design, you risk building an **unstable** system – one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

# Design Process

---

- It is an **iterative** process through which requirements are translated into a “blueprint” for constructing the software.
- Initially, the **blueprint** depicts a holistic view of software. That is the design is represented at a high level of abstraction- a level that can be directly traced into the specific system objective and more detailed data, functional, and behavioral requirements.
- As design iteration occur, **subsequent refinement** leads to design representations at much lower level of abstraction with subtle connection to requirements.

# Design Process: Software Design Characteristics

---

Three characteristics that serve as a guide for the evaluation of a good design.

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

# Technical Criteria for Good Design

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
  1. For smaller systems, design can sometimes be developed linearly.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

# Design Principles

---

- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

*From Davis [DAV95]*

# Design Quality Attributes (FURPS)

- **Functionality**: evaluate the feature set and capabilities of the program, the generality of the functions that are delivered , and the security of the overall system.
- **Usability** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- **Reliability** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.
- **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines the ability to extend the program, adaptability, serviceability, maintainability, testability, compatibility, configurability.

*From Hewlett-Packard [Gra87]]*

# Fundamental Concepts

---

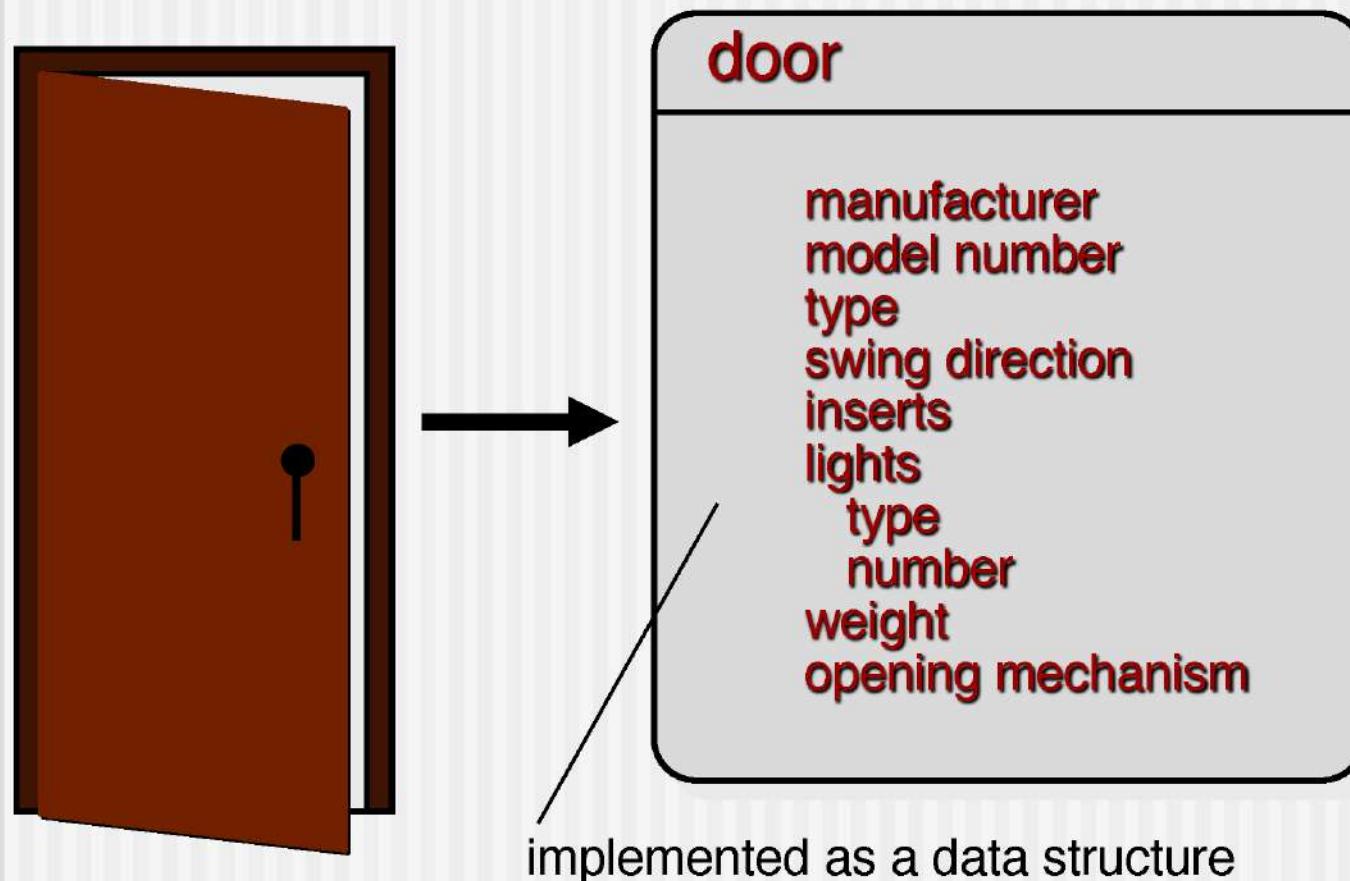
- A set of fundamental software design concepts has evolved over the history of software engineering. They span both traditional and object-oriented software development.
- M.A. Jackson [Jac75] once said: “the beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right.”
- Fundamental software design concepts provide the **necessary framework** for “getting it right”.

# Fundamental Concepts

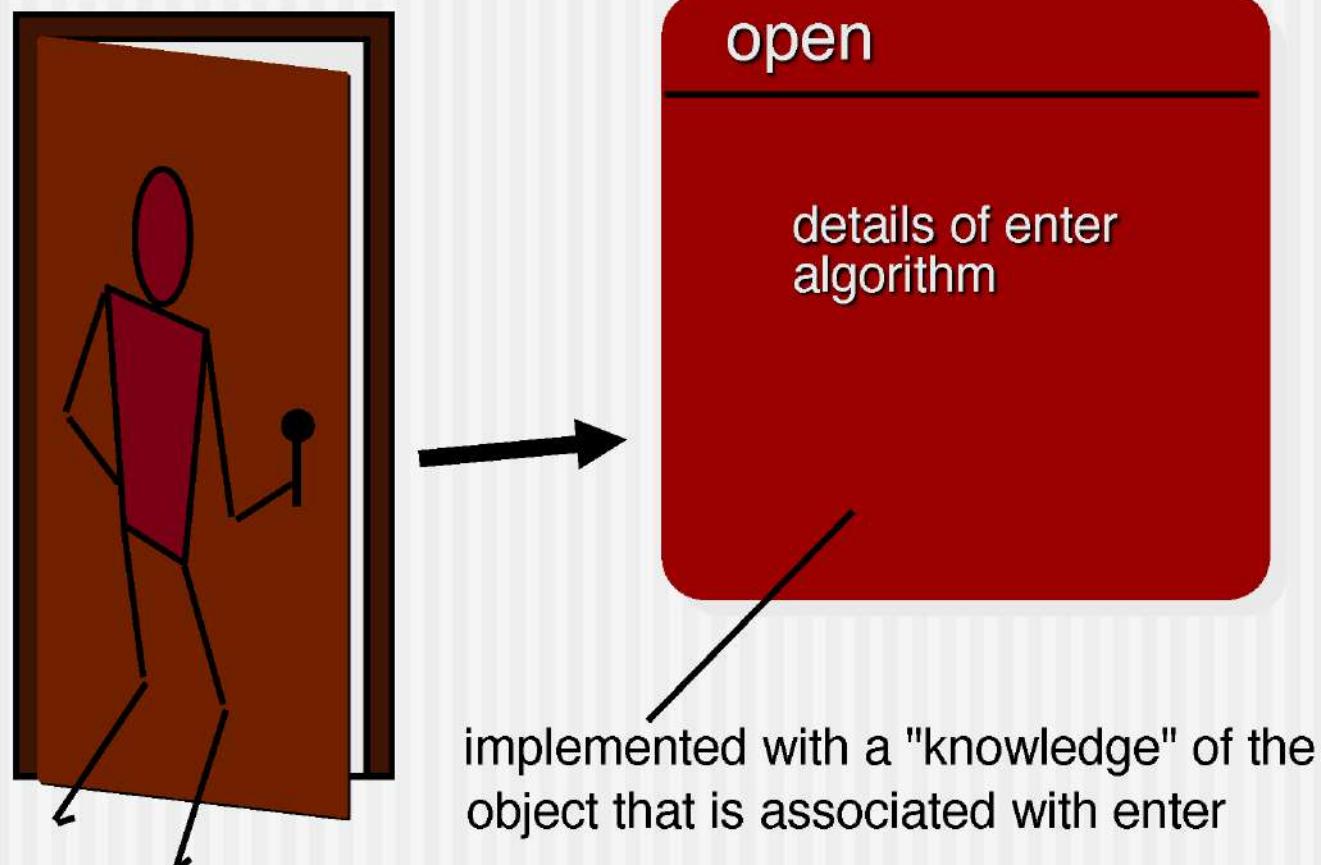
---

- **Abstraction**—data, procedure, control
- **Architecture**—the overall structure of the software
- **Patterns**—”conveys the essence” of a proven design solution
- **Separation of concerns**—any complex problem can be more easily handled if it is subdivided into pieces
- **Modularity**—compartmentalization of data and function
- **Hiding**—controlled interfaces
- **Functional independence**—single-minded function and low coupling
- **Refinement**—elaboration of detail for all abstractions
- **Aspects**—a mechanism for understanding how global requirements affect design
- **Refactoring**—a reorganization technique that simplifies the design
- **OO design concepts**—Appendix II
- **Design Classes**—provide design detail that will enable analysis classes to be implemented

# Data Abstraction



# Procedural Abstraction



# Architecture

---

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]

What properties should be specified?

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

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

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

# Pattern

---

- Brad Appleton defines design pattern in the following manner: A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concern.
- The intent of each design pattern is to provide a description that enables a designer to determine
  - (1) whether the pattern is applicable to the current work,
  - (2) whether the pattern can be reused (hence, saving design time),
  - (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

# Design Pattern Template

---

## *Design Pattern Template*

**Pattern name**—describes the essence of the pattern in a short but expressive name

**Intent**—describes the pattern and what it does

**Also-known-as**—lists any synonyms for the pattern

**Motivation**—provides an example of the problem

**Applicability**—notes specific design situations in which the pattern is applicable

**Structure**—describes the classes that are required to implement the pattern

**Participants**—describes the responsibilities of the classes that are required to implement the pattern

**Collaborations**—describes how the participants collaborate to carry out their responsibilities

**Consequences**—describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

**Related patterns**—cross-references related design patterns

# Separation of Concerns

---

- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

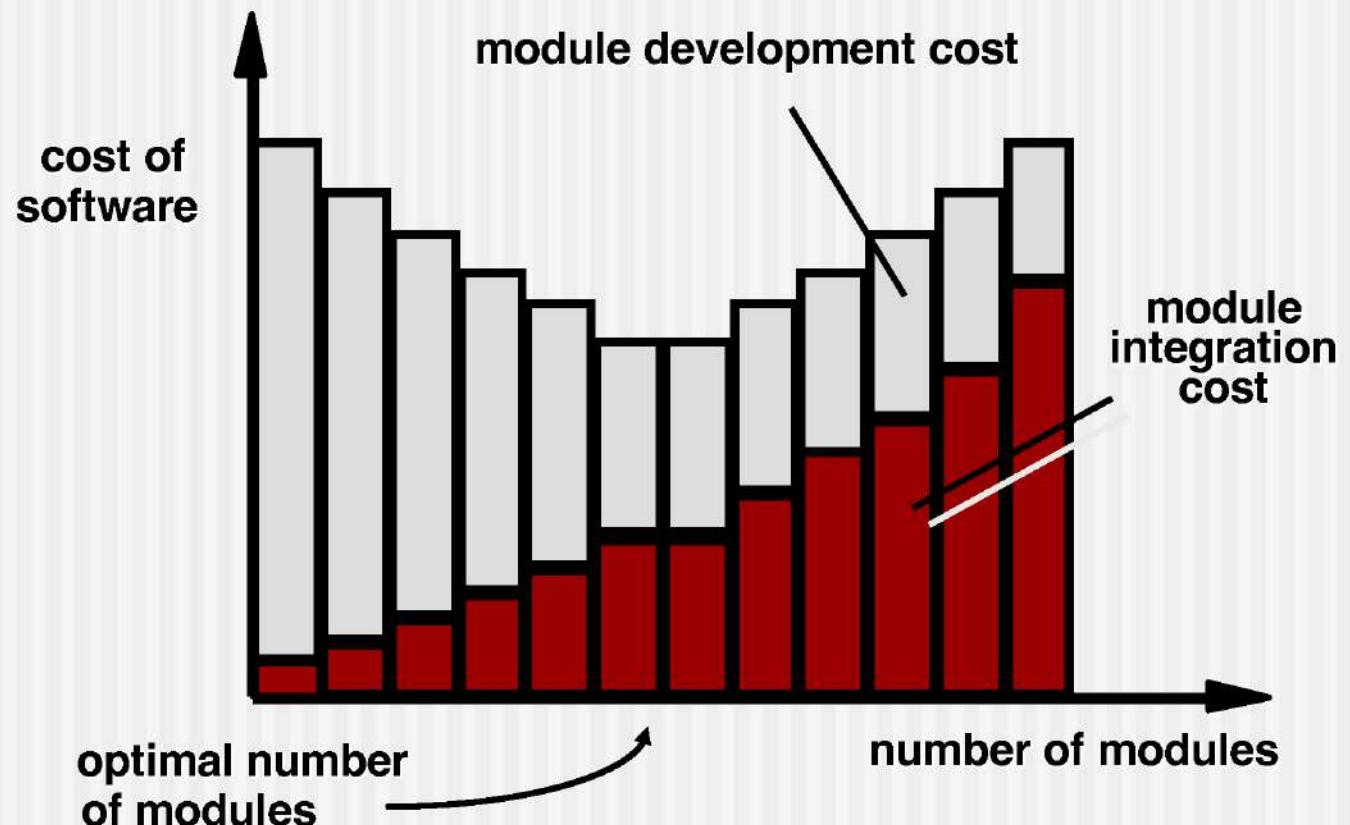
# Modularity

---

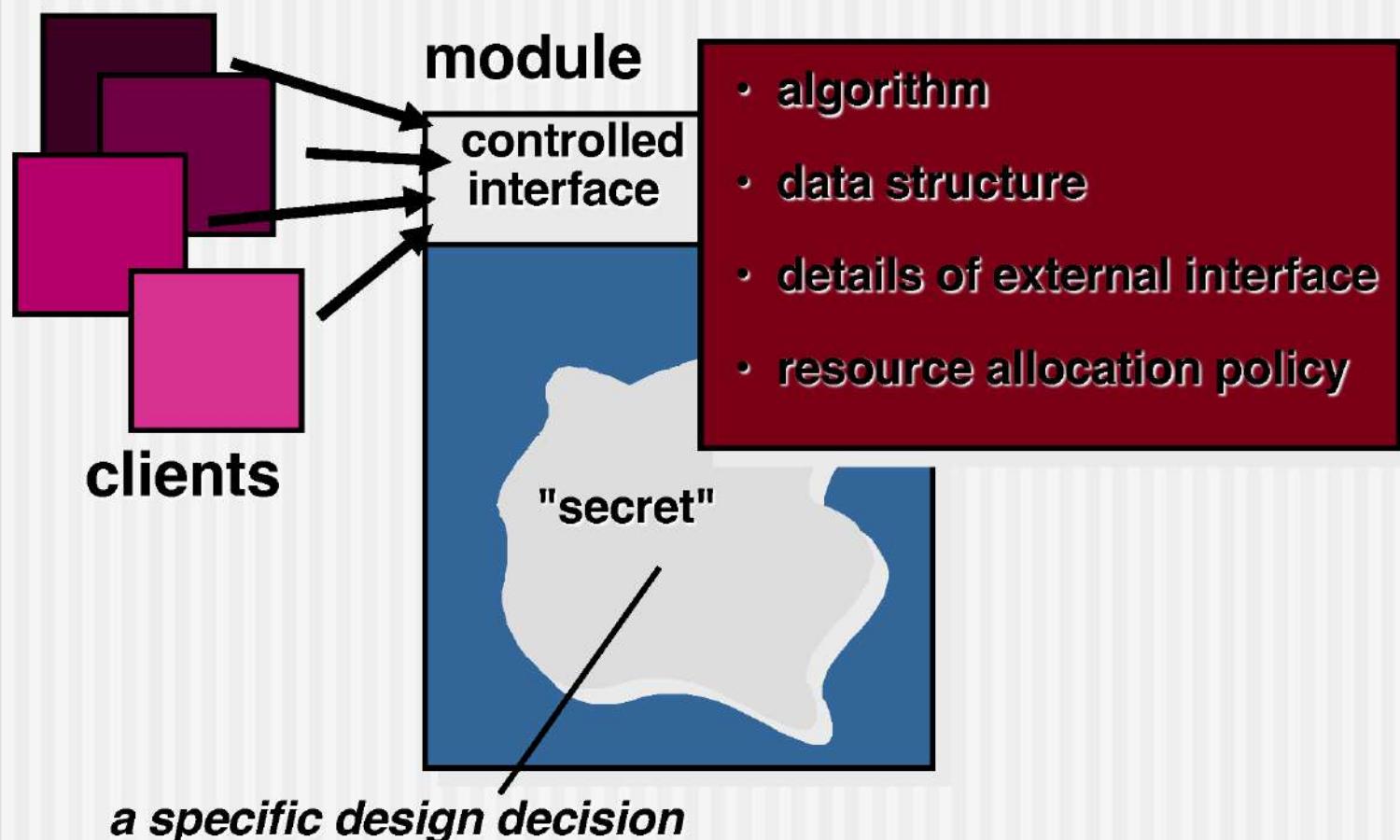
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

# Modularity: Trade-offs

*What is the "right" number of modules for a specific software design?*



# Information Hiding

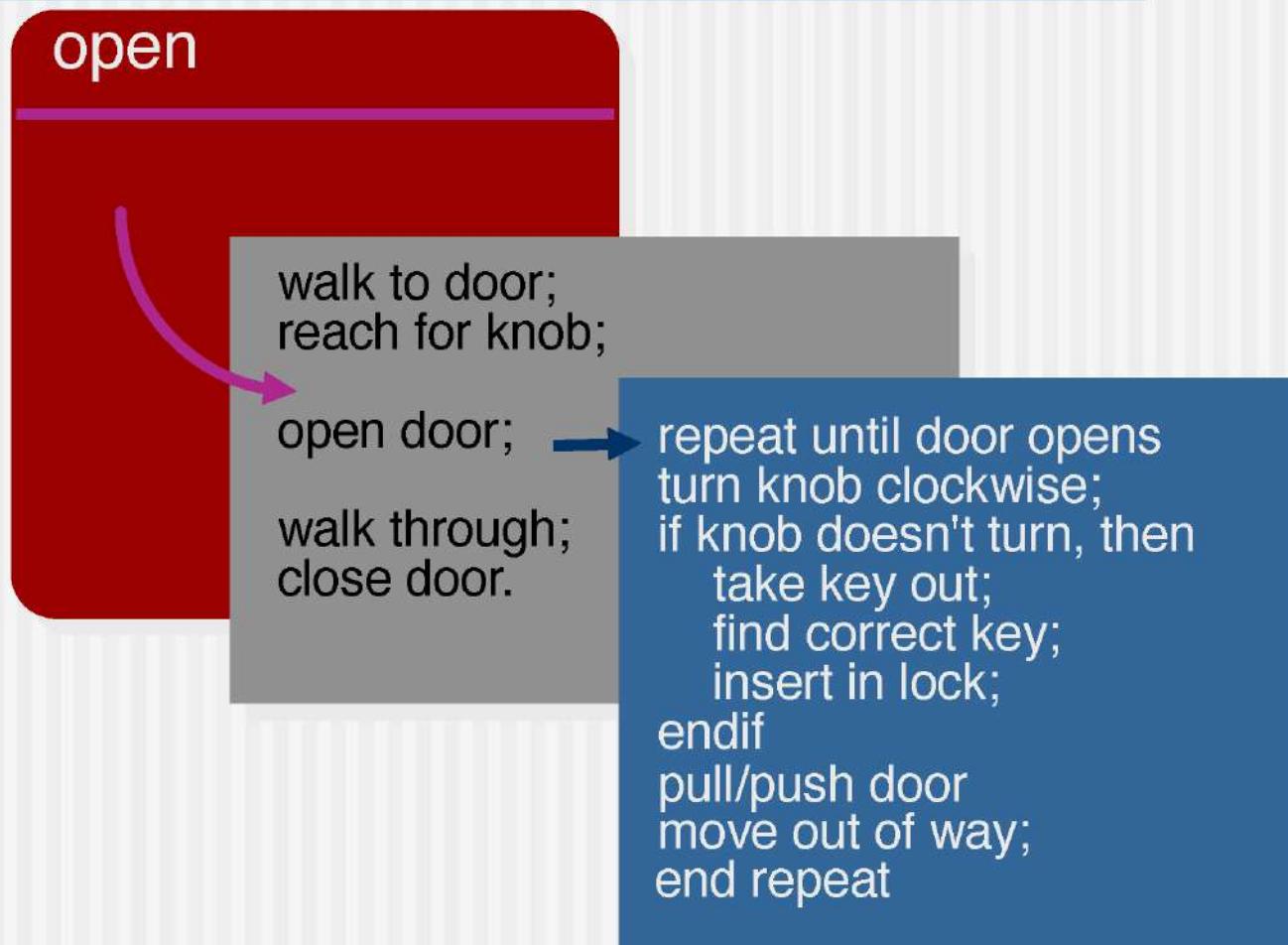


# Why Information Hiding?

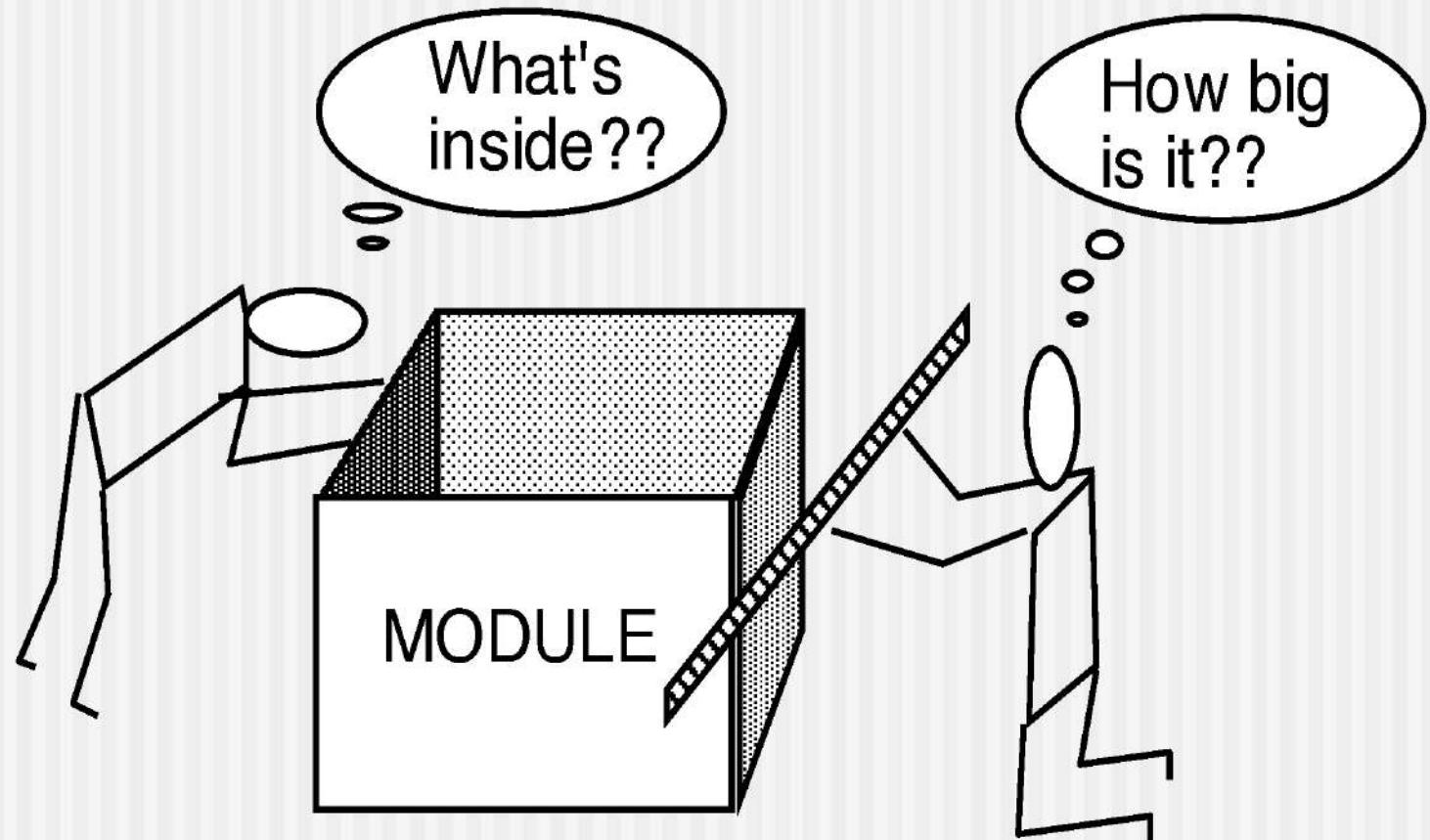
---

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

# Stepwise Refinement



# Sizing Modules: Two Views



# Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

# Aspects

---

- Ideally, a requirements model can be organized in a way that allows you to isolate each concern so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.
- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

# Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet**. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using SafeHomeAssured.com*. This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A\** is a design representation for requirement *A* and *B\** is a design representation for requirement *B*. Therefore, *A\** and *B\** are representations of concerns, and *B\** *cross-cuts* *A\**.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B\**, of the requirement, *a registered user must be validated prior to using SafeHomeAssured.com*, is an aspect of the *SafeHome* WebApp.

# Refactoring

---

- An important design activity suggested for many agile methods, it is reorganization technique that simplifies the design of a component.
- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

# OO Design Concepts

---

- Requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction is high.
- As the design model evolves, you will define a set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- Five different types of design classes, each representing a different layer of the design architecture:
  - User interface classes
  - Business domain classes
  - Process classes
  - Persistent classes
  - System classes

# Design Classes

---

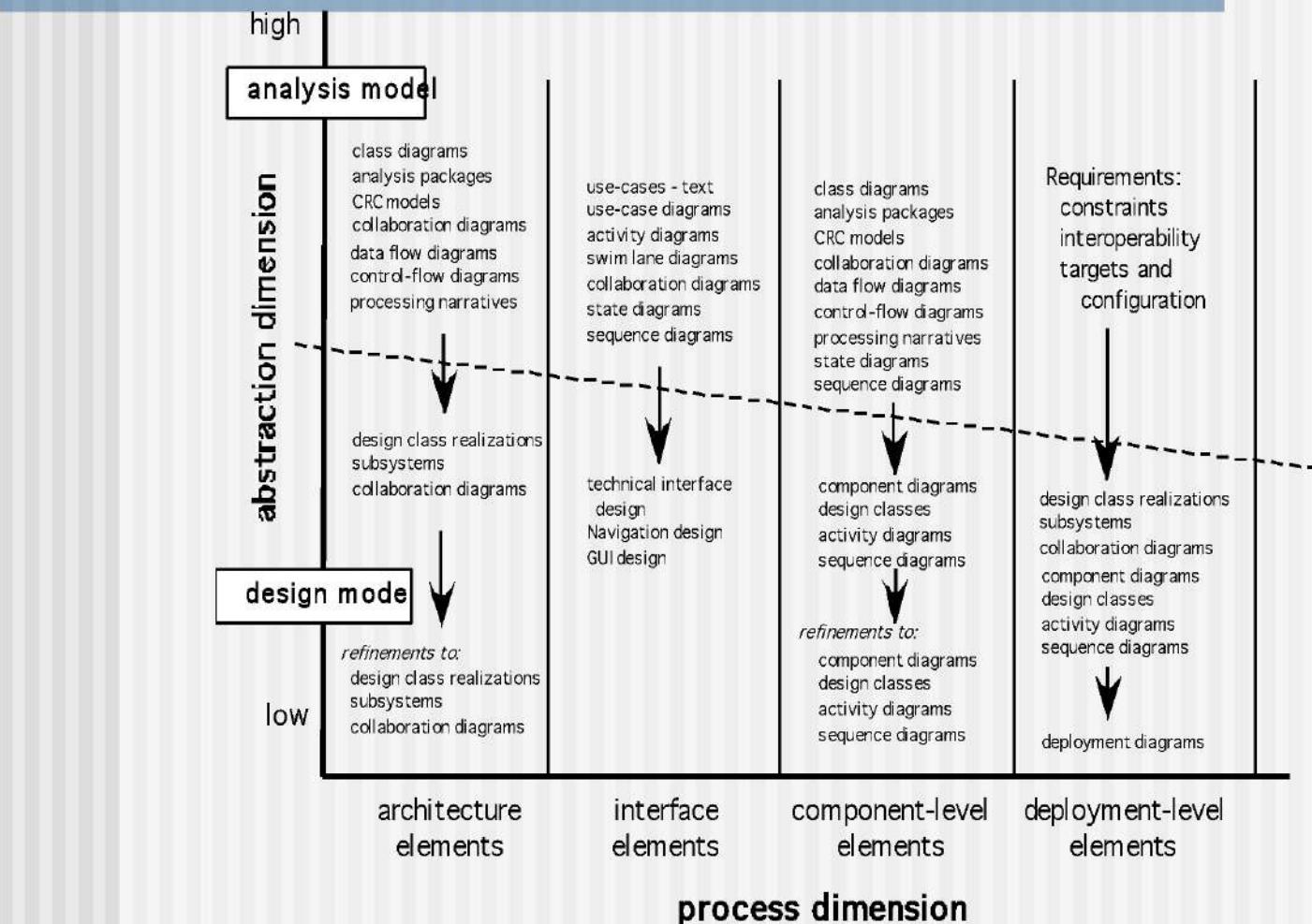
- Design classes in three big categories
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

# Design Classes

---

- Analysis classes are refined during design to become **entity classes**
- **Boundary classes** are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- **Controller classes** are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

# The Design Model



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e*  
(McGraw-Hill, 2009) Slides copyright 2009 by Roger Pressman.

# Design Model Elements

---

- **Data elements**
  - Data model --> data structures
  - Data model --> database architecture
- **Architectural elements**
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles” (Chapters 9 and 12)
- **Interface elements**
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- **Component elements**
- **Deployment elements**

# Architectural Elements

---

- The architectural model [Sha96] is derived from three sources:
  - **information about the application domain** for the software to be built;
  - **specific requirements model elements** such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - **the availability of architectural patterns** (Chapter 12) and **styles** (Chapter 9).

# Interface Elements

- How information flows in and out of the system and how components communicate and collaborate.
- Three major elements:
  - User interface
  - External interface to other system, device, networks or producers or consumers of information
  - Internal interfaces between components.

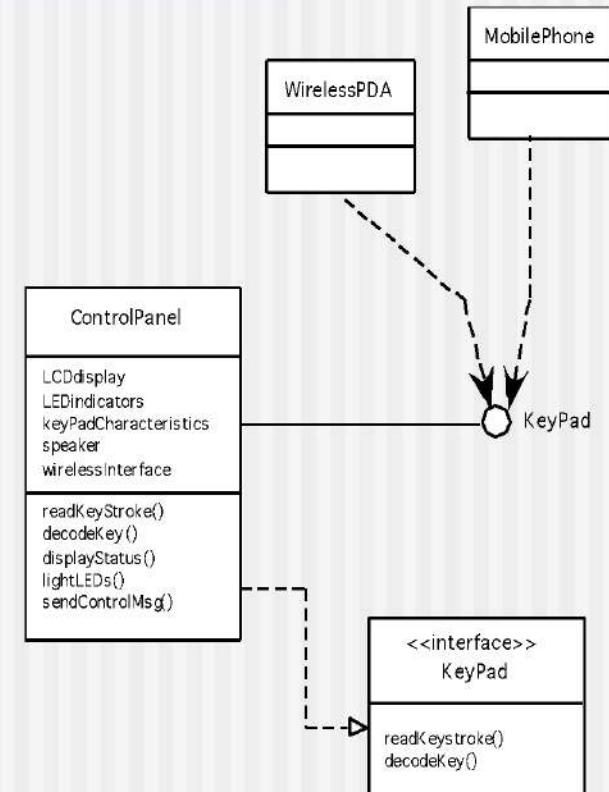


Figure 9.6 UML interface representation for `ControlPanel`

# Component Elements



- Describe the internal details of each software components. It defines data structure for all local data objects and algorithmic details for all processing that occurs within a component and an interface that allows access to all component operations.
- Within the context of object-oriented SE, a component is represented in UML diagram as shown above. SensorManagement performs all functions associated with sensors including monitoring and configuring them.

# Deployment Elements

- How software functionality and subsystems will be allocated within the physical computing environment that will support the software.
  - E.g. Safehome product are configured to operate within three primary computing environments- a home-based PC, the safehome control panel, and a server housed at CPI corp.

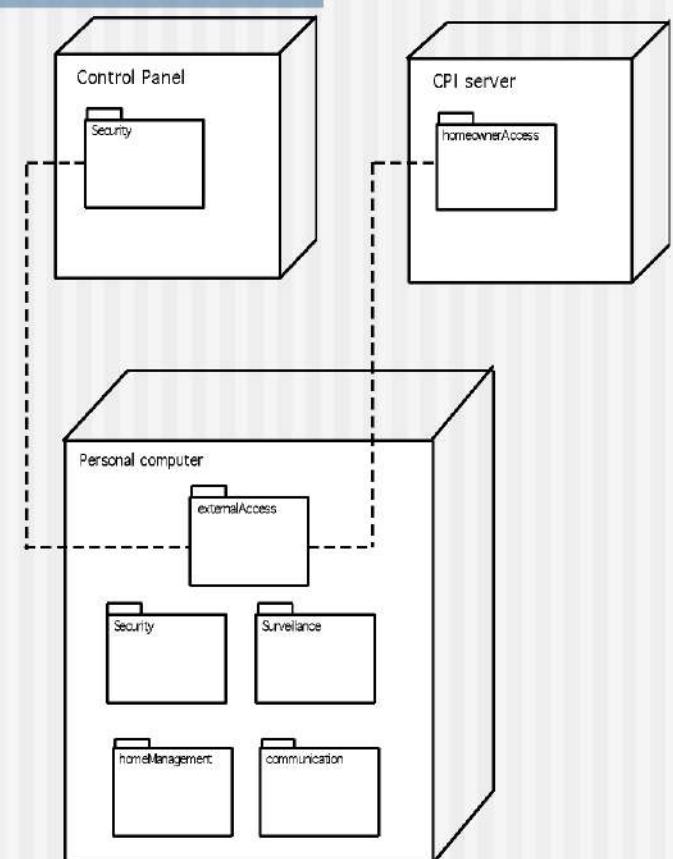


Figure 9.8 UML deployment diagram for *SafeHome*

# Design Engineering

*based on*

Chapter 9 *Software Engineering: A Practitioner's Approach, 6/e*

copyright © 1996, 2001, 2005

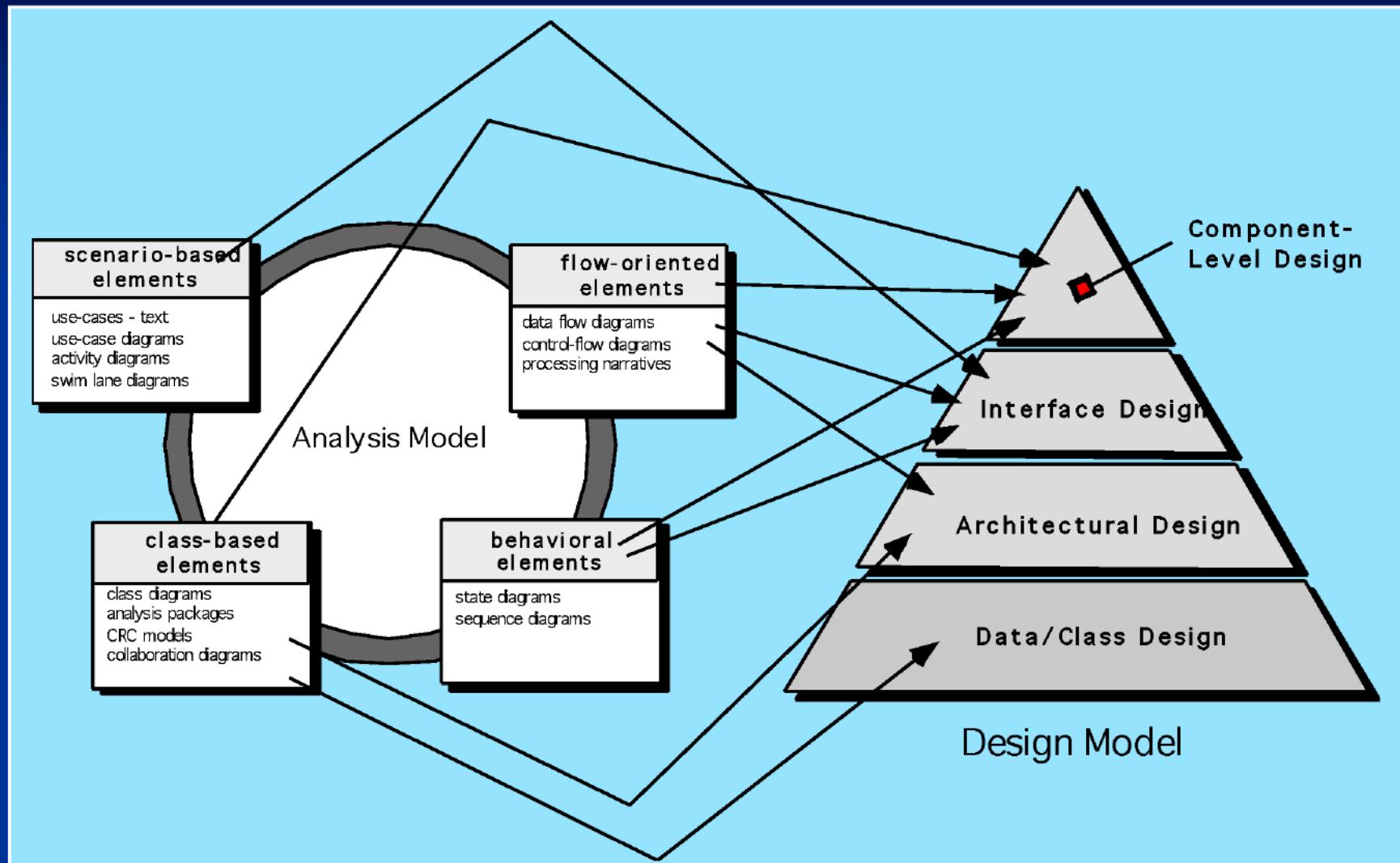
R.S. Pressman & Associates, Inc.

## **For University Use Only**

May be reproduced ONLY for student use at the university level  
when used in conjunction with *Software Engineering: A Practitioner's Approach*.

Any other reproduction or use is expressly prohibited.

# Analysis Model -> Design Model



These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

# Design and Quality

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

# Quality Guidelines

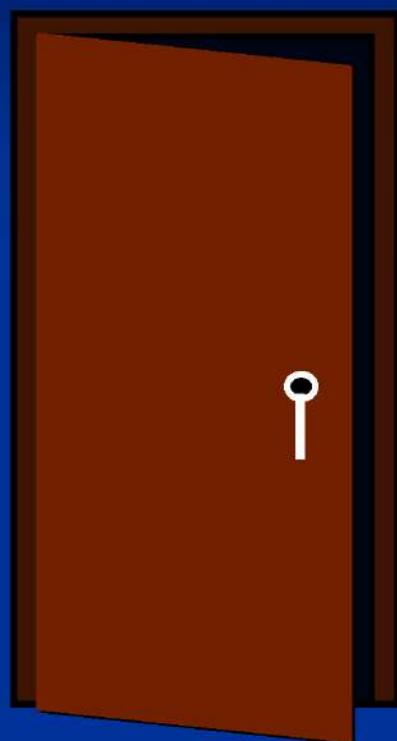
- A design should exhibit an architecture that
  - (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (3) can be implemented in an evolutionary fashion
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to components that exhibit independent functional characteristics.
- A design should be represented using a notation that effectively communicates its meaning.

# Fundamental Concepts

- abstraction—data, procedure, control
- architecture—the overall structure of the software
- modularity—compartmentalization of data and function
- Functional independence—single-minded function and low coupling
- hiding—controlled interfaces
- refinement—elaboration of detail for all abstractions
- Refactoring—a reorganization technique that simplifies the design

abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

# Data Abstraction

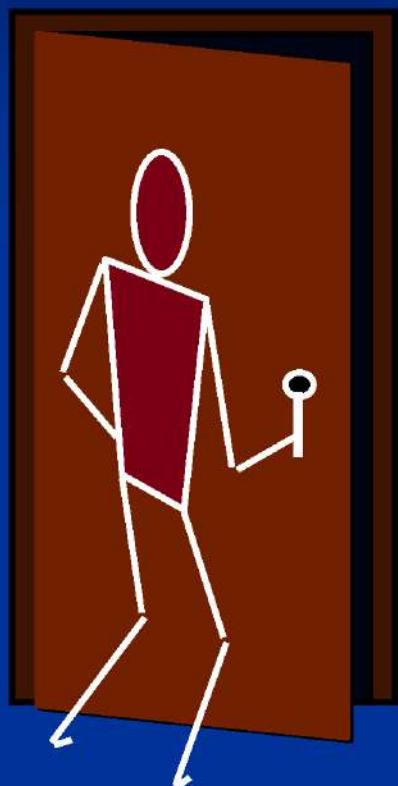


door

manufacturer  
model number  
type  
swing direction  
inserts  
lights  
type  
number  
weight  
opening mechanism

implemented as a data structure

# Procedural Abstraction



implemented with a "knowledge" of the object that is associated with enter

abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

# Architecture

**“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]**

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

**Extra-functional properties.** achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

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

abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

# Modular Design

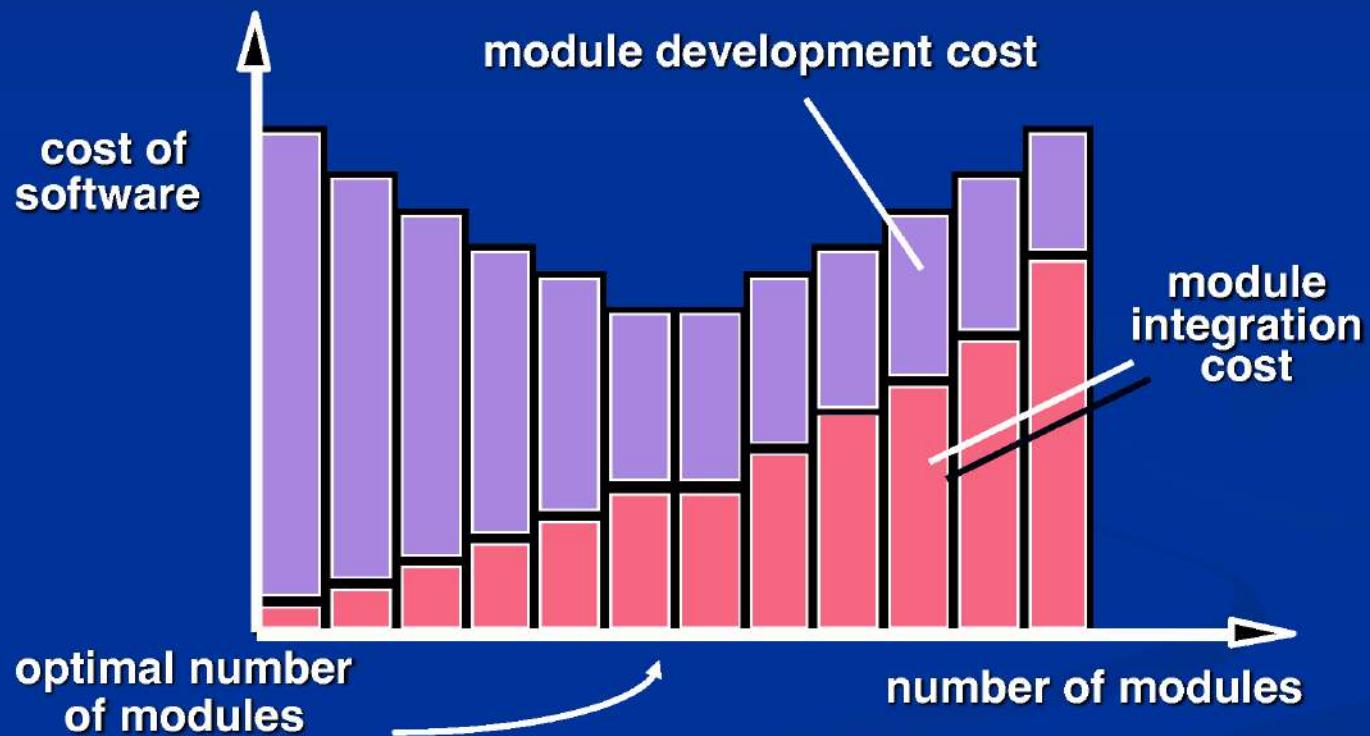
*easier to build, easier to change, easier to fix ...*



These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

# Modularity: Trade-offs

*What is the "right" number of modules for a specific software design?*



abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

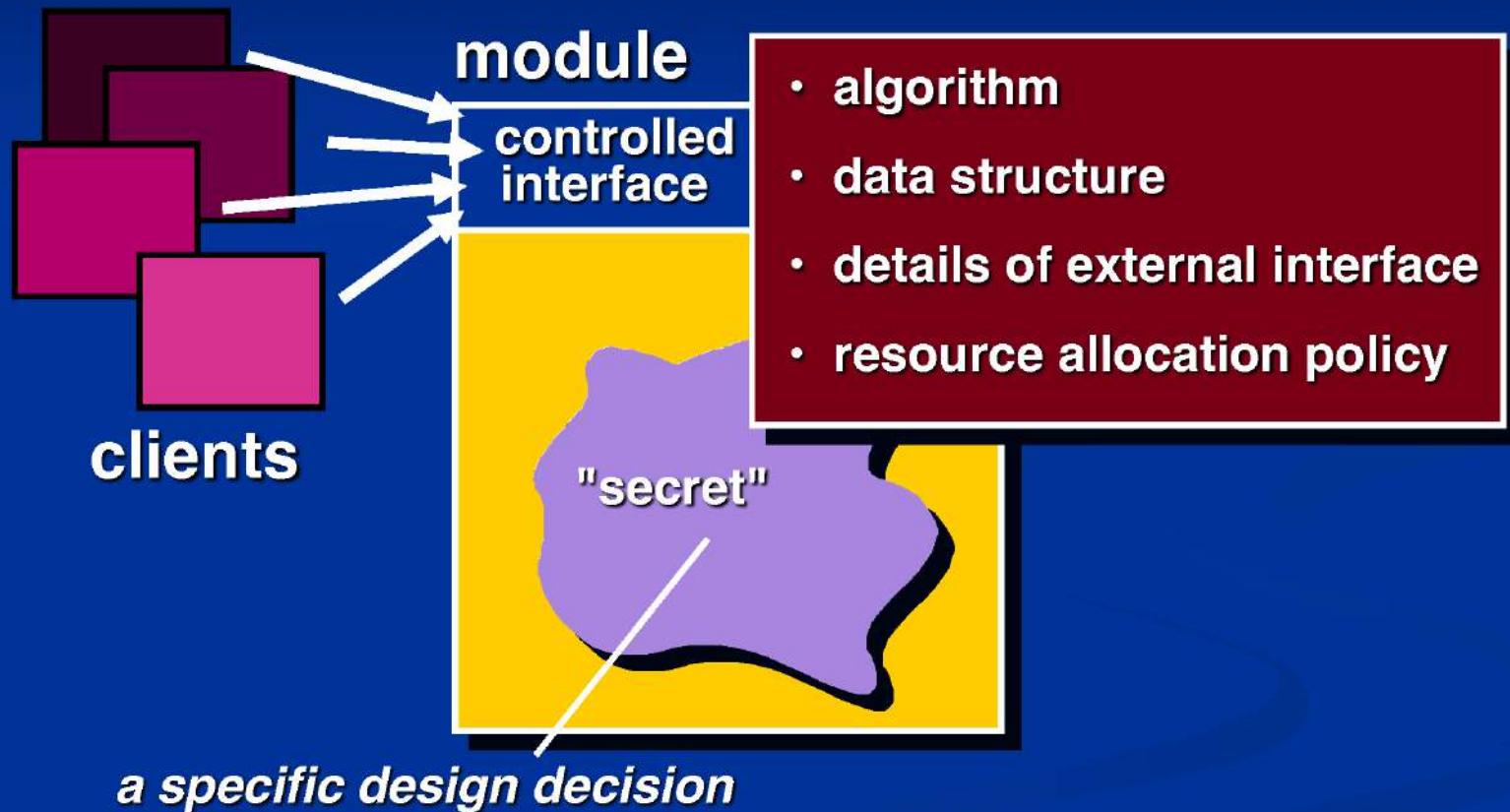
# Functional Independence

**COHESION** - the degree to which a module performs one and only one function.

**COUPLING** - the degree to which a module is "connected" to other modules in the system.

abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

# Information Hiding

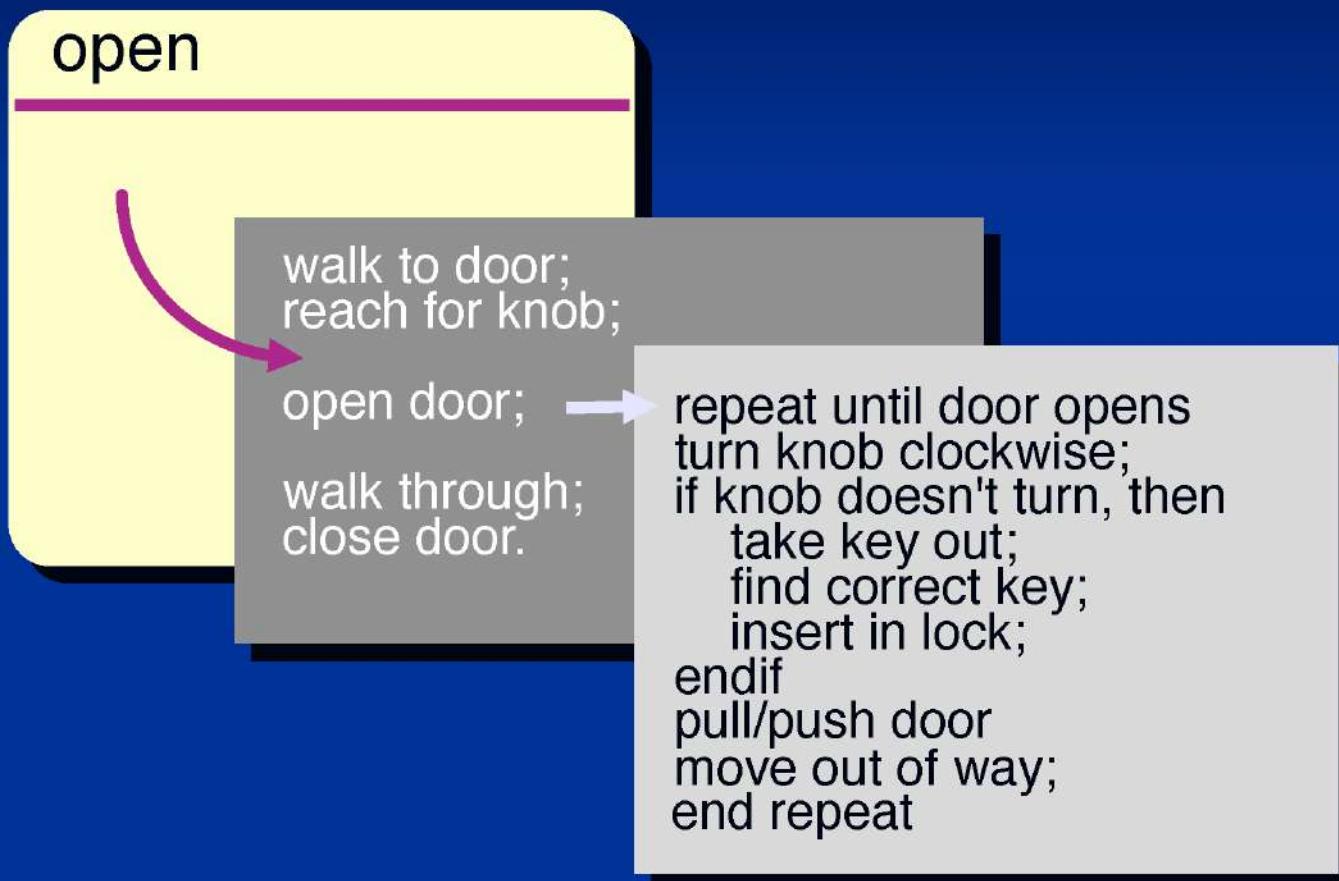


# Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

# Stepwise Refinement



abstraction  
architecture  
modularity  
functional independence  
hiding  
refinement  
refactoring

# Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

# OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Design classes

Inheritance

Messages

Polymorphism

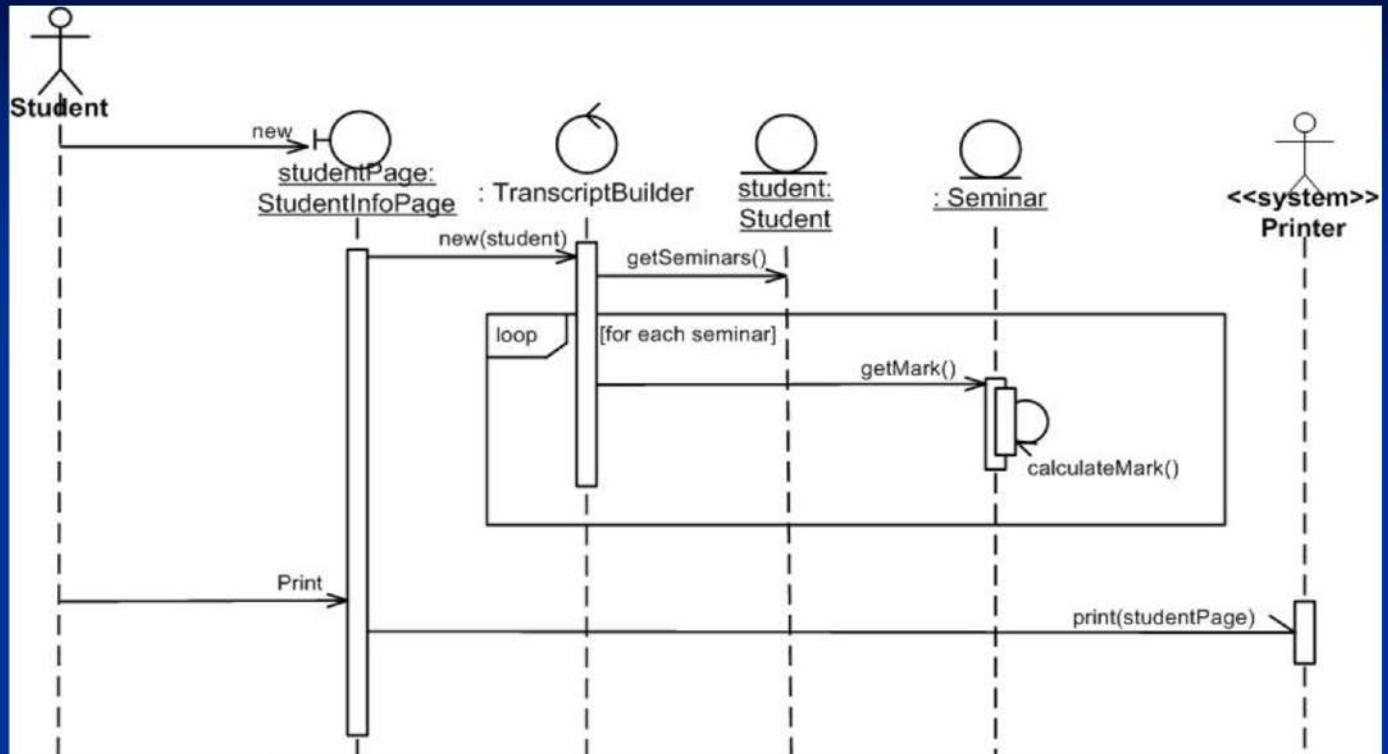
# Design Classes

- Analysis classes are refined during design to become **entity classes**
- **Boundary classes** are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- **Controller classes** are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

# What can be in the top boxes?

(<http://www.agilemodeling.com/artifacts/sequenceDiagram.htm>)

## Outputting transcripts



**Boundary/interface elements:** software elements such as screens, reports, HTML pages, or system interfaces that actors interact with.

**Control/process elements (controllers):** These serve as the glue between boundary elements and entity elements, implementing the logic required to manage the various elements and their interactions. Often implemented using objects, but simple ones using methods of an entity or boundary class.

 These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by P.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

**Entity elements**

Design classes

Inheritance

Messages

Polymorphism

# Inheritance

## ■ Design options:

- The class can be designed and built from scratch. That is, if inheritance is not used.
- The class hierarchy can be searched to determine if a class higher in the hierarchy (a superclass) contains most of the required attributes and operations. The new class inherits from the superclass and additions may then be added, as required.
- The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
- Characteristics of an existing class can be overridden and different versions of attributes or operations are implemented for the new class.

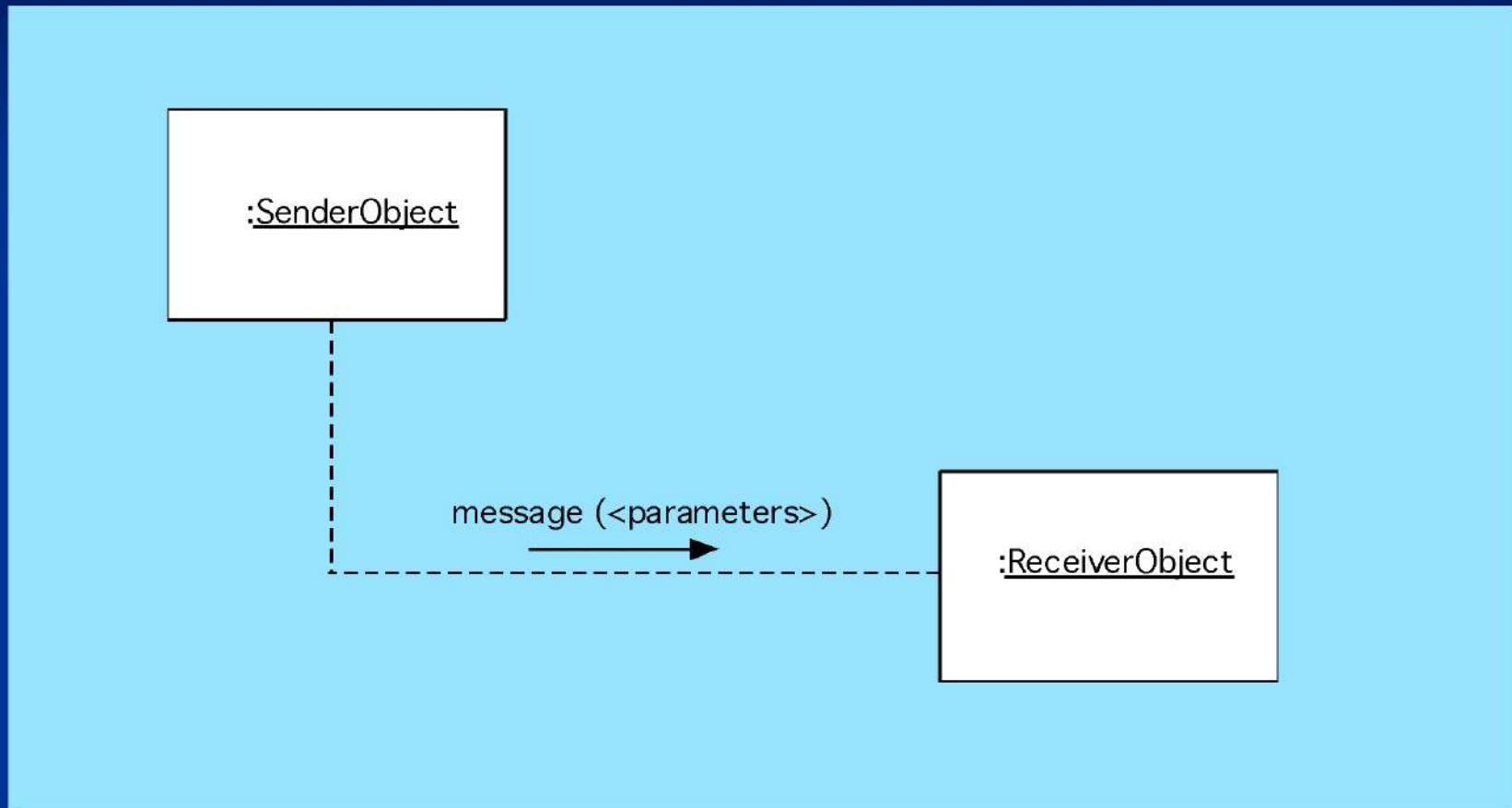
Design classes

Inheritance

Messages

Polymorphism

# Messages



Design classes

Inheritance

Messages

Polymorphism

# Polymorphism

Conventional approach ...

case of graphtype:

```
if graphtype = linegraph then DrawLineGraph (data);
if graphtype = piechart then DrawPieChart (data);
if graphtype = histogram then DrawHisto (data);
if graphtype = kiviat then DrawKiviat (data);
```

end case;

All of the graphs become subclasses of a general class called `graph`. Using a concept called **overloading** [TAY90], each subclass defines an operation called `draw`. An object can send a `draw` message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own `draw` operation to create the appropriate graph.

graphtype draw

# Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles” (Chapter 10)
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

# Interface Elements

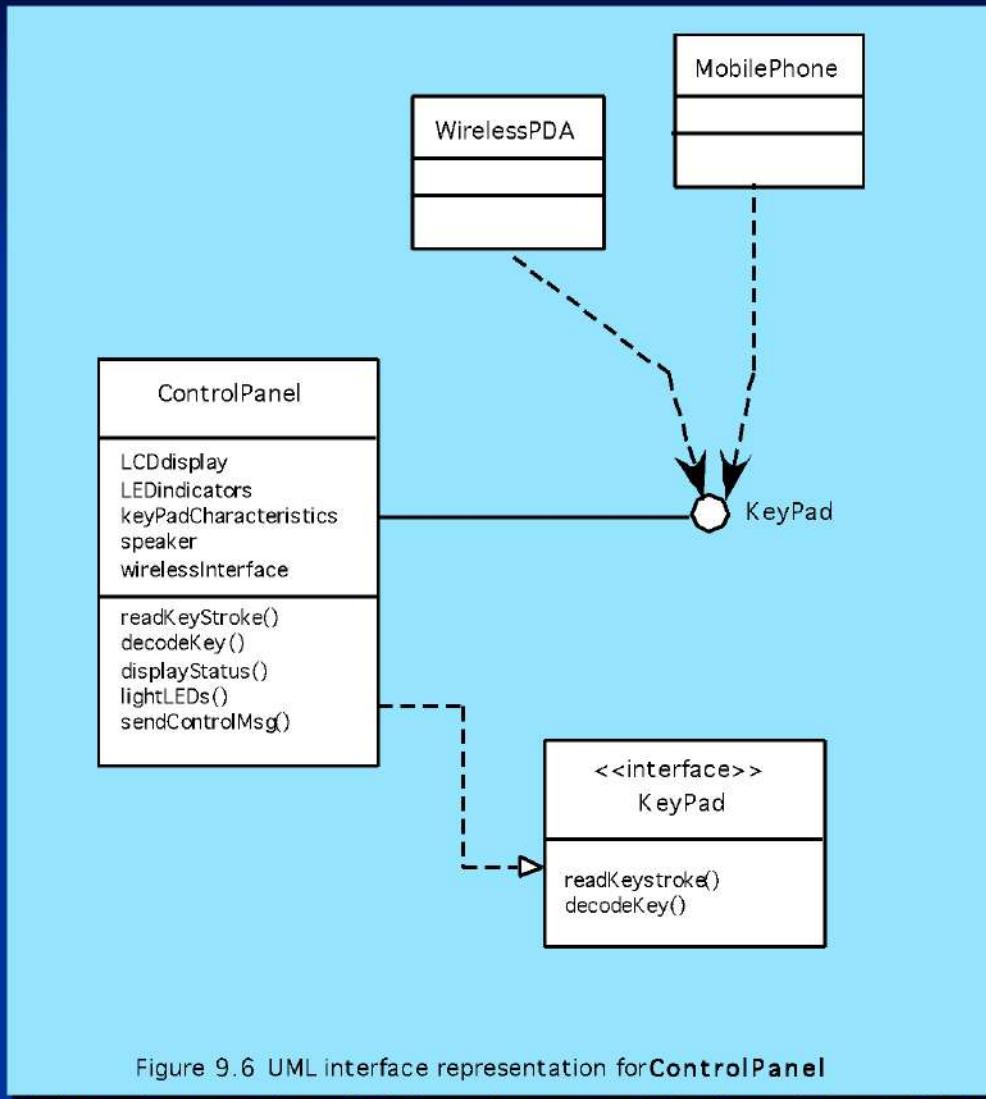
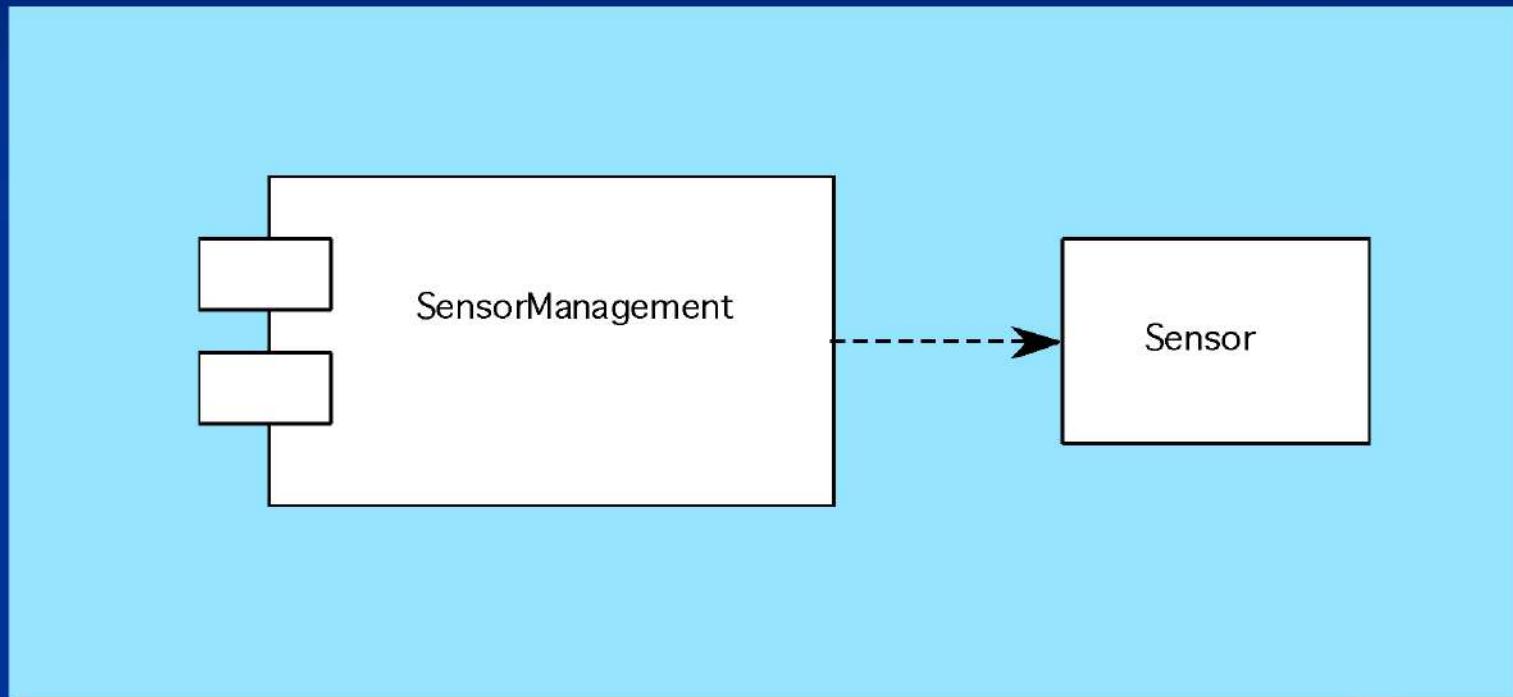


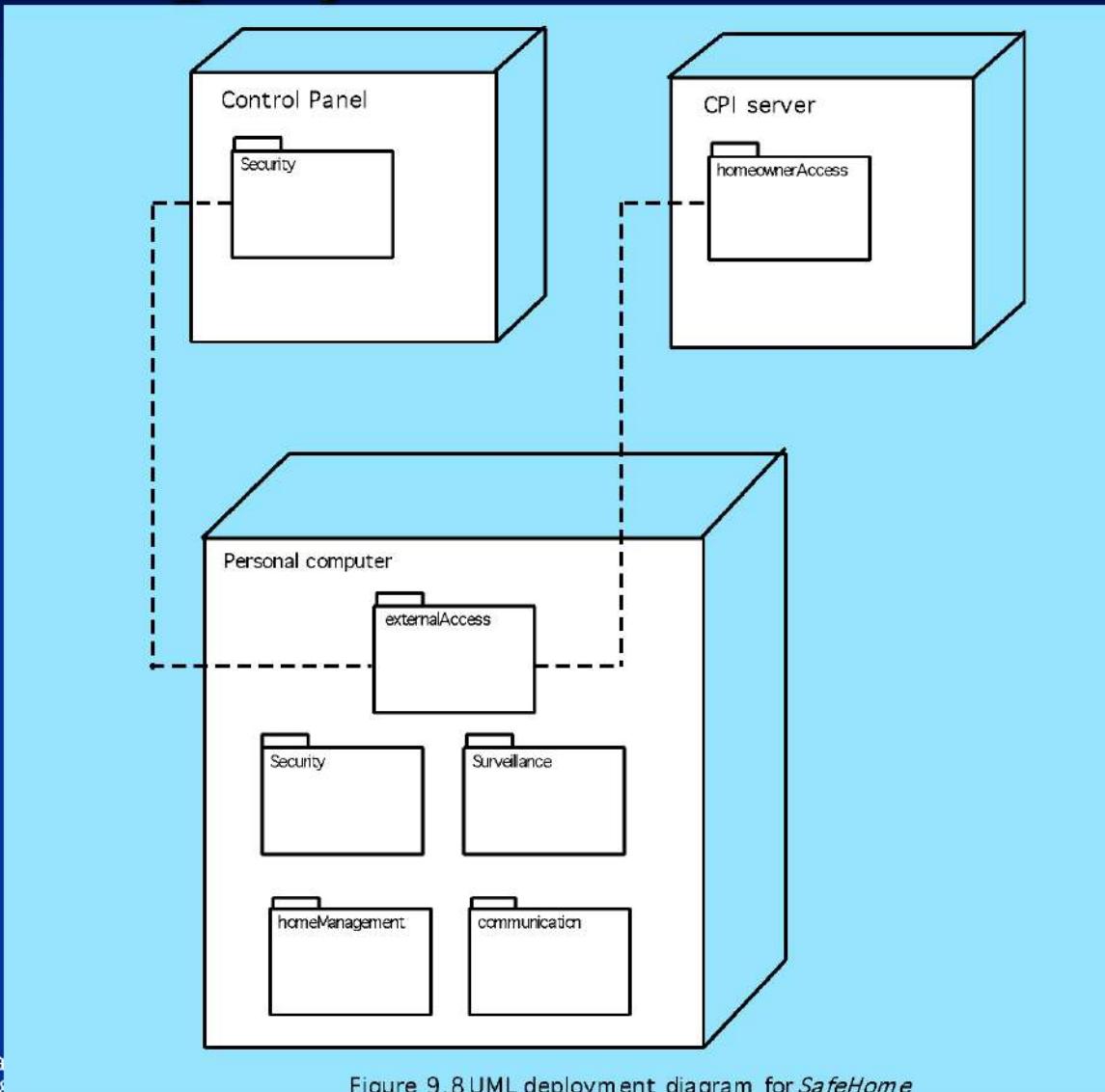
Figure 9.6 UML interface representation for **ControlPanel**

These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

# Component Elements



# Deployment Elements



These courseware materials are  
used with permission by R.S. Pressman &

Figure 9.8 UML deployment diagram for *SafeHome*

ed with

# Omitted Slides

These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

# Patterns

## *Design Pattern Template*

**Pattern name**—describes the essence of the pattern in a short but expressive name

**Intent**—describes the pattern and what it does

**Also-known-as**—lists any synonyms for the pattern

**Motivation**—provides an example of the problem

**Applicability**—notes specific design situations in which the pattern is applicable

**Structure**—describes the classes that are required to implement the pattern

**Participants**—describes the responsibilities of the classes that are required to implement the pattern

**Collaborations**—describes how the participants collaborate to carry out their responsibilities

**Consequences**—describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

**Related patterns**—cross-references related design patterns

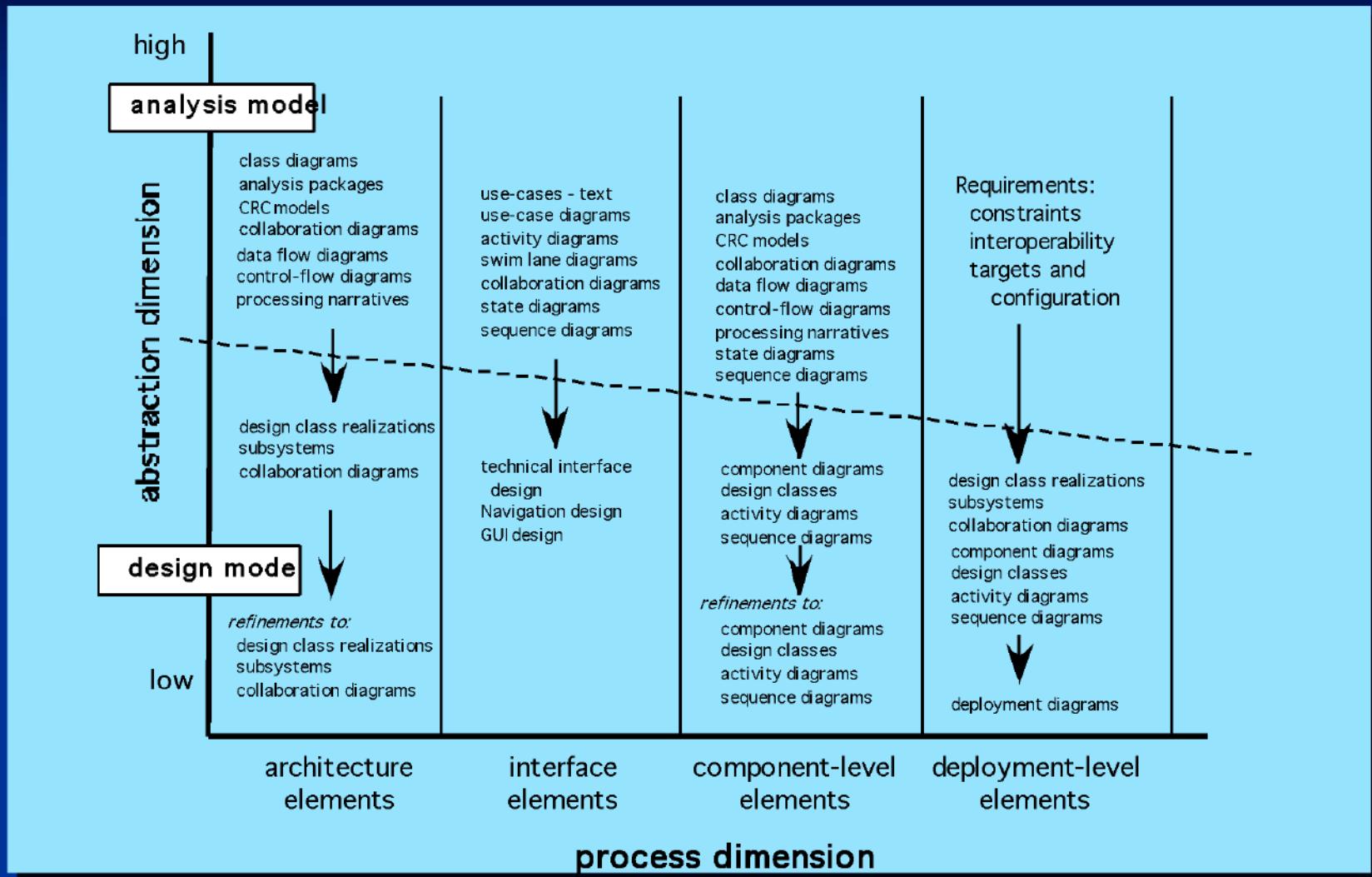
# Design Patterns

- The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution
- A description of a design pattern may also consider a set of design forces.
  - *Design forces* describe non-functional requirements (e.g., ease of maintainability, portability) associated the software for which the pattern is to be applied.
- The **pattern characteristics** (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems.

# Frameworks

- A **framework** is not an architectural pattern, but rather a skeleton with a collection of “plug points” (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain.
- Gamma et al note that:
  - Design patterns are more abstract than frameworks.
  - Design patterns are smaller architectural elements than frameworks
  - Design patterns are less specialized than frameworks

# The Design Model



These courseware materials are to be used in conjunction with *Software Engineering: A Practitioner's Approach*, 6/e and are provided with permission by R.S. Pressman & Associates, Inc., copyright © 1996, 2001, 2005

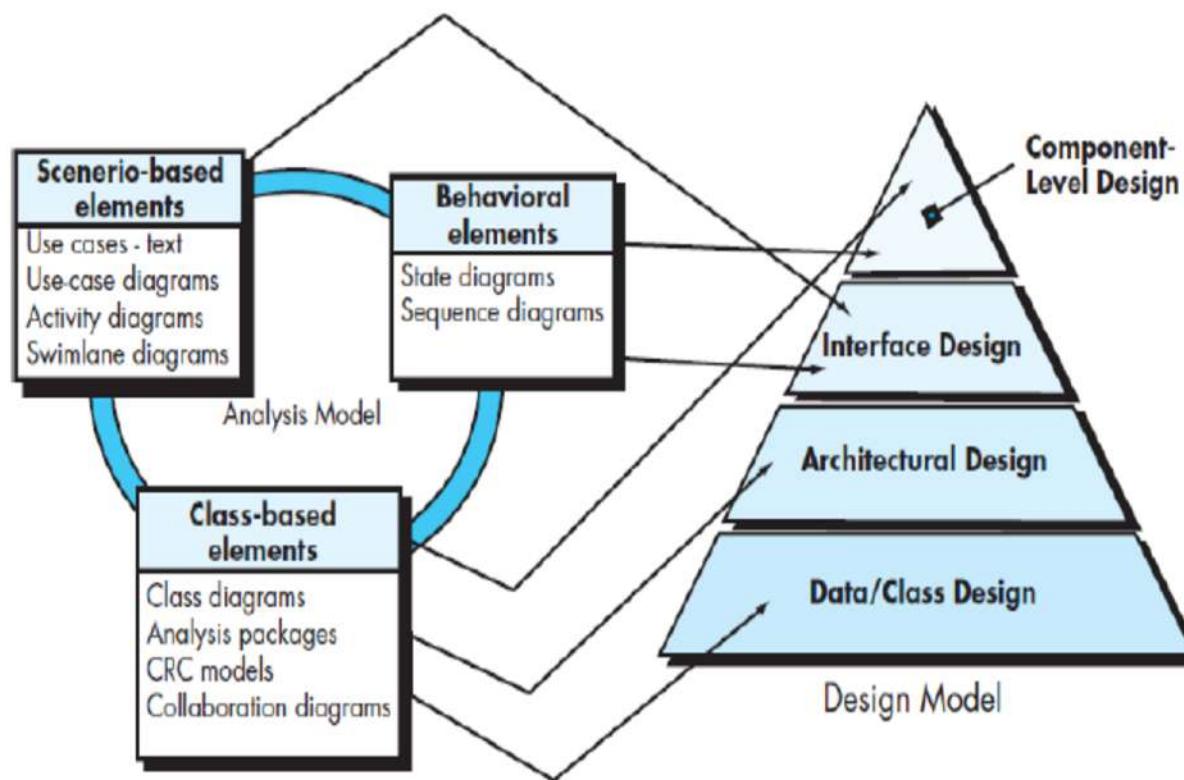
# DESIGN ENGINEERING

- Goal: to produce a model or representation that exhibits firmness, commodity and delight
- To accomplish this, designer must practice diversification and convergence
- Computer software design changes continually as new methods, better analysis, and broader understanding evolve
- Design within the context of software engineering
  - Last software engineering action within the modeling activity & sets the stage for construction
  - Flow of information during software design is shown in the fig –Translating the Requirements model into Design model
  - **Data / Class design** transforms analysis models into *design class realizations* & the *requisite data structures* required to implement the software
  - **Architectural design** defines the relationships between major structural elements of the software, the architectural styles & design patterns

- that can be used to achieve the requirements of the system
- The architectural design representation derived from the system specification, the analysis model, & the interactions of subsystems defined within the analysis model
- Interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it.
- Component-level design** transforms structural elements of the software architecture into a procedural description of software components

- Importance of software design: quality**

- Design Process and Design Quality
  - Software design is an iterative process
  - Represented at a higher level of abstraction
- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction



Throughout the design process, the quality of evolving design is assessed with a series of **formal technical reviews or design walkthroughs**

- McGlaughlin's 3 characteristics (goals) as a guide for the evaluation of a good design:
  - Design should implement all of the explicit requirements and it must accommodate all of the implicit requirements desired by stakeholders
  - Design should be a readable, understandable guide for those who generate code; who test and subsequently support the software.
  - Design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.
- Quality Guidelines – establish technical criteria for a good design
  - Design should exhibit an architecture that
    1. Has been created using recognizable architectural styles or patterns
    2. is composed of components that exhibit good design characteristics
    3. can be implemented in an evolutionary fashion

- Design should be modular
  - Design should contain **distinct representations** of data, architecture, interfaces, and components
  - Design should **lead to data structures** that are appropriate for the classes to be implemented
  - Design should **lead to components** that exhibit independent functional characteristics
  - Design should **lead to interfaces** that reduce the complexity of connections between components and with the external environment
  - Design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
  - Design should be represented using a notation that effectively communicates its meaning
- Quality Attributes** –HP's set of software quality attributes as acronym –**FURPS** – Functionality, Usability, Reliability, Performance and Supportability
  - Functionality:** assessed by evaluating the feature set, capabilities of program, generality of functions, & security of the system
  - Usability:** assessed by considering human factors, aesthetics, consistency and documentation

–**Reliability**: evaluated by measuring the frequency & severity of failure, accuracy of o/p results, MTTF, ability to recover from failure, & predictability

–**Performance**: measured using processing speed, response time, resource consumption, throughput, and efficiency

–**Supportability**: combines extensibility, adaptability, and serviceability, maintainability, testability, compatibility, configurability

### •**Design Concepts**

–M. A. Jackson [Jac75] once said: “**The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right.**”

–Fundamental software design concepts provide the necessary framework for “getting it right.”

### –**Abstraction:**

•At highest level of abstraction: solution provided using the language of problem environment

•At lower levels of abstraction: solution provided as detailed description

## Procedural and Data abstractions

- *Procedural abstraction* refers to a sequence of instructions that have a specific and limited function

- E.g. “open a door”

- Walk to the door

- Reach out and grasp knob

- Turn knob

- Pull door

- Step away from moving door

- *Data abstraction* a named collection of data that describes a data object

- E.g. “door”

- Door type

- Swing direction

- Opening mechanism

- Weight

- Dimensions

## **–Architecture**

- Alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”
- To derive an architectural rendering of a system which serves as a framework for conducting more detailed design activities
- Shaw and Garlan[Sha95a] describe a set of properties that should be specified as part of an architectural design
  - Structural properties:** define “the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another”.
  - Extra-functional properties:** address “ how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics”.
  - Families of related systems:** “draw upon repeatable patterns that are commonly encountered in the design of families of similar systems”

- Given the specification of these properties, the architectural design is represented using one or more number of different models:

- Structural models: organized collection of program components
  - Framework models: increase the level of design abstraction
  - Dynamic models: addresses the behavioral aspects
  - Process models: focus on the design of the business or technical process
  - Functional models: represents the functional hierarchy of a system

- Number of different Architectural Description Languages (ADL) developed to represent these models

- **Patterns**

- a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns
    - Intent of each design pattern is to provide a description that enables a designer to determine

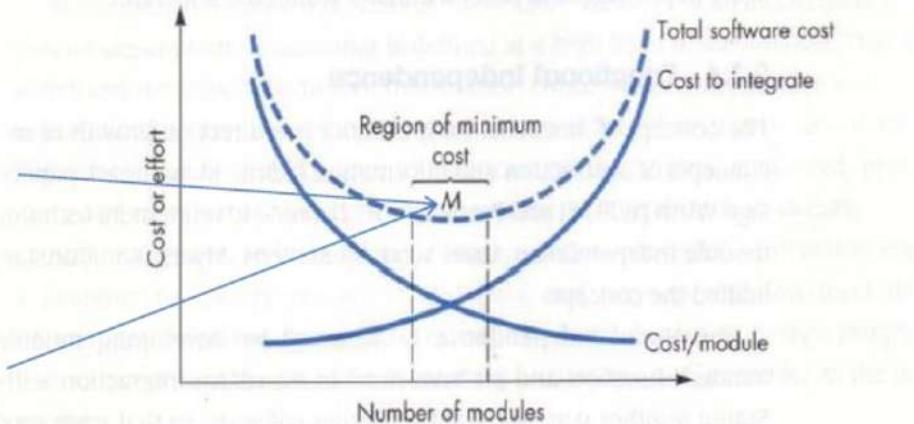
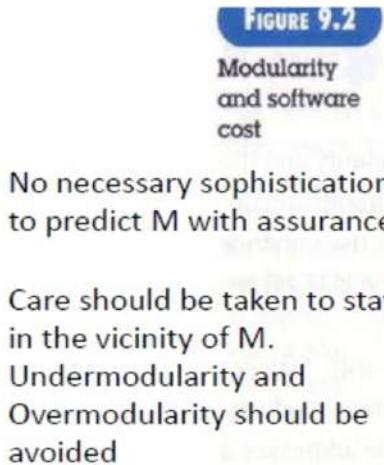
- whether the pattern is applicable to the current work
- whether the pattern can be reused and
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern

### –**Separation of Concerns**

- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- divide-and-conquer strategy

### –**Modularity**

- most common manifestation of separation of concerns
- Monolithic software cannot be easily grasped => no. of control paths, no. of variables, etc.
- The cost to develop an individual module decreases as the total number of modules increases
- However, as the no. of modules grows, the effort associated with integrating the modules also grows.



### – Information Hiding

- How do we decompose a software solution to obtain the best set of modules?
- Principle: modules be “characterized by design decisions that each hides from all others”
- Hiding defines & enforces access constraints to both procedural detail within a module & any local data structure used by the module handover

## **–Functional Independence**

- Direct outgrowth of modularity and the concepts of abstraction and information hiding
- Design software so that each module addresses a specific sub-function of requirements
- Has a simple interface
- Easier to maintain
- Error propagation is reduced
- Reusable modules are possible
- Independence assessed using 2 qualitative criteria:
  - Cohesion: indication of relative functional strength of a module
  - Coupling: indication of relative interdependence among modules

## Cohesion

–Natural extension of the information hiding concept

–Should ideally do just one thing

- **Coupling**

–Depends on the interface complexity between modules, the point at which entry or reference is made to a module & what data pass across the interface

–**Refinement**

- A process of elaboration

- Abstraction and Refinement are complementary concepts

- Refinement helps the designer to reveal low-level details as design progresses

–**Refactoring**

- Important design activity for agile methods

- “Is the process of changing a software system in such a way that it does not alter the external behavior of the design yet improves its internal structure”.

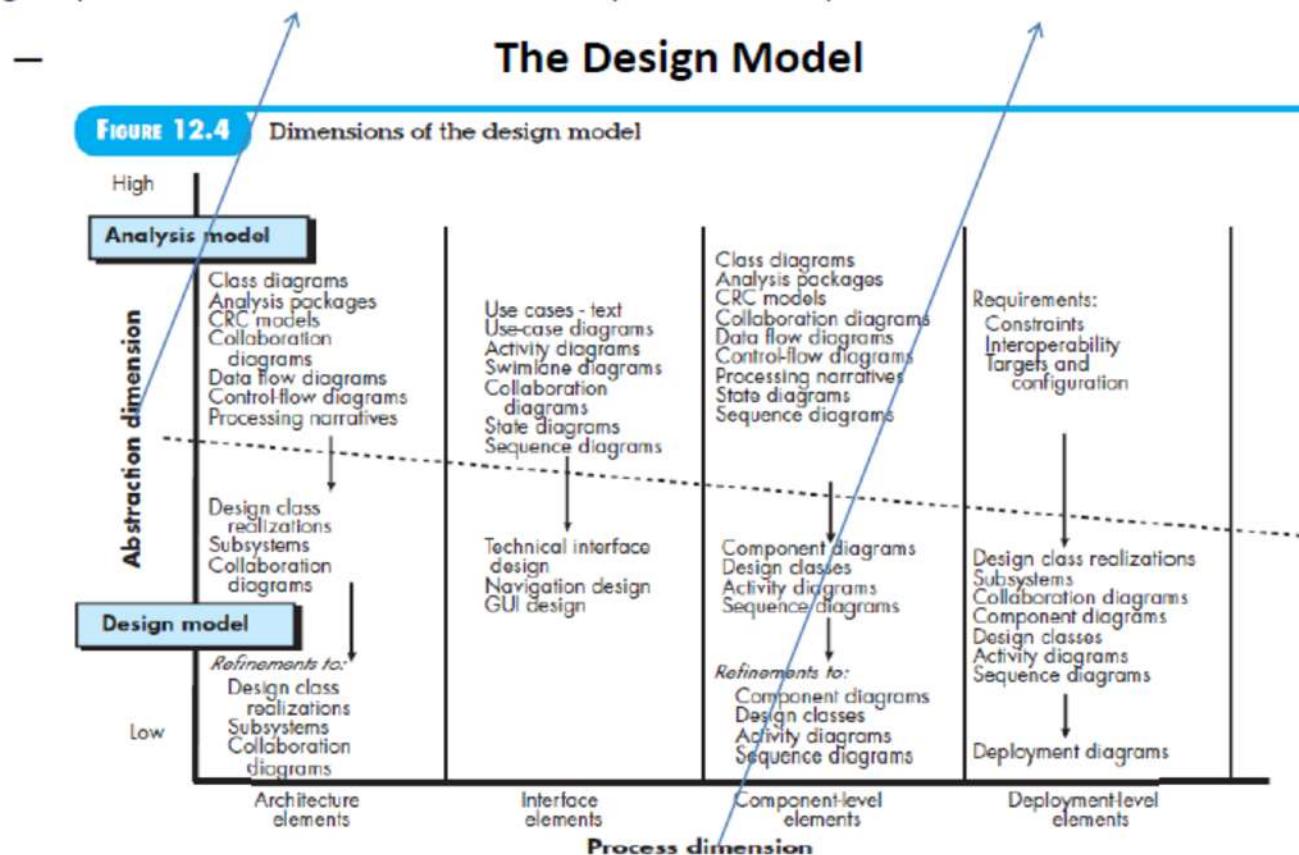
- Existing design is examined for
  - Redundancy
  - Unused design elements
  - Inefficiency or Unnecessary algorithms
  - Poorly constructed or inappropriate data structures
- **Design classes**
  - As the design model evolves, the software team must define a set of design classes that
    - (i) Refine the analysis classes by providing design detail that will enable the classes to be implemented &
    - (ii) Create a new set of design classes that implement a software infrastructure to support the business solution

- 5 different types each representing a different layer of design architecture are suggested:
- –**User Interface Classes**: define all abstractions that are necessary for human computer interactions
- –**Business domain Classes**: are often refinements of the analysis classes
- –**Process Classes**: implements lower-level business abstractions required to fully manage the business domain classes
- –**Persistent Classes**: represent data stores that will persist beyond the execution of the software
- –**System Classes**: implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world
- •Each design class is reviewed to ensure that it is “well-formed” which define 4 characteristics:

- Complete and sufficient
- –Primitiveness: methods associated with a design class should be focused on accomplishing one service for the class
- –High cohesion: small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities
- –Low coupling: design classes within a subsystem should have only limited knowledge of classes in other subsystems. This restriction is called “Law of Demeter” and suggest that a method should only send messages to methods in neighboring classes

represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively

indicates the evolution of the design model as design tasks are executed as part of the software process



- Elements of the design model use many of the same UML diagrams 7 that were used in the analysis model
- –Difference:
  - refined and elaborated as part of the design
  - More implementation specific detail is provided
  - Architectural structure and style, components that reside within the architecture and
  - Interfaces between the components and with the outside world
- Design model has 4 major elements
  - Data design elements
  - Architectural design elements
  - Interface design elements
  - Component-level design elements

- –Data Design Elements
- creates a model of data and/or information that is represented at a high level of abstraction.
- 

<b>Structure of the data</b>	<b>Always been an important part of software design</b>
At the program-component level	the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
At the application level	the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system
At the business level	the collection of information stored in distinct databases and reorganized into a “data warehouse” enables data mining or knowledge discovery that can have an impact on the success of the business itself.

- Architectural Design Elements
- •Equivalent to the floor plan of a house which gives us an overall view of the house
- •Give us an overall view of the software
- •the model derived from 3 sources:
  - Information about the application domain for the software to be built
  - specific requirements model elements such as use cases or analysis classes, their relationships & collaborations for the problem at hand &
  - the availability of architectural styles and patterns
- –Interface Design Elements
- •Equivalent to a set of detailed drawings for the doors, windows, & external utilities of a house
- •Detailed drawings tells us how things and information flow into and out of the house and within the rooms that are part of the floor plan

- Depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture
- •3 important elements of interface design:
  - 1.The user interface (UI)
  - 2.External interfaces to other systems, devices, networks, or other producers or consumers of information
  - 3.Internal interfaces between various design components
- •UI design incorporates
  - Aesthetic elements
  - Ergonomic elements (e.g., information layout and placement, metaphors, UI navigation)
  - Technical elements (e.g., UI patterns, reusable components)
- •External Interface design incorporates

- Error checking and appropriate security features
- Design of internal interface closely aligned with component-level design
- To enable communication and collaboration between operations in classes => messaging
- Messaging: designed to accommodate the requisite information transfer and the specific functional requirements of the operation that has been requested
- In UML, an interface is defined as:
  - “An interface is a specifier for the externally-visible operations of a class, component, or other classifier without specification of internal structure”
  - More simply stated “an interface is a set of operations that describes some part of the behavior of a class and provides access to these operations”

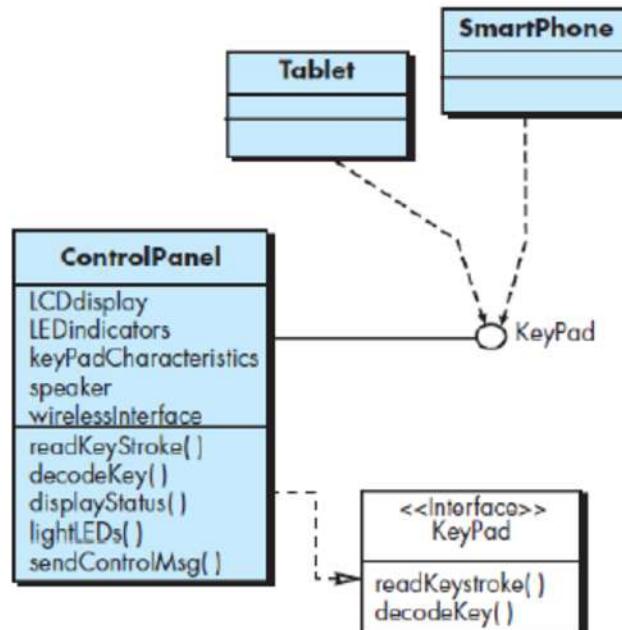
- E.g. SafeHome Security Function
- Interface: Control Panel class

**FIGURE 12.5**

Interface representation for ControlPanel

**ControlPanel** class provides KeyPad operations as part of its behavior.

In UML, this is characterized as a **realization**. That is, part of the behavior of **ControlPanel** will be implemented by realizing **KeyPad** operations. These operations will be provided to other classes that access the interface



## – Component-level Design Elements

- Equivalent to a set of detailed drawings for each room in a house
- Component-level design for software fully describes the internal design of each software component
- To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations

**FIGURE 12.6**

A UML compo-  
nent diagram



## Deployment-level Design Elements

- Indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
- E.g. Safe Home Product
  - Configured to operate within 3 primary computing environments
    - » A home-based PC,
    - » The Safe-Home control panel, and
    - » A server (providing Internet services)
  - Descriptor form
  - Deployment diagram shows the computing environment but does not explicitly indicate configuration details

# Creating an Architectural Design

- Objective: To provide a systematic approach for the derivation of the architectural design –**preliminary blueprint** from which the software is constructed

- Architecture is not the operational software
  - It is a representation that enables a software engineer to
- 1.Analyze the effectiveness of the design in, meeting its stated requirements
- 2.Consider architectural alternatives at a stage when making design changes is still relatively easy, and
- 3.Reduce the risks associated with the construction of the software
  - In the context of architectural design, a software component can be => a simple program module, or an object-oriented class, or it can be extended to include databases and middleware
  - The design of software architecture considers 2 levels of the design pyramid –***data design and architectural design***

# **Data Design**

- Data Design

- To represent the data component of the architecture in conventional systems and class definitions in object-oriented systems

- Architectural Design

- Focuses on the representation of the structure of software components, their properties and interactions

- Data design action translates data objects defined as part of the analysis model into data structures at the software component level and a database architecture at the application level

- **Data Design at the Architectural level**

- Big Data / Databases with hundreds of TeraBytes

- How to extract useful information from this data environment, particularly when the information desired is cross-functional?

- Solution:
- **Data mining techniques**, also called ***Knowledge Discovery in Databases (KDD)*** that navigate through existing databases in an attempt to extract appropriate business-level info
- Also have **Data Warehousing** which adds an additional layer to the data architecture
  - A ***data warehouse*** is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business
    - A large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business
  - Data Design at the Component level
    - A set of principles used to specify and design data structures
    - The systematic analysis principles applied to function and behavior should also be applied to data

# Architectural Styles and Patterns

- Architectural style –a template for construction; is a transformation that is imposed on the design of an entire system
- Software built exhibits one of many architectural styles; each style describes a system category that encompasses
  - 1. A set of components that perform a function required by a system
  - 2. A set of connectors that enable “communication, coordination, and cooperation” among components
  - 3. Constraints that define how components can be integrated to form the system and
  - 4. Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts
- An Architectural pattern –imposes a transformation on the design of an architecture
- But a pattern differs from a style in a number of fundamental ways:

The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety

2.A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)

3.Architectural patterns tend to address specific behavioral issues within the context of the architecture

- A Brief Taxonomy of Architectural Styles**

- Data-Centered Architectures**

- Data-Flow Architectures**

- Call and Return Architectures:**

- Main program / subprogram Architectures**

- Remote Procedure Call Architectures**

- Object-oriented Architectures**

- Layered Architectures**

To achieve a program structure that is relatively easy to modify and scale.

Client software accesses the data independent of any changes to the data or the actions of other client software.

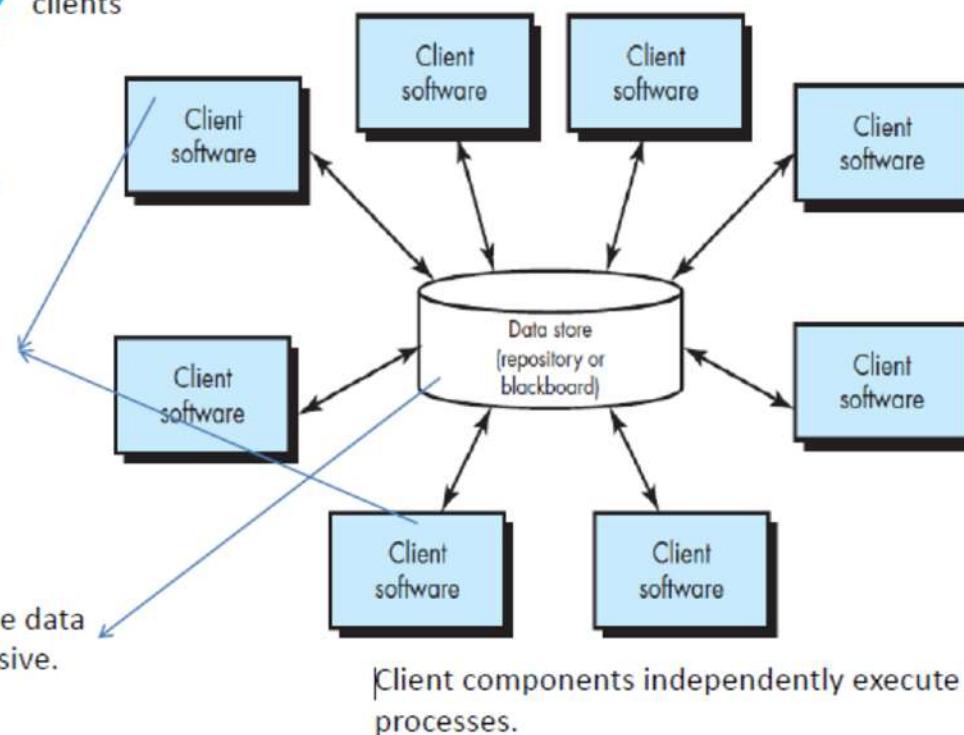
promote *integrability*, i.e., existing components can be changed and new client components added to the architecture without concern about other clients

FIGURE 13.1

Data-centered architecture

A data store resides centrally & is accessed frequently by other components that update, add, delete, or otherwise modify data within the store

In some cases the data repository is passive.

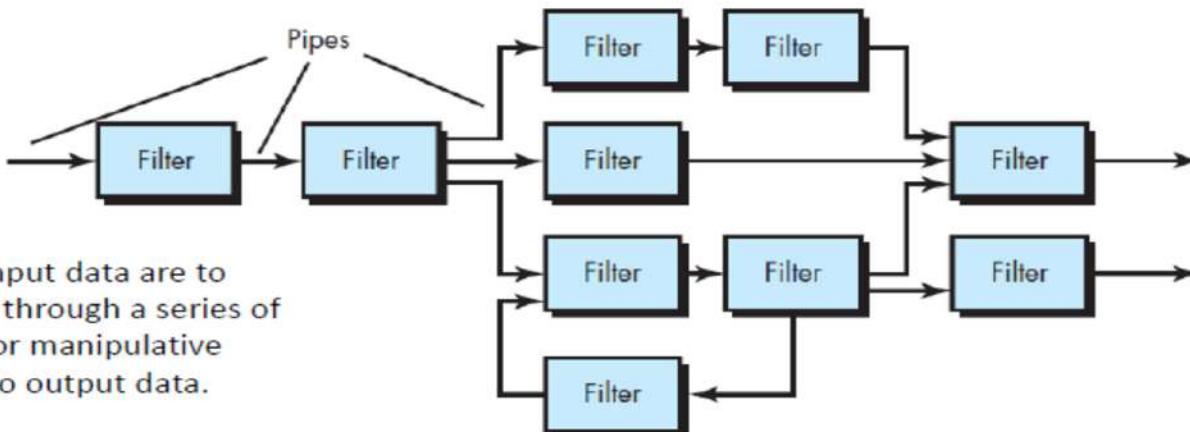


Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form

FIGURE 13.2

Data-flow architecture

Applied when input data are to be transformed through a series of computational or manipulative components into output data.



Pipes and filters

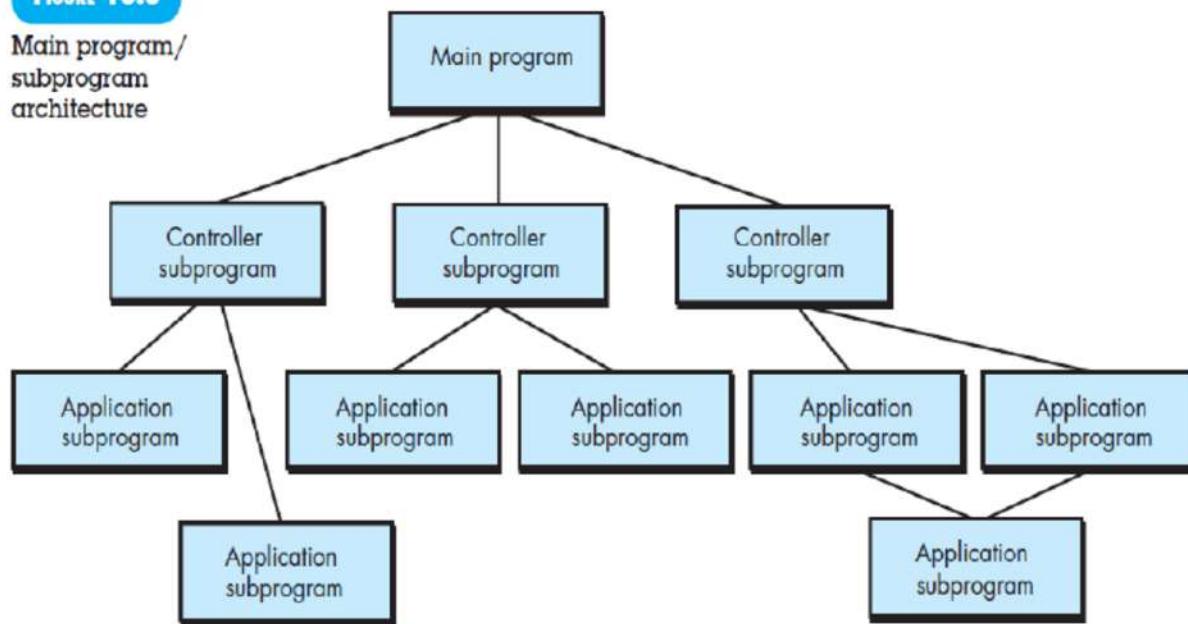
A pipe-and-filter pattern (Figure 13.2) has a set of components, called *filters*, connected by pipes that transmit data from one component to the next.

If the data flow degenerates into a single line of transforms, it is termed **batch sequential**.

This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it

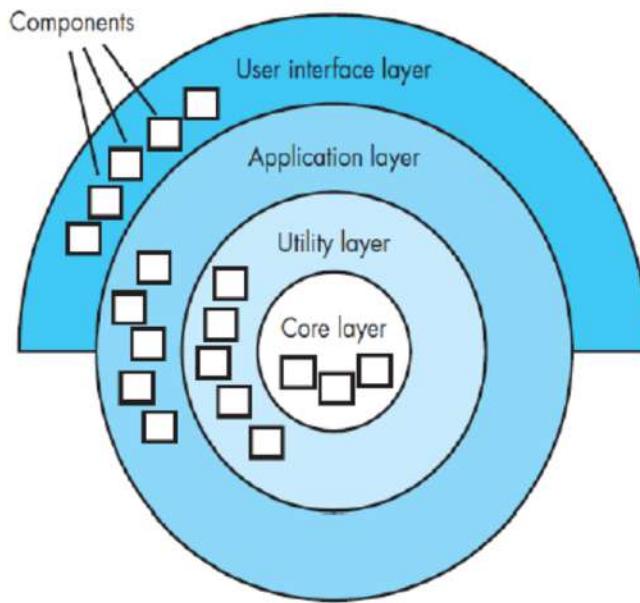
**FIGURE 13.3**

Main program/  
subprogram  
architecture



**FIGURE 13.4**

Layered architecture



## Architectural Patterns

- Similar to the kitchen pattern needed for a house
- Consider the rules for placements of the things relative to workflow in the room (sink, tap, cabinets, wall switches, etc...)
- For software define a specific approach for handling some behavioral characteristics of the system
- Addresses issues such as
  - Concurrency: to handle multiple tasks through “parallelism”
    - One approach: using an OS process management pattern that allows components to execute concurrently
    - Another approach: to define a task scheduler pattern
  - Persistence: data persists if it survives past the execution of the process that created it

–2 architectural patterns used to achieve persistence:

- »A DBMS pattern
- »An application level persistence pattern (e.g. word processing software that manages its own document structure)
- Distribution: addresses the manner in which systems communicate with one another in a distributed environment
- 2 elements to this problem
  - »The way in which entities connect to one another, &
  - »The nature of communication that occurs
- Most common architectural pattern –*broker*
- A broker acts as the middle man between the client component and the server component. E.g. CORBA architecture

## Organization and Refinement

–Important to establish a set of design criteria that can be used to assess an architectural design

–Control:

- How is control managed within the architecture?
- What is the role of components within this control hierarchy?
- How do components transfer control within the system?
- Is control synchronized or do components operate asynchronously?

–Data:

- How are data communicated between components?
- What is the mode of data transfer?
- How do functional components interact with data components?
- How do data and control interact within the system?

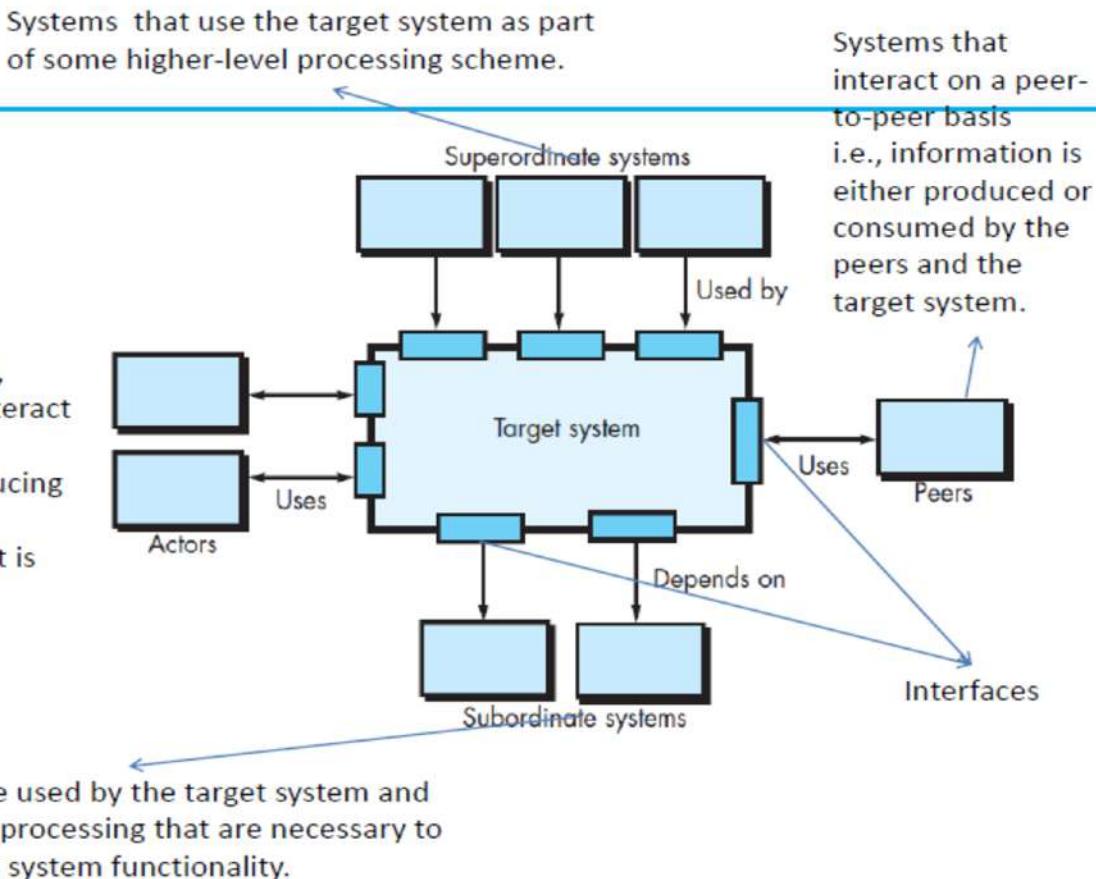
## Architectural Design

- The context is that the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction
- Once the context is modeled and all external software interfaces are described,
  - The designer specifies the structure of the system by defining and refining software components that implement the architecture
  - Continues iteratively until a complete architectural structure has been derived
- Representing the System in Context**
  - Architectural context represents how the software interacts with entities external to its boundaries

**FIGURE 13.5**

Architectural context diagram  
Source: Adapted from [Bos00].

Entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.



## Architectural Context Diagram mapped with Home Security function of Safe Homeproduct

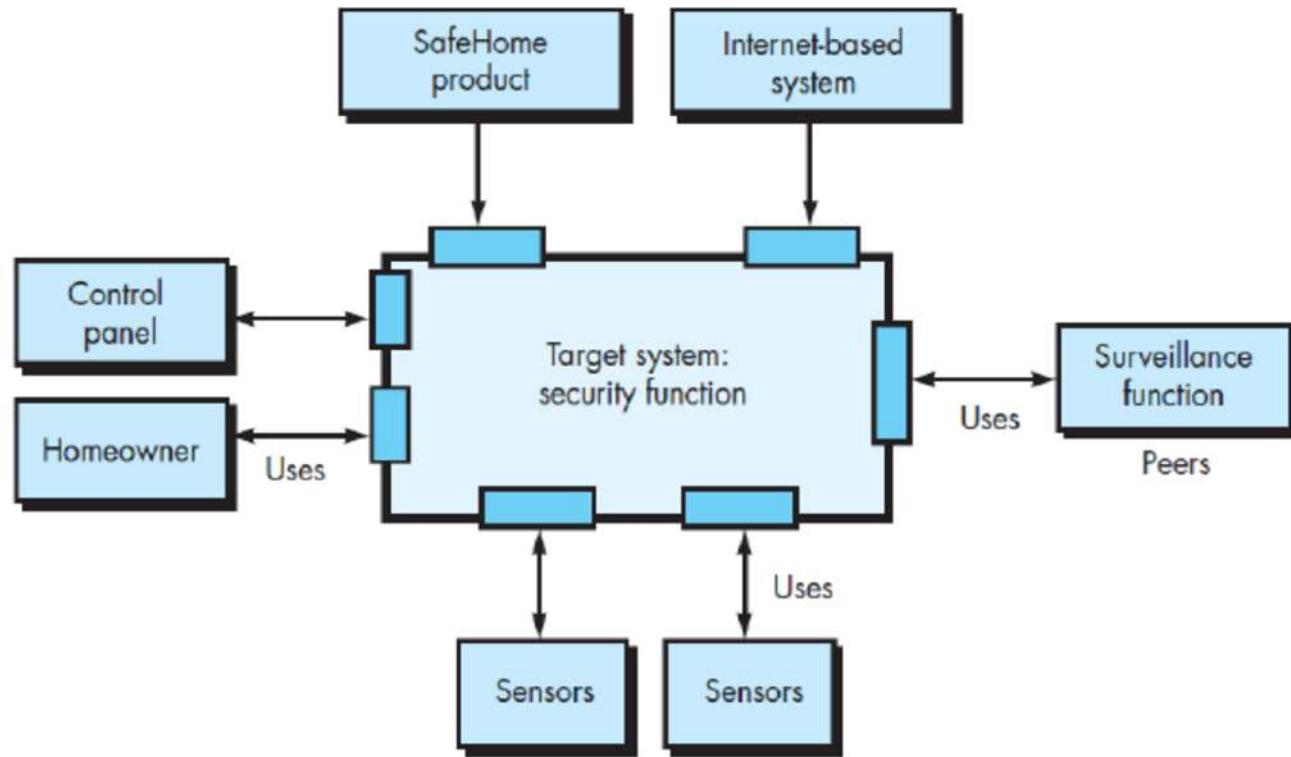
- Overall Safe Homeproduct Controller & the Internet based system => Superordinate to the Security function
- Surveillance function => peer system
- Homeowner and control panels => actors that are both producers and consumers of information used-produced by security software
- Sensors => subordinate

### –Defining Archetypes

- An Archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- Target system architecture is composed of archetypes which represent stable elements of the architecture
- The following archetypes are defined:
  - Node: represents a cohesive collection of i/o elements of the home security function. E.g. a node may be comprised of various sensors and a variety of alarm indicators

**FIGURE 13.6**

Architectural context diagram for the *SafeHome* security function



## Refining the Architecture into Components

- How are components chosen?
- Components of the software architecture are derived from 3 sources:
  - **The application domain:** Analysis classes represent entities within the business domain
  - **The infrastructure domain:** memory management components, communication components, database components, and task management components integrated into software architecture
  - **The interface domain:** imply one or more specialized components that process the data that flow across the interface
- Continuing the Safe Home security function example, the components that address the following functionality are:
  - External Communication Management: e.g. Internet based systems, external alarm notification
  - Control Panel Processing

## Mapping Data Flow into a Software Architecture

- Comprehensive mapping from the analysis model to architectural styles doesn't exist
- No practical mapping available
- Designer must approach the translation of requirements to design for these styles by using the techniques discussed previously
- One approach to architectural mapping:
  - Enabling a designer to derive complex call and return architecture from DFDs within the analysis model
  - Called as Structured Design: provide convenient transition from a DFD to software architecture
  - **Transform Mapping / Transform Flow**
    - Information enters the system along paths that transforms external data into an internal form

Identified as Incoming Flow

At the Kernel of the software, a transition occurs;

- Incoming data are passed through a transform center and begin to move along paths that now lead “out” of the software => data moving along these paths are called **outgoing flow**

- **Transform Mapping:** set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style

- **Design steps initiated to map DFDs into an architecture that describe information flow within the SafeHomeSecurity function:**

1. Review the fundamental system model (Context Data Flow Diagram)

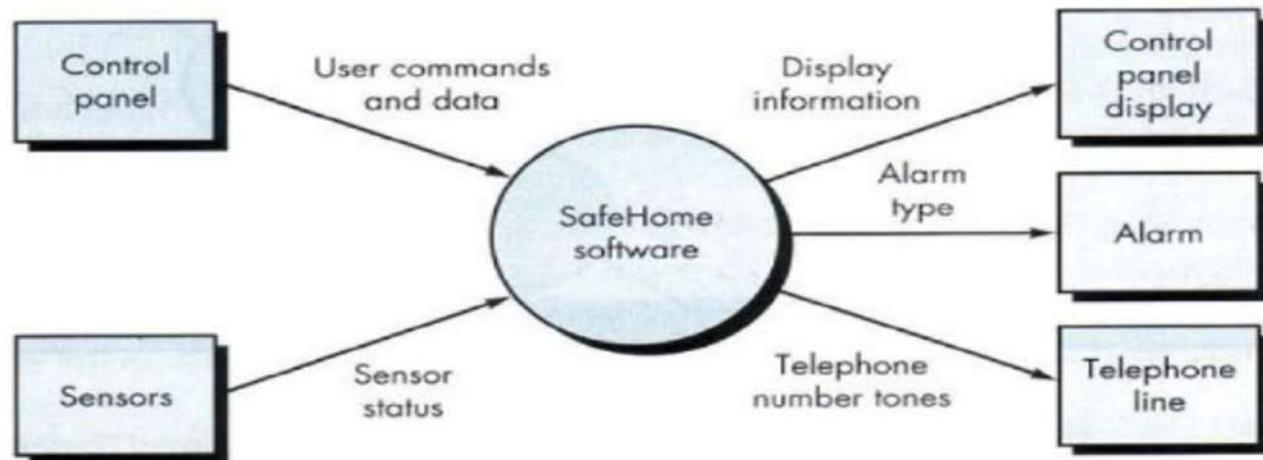
2. Review and Refine DFDs for the software

- » The level 2 DFD for monitor sensors is examined and a level 3 DFD is derived

**FIGURE 10.11**

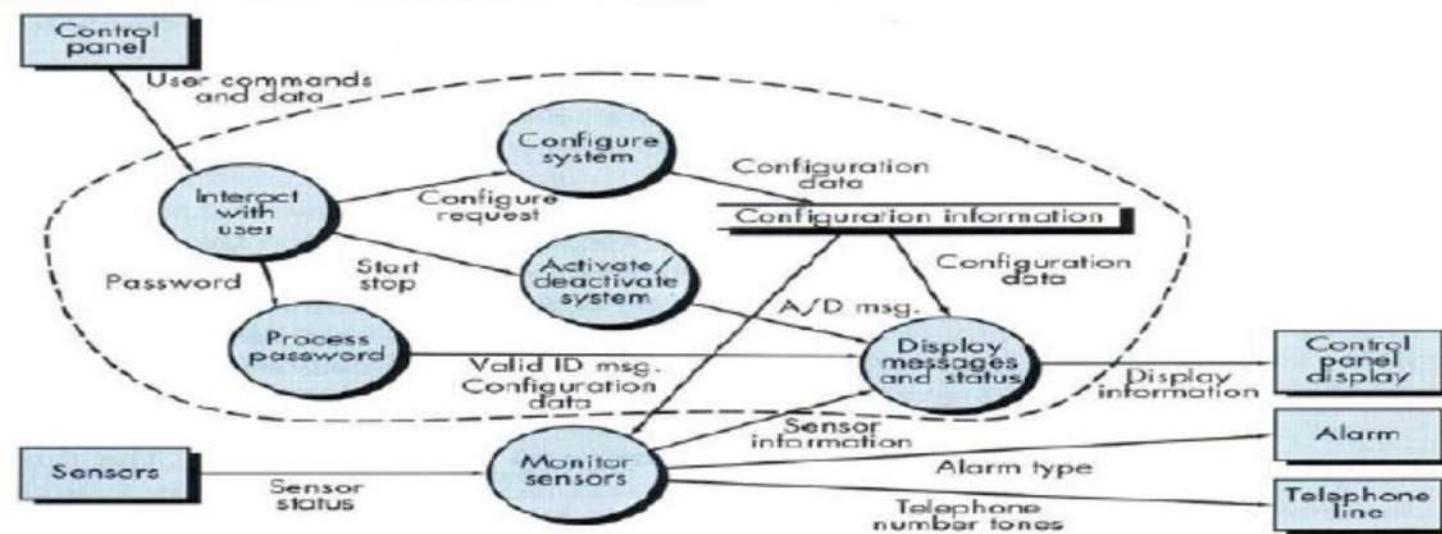
Context level DFD for the *SafeHome* security function

Represents the external producers and consumers of data flow into and out of the function



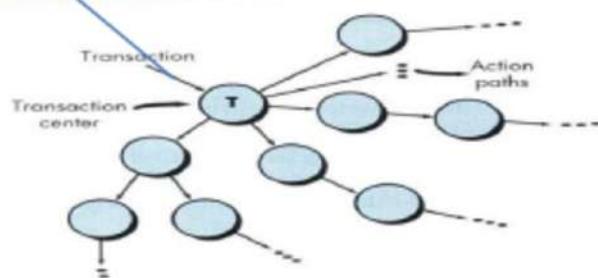
**FIGURE 10.12**

Level 1 DFD for the *SafeHome* security function



The hub of information flow from which many paths originate

FIGURE 10.10  
Transaction flow



3. Determine whether the DFD has transform or transaction flow characteristics

- » **Transaction Flow**
- » Information flow is often characterized by a single data item, called a **transaction**, that triggers other data flow along one of many **action paths**

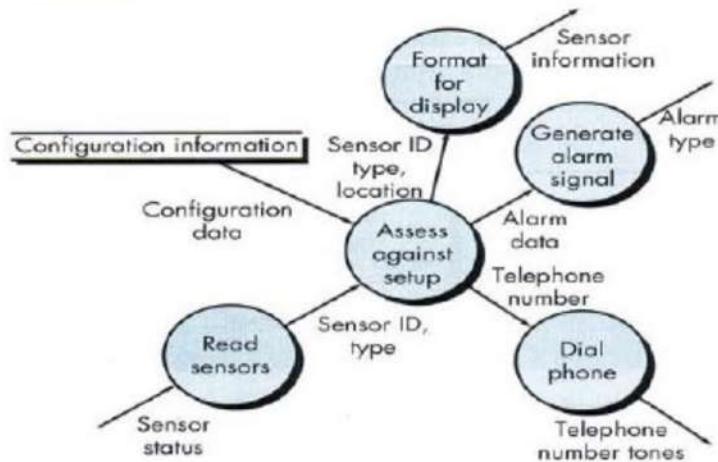
Converts external world information into a transaction

- » At level 3, each transform in the DFD exhibits relatively high cohesion
- » The process implied by a transform performs a single, distinct function that can be implemented as a component in the SafeHome software

**FIGURE 10.13**

Level 2 DFD  
that refines  
the monitor  
sensors  
transform

Transform center lie within  
the 2 shaded boundaries  
that run from top to bottom

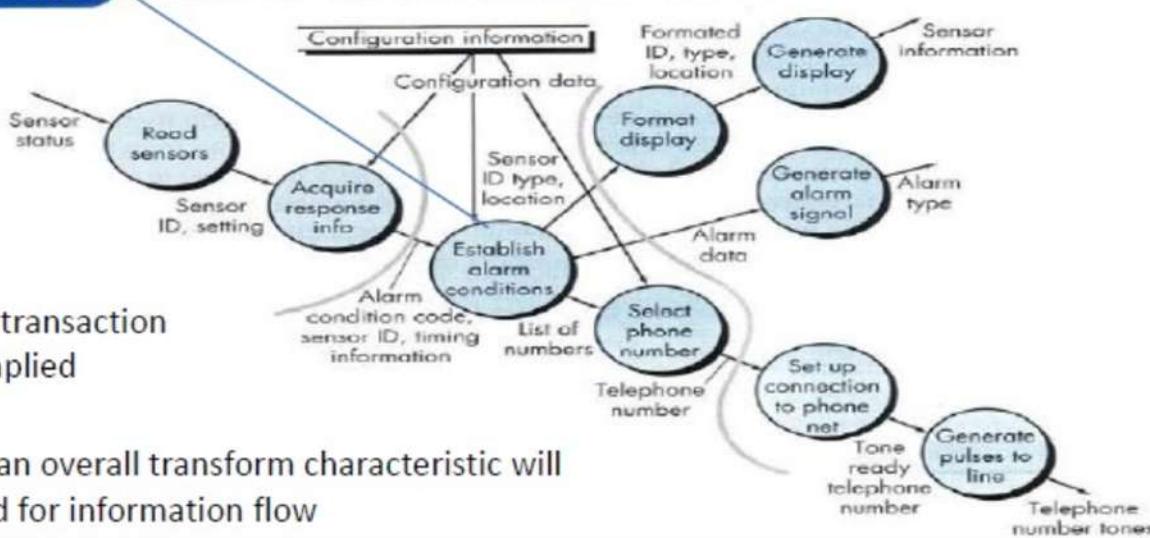


**FIGURE 10.14**

Level 3 DFD for monitor sensors with flow boundaries

No distinct transaction  
centre is implied

Therefore, an overall transform characteristic will  
be assumed for information flow



- » Here, the designer selects global flow characteristics based on the prevailing nature of DFD
  - » In addition, local regions of transform or transaction flow are isolated
  - » These sub-flows used to refine program architecture
4. Isolate the transform centre by specifying incoming and outgoing flow boundaries
    - » Incoming and Outgoing flow boundaries are open to interpretation as the designers may select different points in the flow as boundary locations
    - » Alternative design solutions derived from placing different flow boundaries
  5. Perform “First-level Factoring”
    - » Factoring results in a program structure in which top-level components perform decision-making
    - » Low-level components perform most input, computation and output work

- » Middle-level components perform some control and do moderate amounts of work

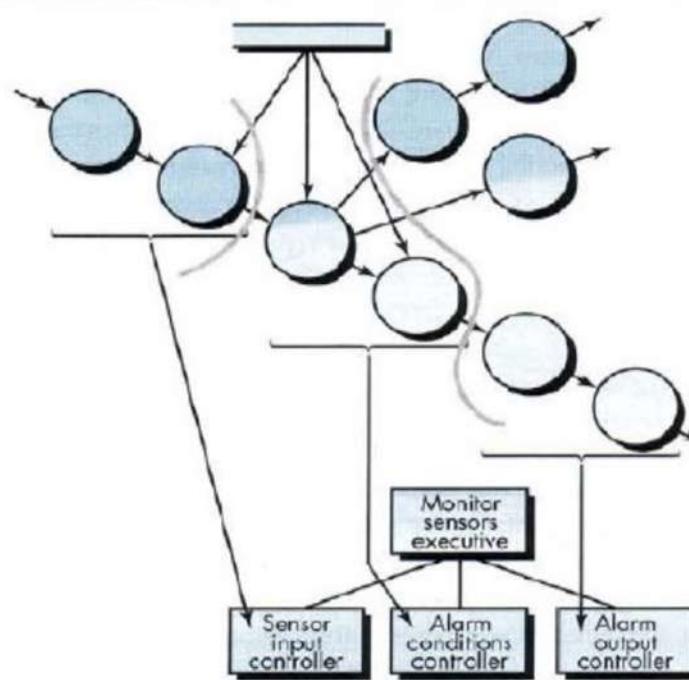
FIGURE 10.15

First-level  
factoring for  
*monitor  
sensors*

Incoming Information Processing  
Controller coordinates receipt of all  
incoming data

Transform Flow Controller  
supervises all operations on data

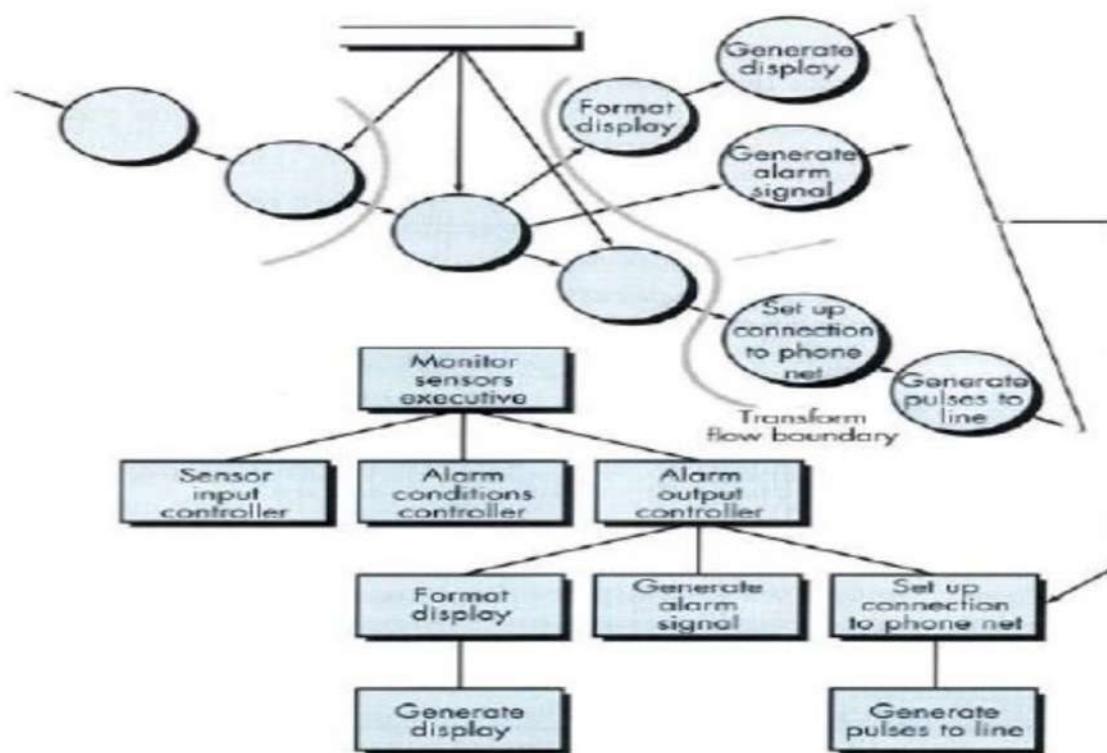
Outgoing Information Processing  
Controller coordinates production  
of output information



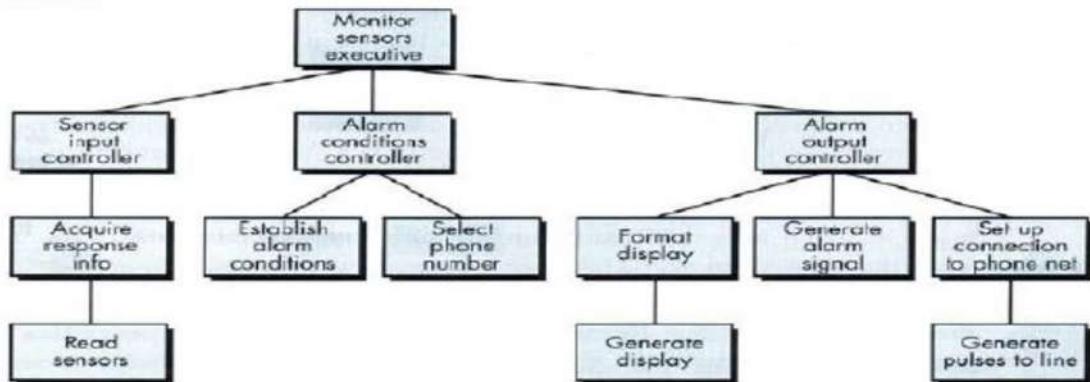
## 6. Perform “Second-level Factoring”

FIGURE 10.16

Second-level factoring for monitor sensors



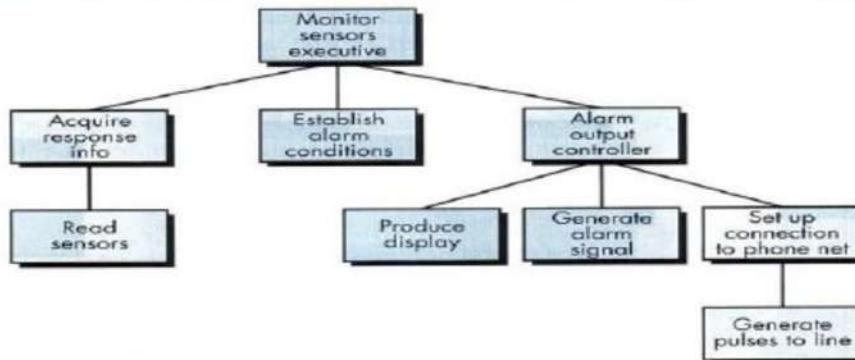
**FIGURE 10.17** First iteration structure for monitor sensors



7. Refine the First-Iteration architecture using design heuristics for improved software quality

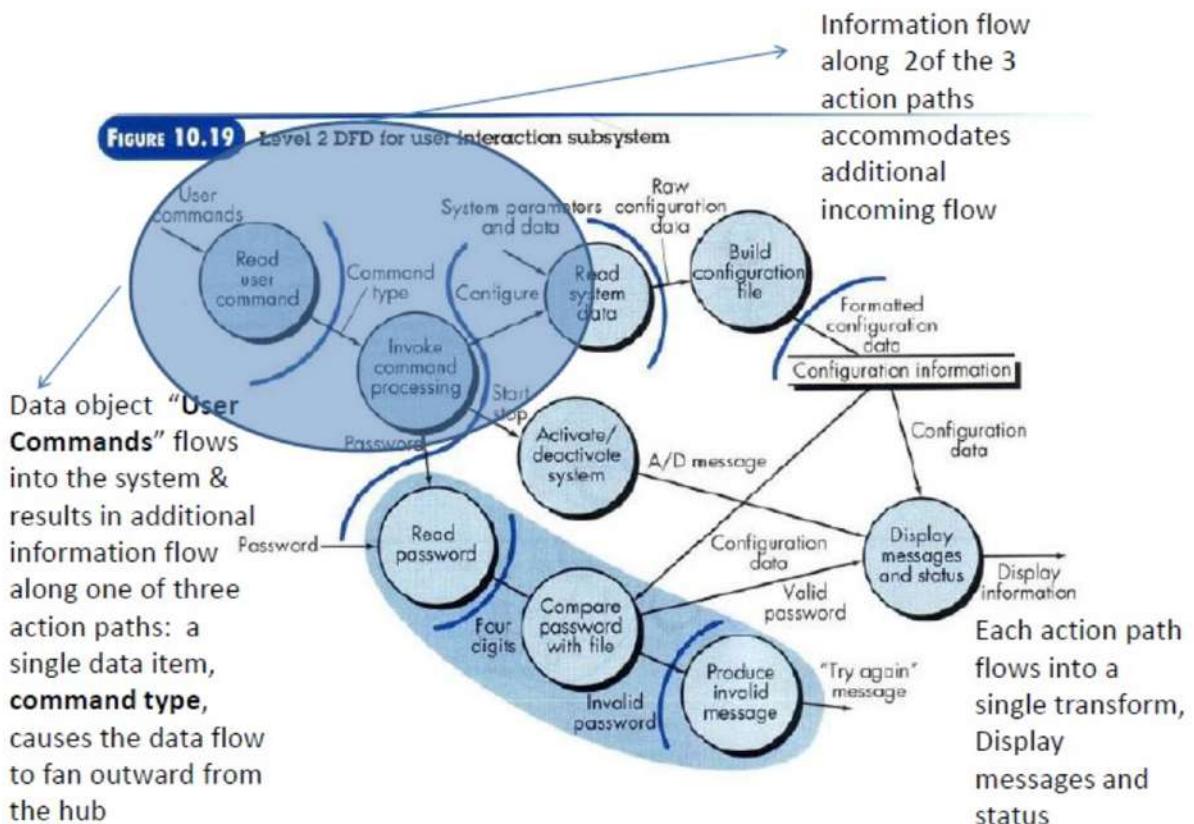
**FIGURE 10.18**

Refined program structure for monitor sensors



## –Transaction Mapping

- A single data item triggers one or more information flows that effect a function implied by the triggering data item
- This data item is called a “transaction”
- Illustrated by considering the user interaction subsystem of the *Safe Homesecurity* function
- Design steps used to map transaction flow into a software architecture –identical to transform mapping
  - Major difference lies in the mapping of DFD to software structure
- From the Level 1 DFD (fig.10.12), the data flow is refined as a Level 2 DFD as shown in fig. 10.19

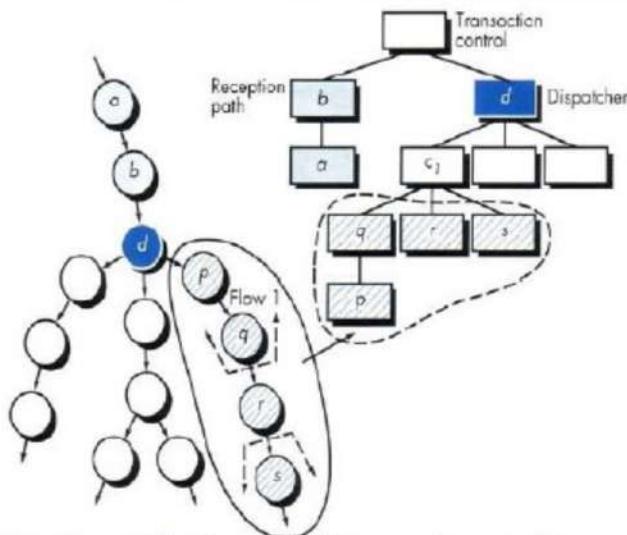


**Step 1: Review the fundamental system model**

- Step 2: Review and Refine DFDs for the software**
- Step 3: Determine whether the DFD has transform or transaction flow characteristics**
  - Flow along 2 of the action paths starting from the “Invoke command processing” bubble appears to have transform flow characteristics
  - Flow boundaries must be established for both flow types
- Step 4: identify the transaction center and the flow characteristics along each of the action paths –“Invoke Command Processing” which lies at the origin of a no. of action paths**
- Step 5: Map the DFD in a program structure amenable to transaction processing**
- Step 6: Factor and refine the transaction structure and the structure of each action path**
- Step 7: Refine the first-iteration architecture using design heuristics for improved software quality**

FIGURE 10.20

Transaction mapping

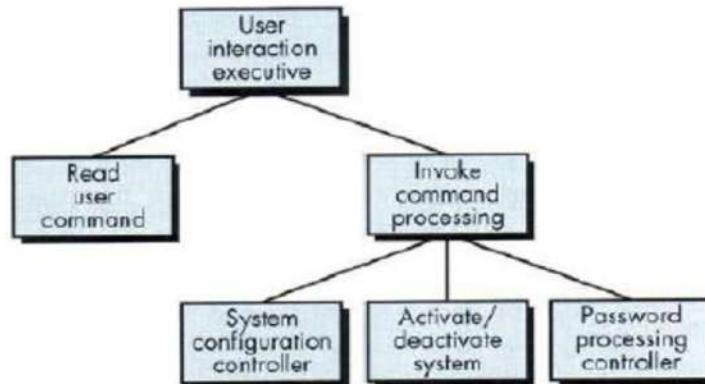


- Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch
- Structure of incoming branch developed in the same way as transform mapping
- Structure of dispatch branch contains a dispatcher module that controls all subordinate action modules
  - Read user command and Activate/Deactivate system map directly into the architecture without intermediate control modules
  - Transaction center “Invoke command processing” maps directly into a dispatcher module of the same name

**FIGURE 10.21**

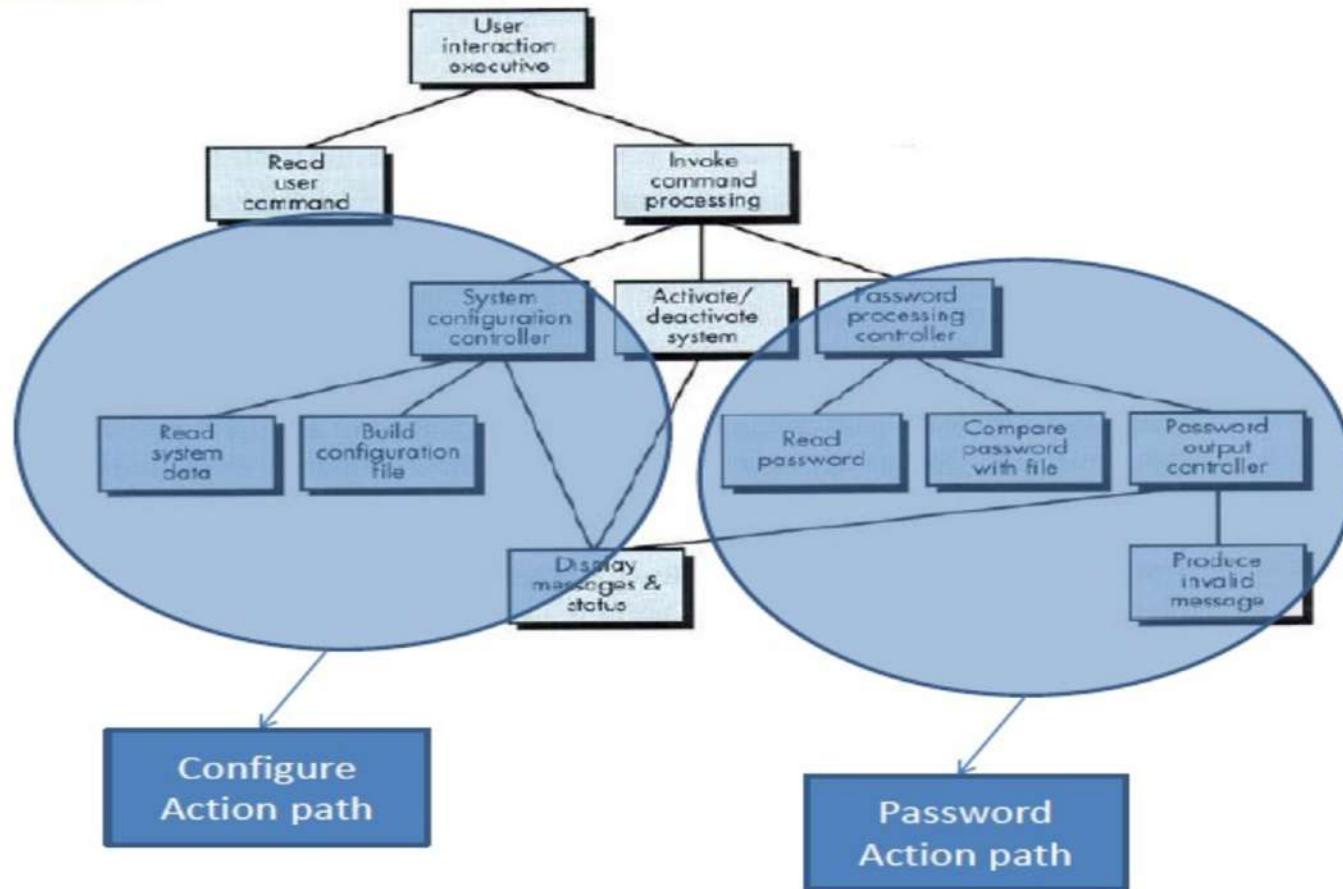
First-level  
factoring for  
user interac-  
tion subsystem

Related to Step 5



- Each action path of the DFD has its own information flow characteristics => transform or transaction flow may be encountered
- Refining the Architectural Design
  - Software designer should be concerned with developing a representation of software that will meet all the functional and performance requirements and merit acceptance based on design measures and heuristics
  - Design refinement strive for the smallest no. of components that is consistent with effective modularity &
  - The least complex data structure that adequately serves information requirements

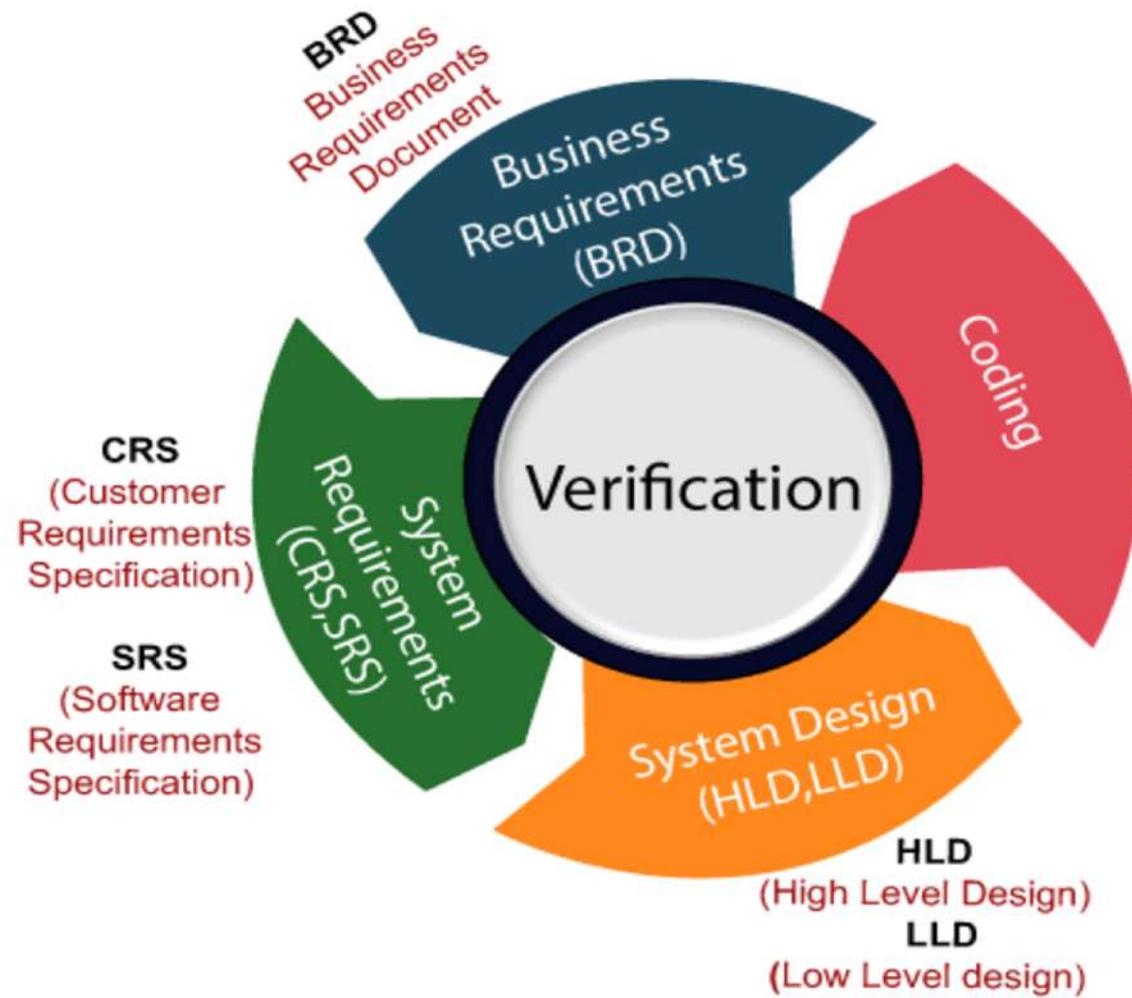
**FIGURE 10.22** First iteration architecture for user interaction subsystem



# TESTING STRATEGIES

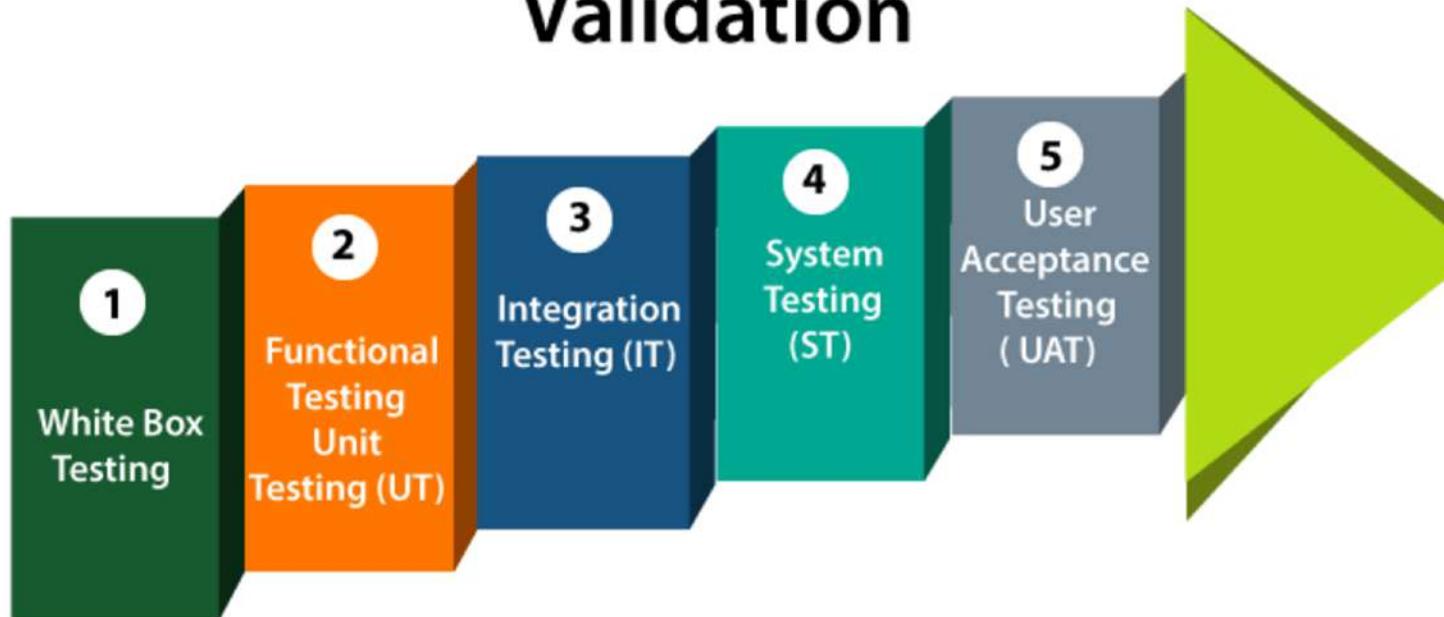
- **General characteristics of Software Testing**
  - –Formal Technical Reviews
  - –Testing begins at the component level and works “outward” towards the integration of the computer system
  - –Diff. testing techniques appropriate at diff. points in time
  - **Verification and Validation**
    - Verification refers to the set of activities that ensure that software correctly implements a specific function
    - Validation refers to the set of activities that ensure that the software built is traceable to customer requirements
    - Boehm states this in another way:
    - **Verification: are we building the right product?**
    - **Validation : are we building the product right?**

- Verification



- Validation

# Validation



- A Software Testing strategy for Conventional Software Architectures

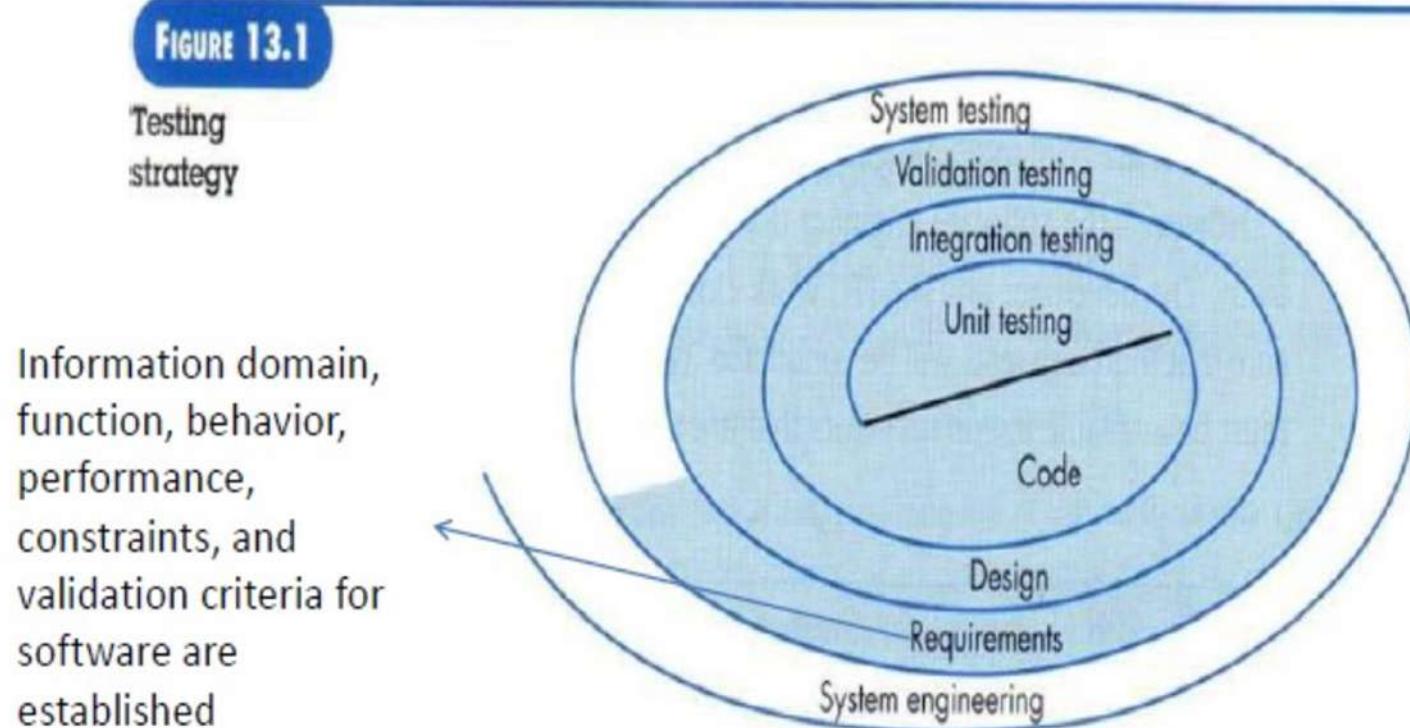
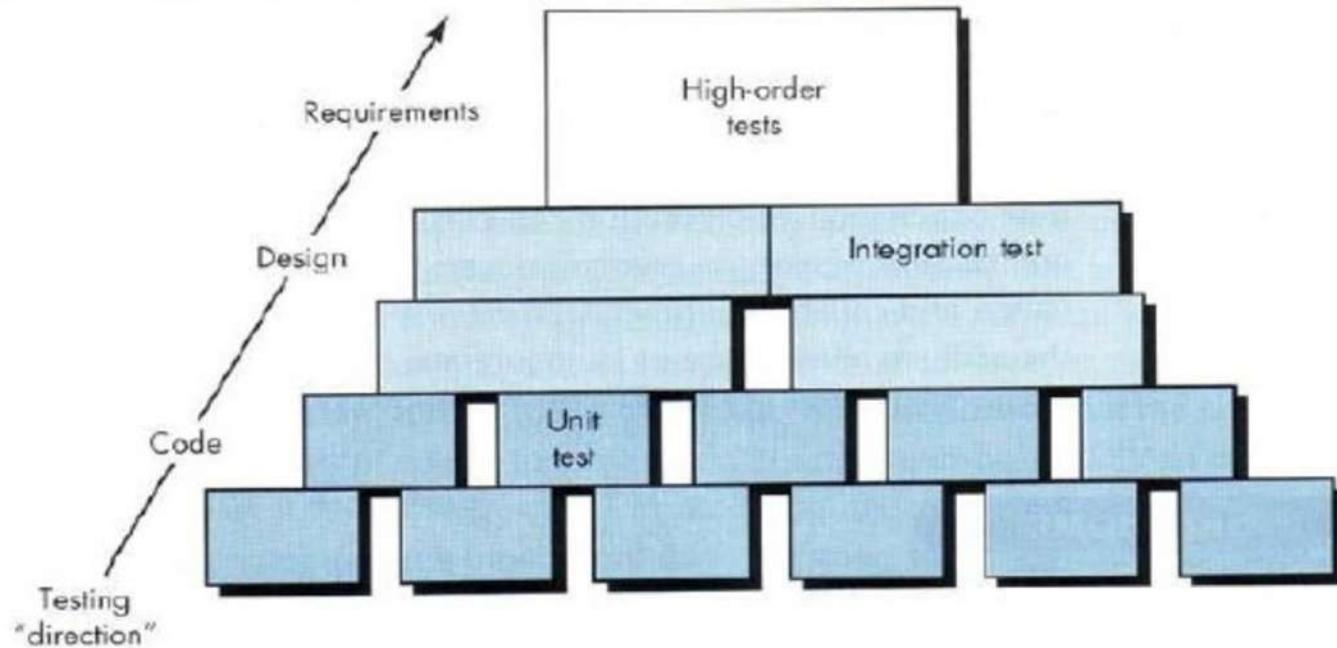


FIGURE 13.2

Software testing steps



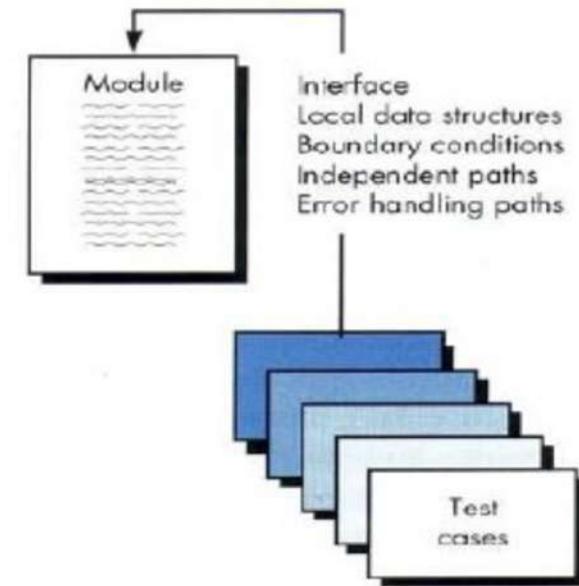
- A Software Testing strategy for Object-oriented Software Architectures
- Criteria for Completion of Testing
  - When are we done testing? No definitive answer

- Test strategies for Conventional Software
- –It takes an incremental view of testing, beginning with
  - The testing of individual program units
  - Moving to tests designed to facilitate the integration of the units &
  - Culminated with tests that exercise the constructed system
- –Unit Testing
  - Important control paths are tested to uncover errors within the boundary of the module
  - Focuses on the internal processing logic and data structures within the boundaries of a component
  - Selective Testing of execution paths is an essential task during the unit test

FIGURE 13.3

Unit test

### Unit Test Considerations



- Test cases designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow
- More common errors are
  - Incorrect arithmetic precedence
  - Mixed mode operations
  - Incorrect initialization
  - Precision inaccuracy
  - Incorrect symbolic representation of an expression

- Test cases
- Comparison of different data types
- –Incorrect logical operators or precedence
- –Expectation of equality when precision error makes equality unlikely
- –Incorrect comparison of variables
- –Improper or nonexistent loop termination
- –Failure to exit when divergent iteration is encountered
- –Improperly modified loop variables

- Unit test procedures

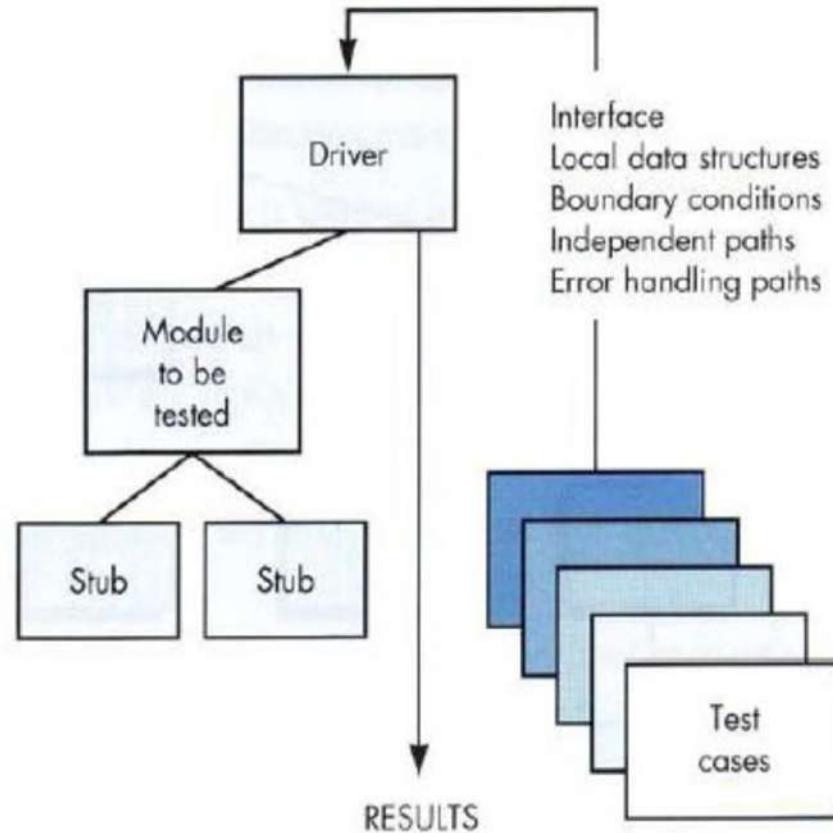
- Drivers and Stubs must be developed for each unit test
- **Driver**: a “main program” that accepts test case data, passes such data to the component, & prints relevant results
- **Stubs**: a “dummy subprogram” serve to replace modules that are subordinate to the component to be tested

**FIGURE 13.4**

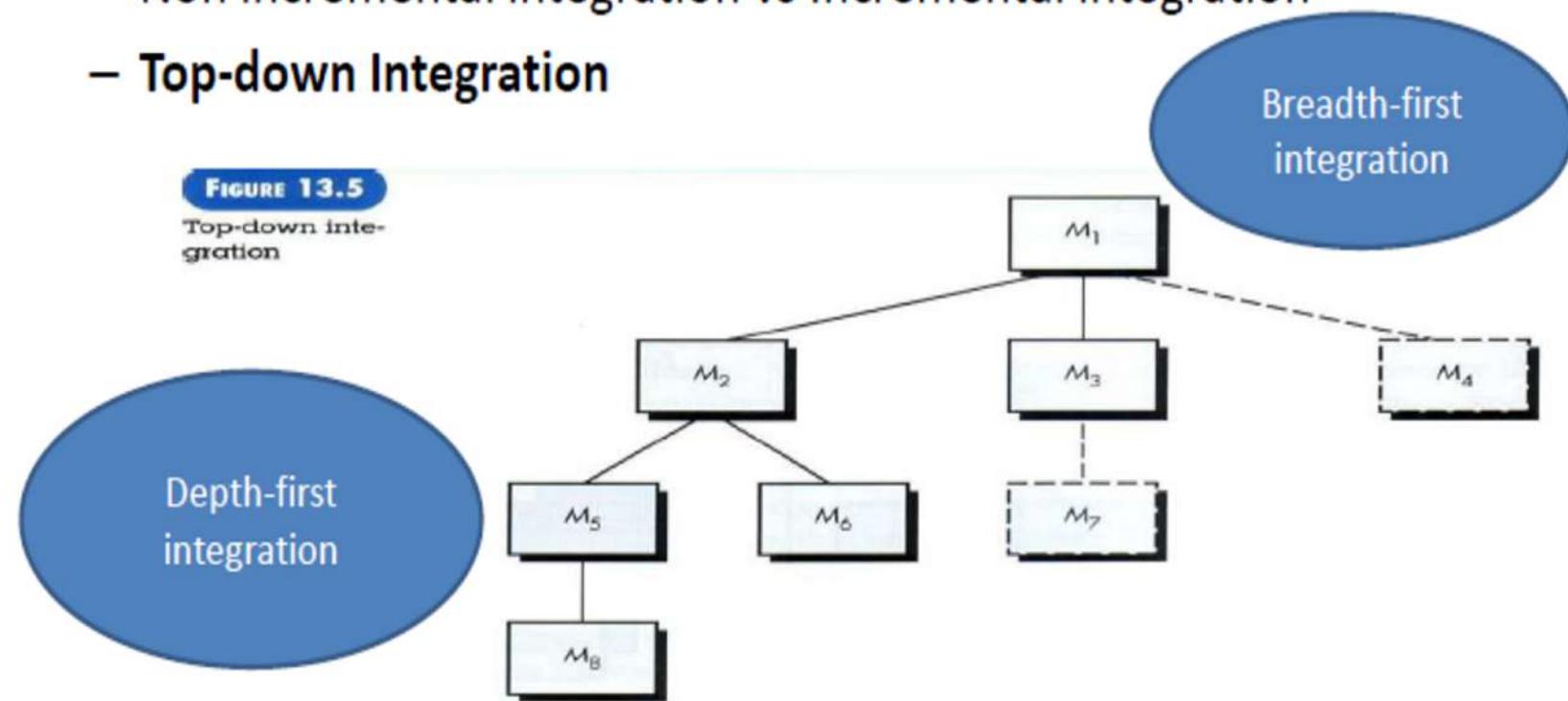
Unit test envi-  
ronment

Drivers and Stubs  
represent overhead

Unit Testing is  
simplified when  
a component  
with high  
cohesion is  
designed



- Integration Testing
  - A systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing
  - Non incremental integration vs Incremental integration
  - **Top-down Integration**

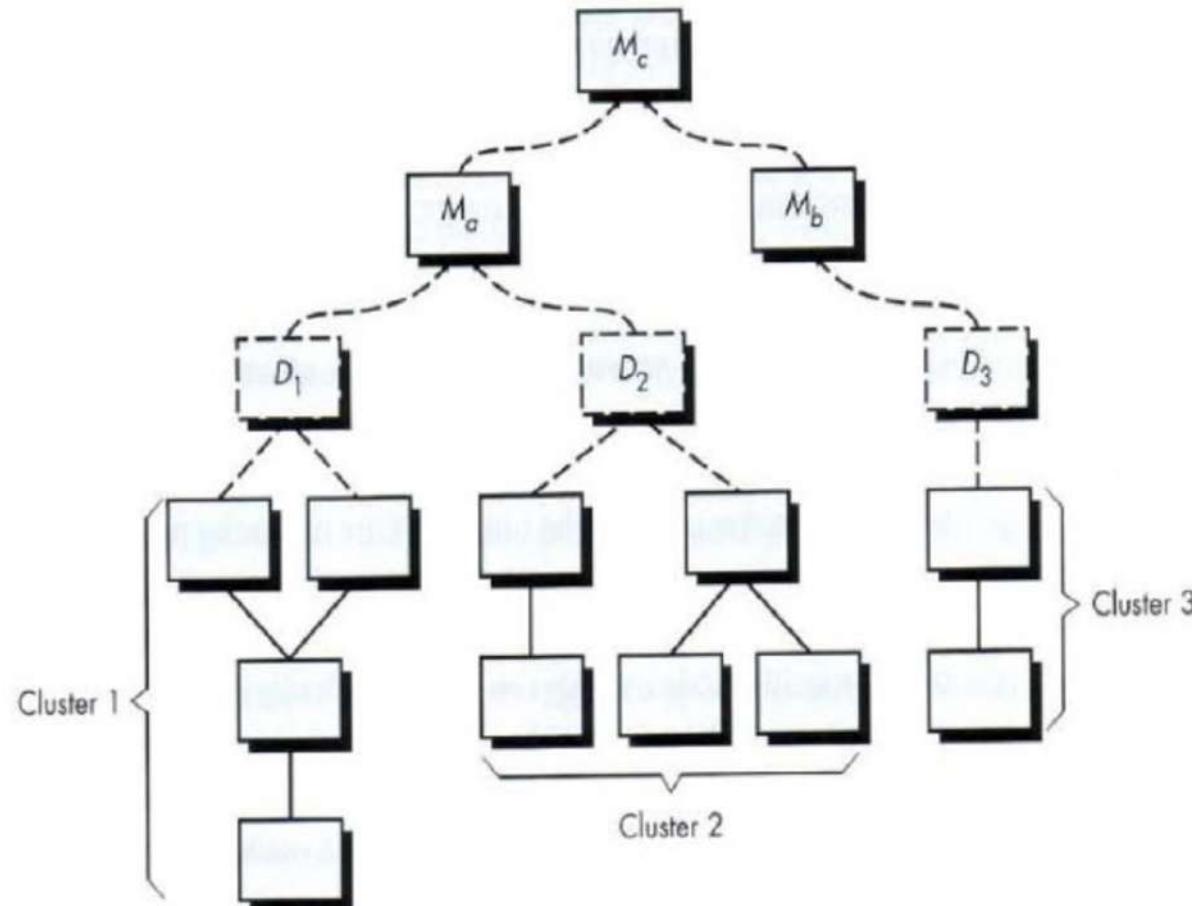


- Integration process performed in a series of 5 steps:
  - Main control module used as a test driver & stubs substituted for all components directly subordinate to the main control module
  - Stubs replaced one at a time with actual components
  - Tests conducted as each component is integrated
  - On completion of each set of tests, another stub is replaced with real component
  - Regression testing conducted to ensure that new errors not been introduced.
- Top-down Integration relatively uncomplicated, but in practice logistical problems arise
- Most common problem: when processing at low levels in the hierarchy is required to test upper levels

- Tester has 3 choices
  - Delay many tests until stubs are replaced with actual modules,
  - Develop stubs that perform limited functions that simulate the actual module (or),
  - Integrate the software from bottom of the hierarchy upward
- Bottom-up Integration
  - Begins construction and testing with atomic modules
  - No need for stubs
- Each time a new module is added as part of integration testing, software changes
  - New data flow paths established,
  - New i/o may occur,
  - New control logic invoked
- Due to these changes, problems persist: hence Regression Testing has to be done

**FIGURE 13.6**

Bottom-up  
integration



- Regression Testing
  - is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
  - is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be conducted manually by re-executing a subset of all test cases or
  - Using automated capture/playback tools
  - Regression test suite contains 3 different classes of test cases:
    - A representative sample of tests that will exercise all software functions
    - Additional tests that focus on s/w functions that are likely to be affected by the change
    - Tests that focus on s/w components that have been changed

- Smoke Testing
  - Designed as a pacing mechanism for time-critical projects, allowing the software team to assess its projects on a frequent basis
  - It should exercise the entire system from end to end
  - Provides a no. of benefits when it is applied on complex, time-critical software engineering projects:
    - Integration risk is minimized
    - Quality of the end product is improved
    - Error diagnosis and corrections are simplified
    - Progress is easier to access

Smoke test is done to make sure that the critical functionalities of the program are working fine, whereas

Sanity testing is done to check that newly added functionalities, bugs, etc., have been fixed.

**Smoke Testing has a goal to verify “stability” whereas  
Sanity Testing has a goal to verify “rationality”.**

Smoke testing is a subset of acceptance testing whereas Sanity testing is a subset of Regression Testing.

- As integration testing is conducted, tester should identify critical modules
- A critical module has one or more of the following characteristics:
  - Addresses several software requirements
  - Has a high level of control
  - Is complex or error-prone or
  - Has definitive performance requirements
- Critical modules should be tested as early as possible
- In addition, regression testing should focus on critical module functions
- Integration Test Documentation: Test SpecificationM

- Integration Testing in the OO Context
- –2 different strategies:
- •**Thread-based Testing**
- –Integrates the set of classes required to respond to one input or event for the system
- –Each thread is integrated and tested individually
- –Regression testing is applied to ensure that no side-effects occur
- •**Use-based Testing**
- –Testing those independent classes that use very few server classes
- –After that dependent classes that uses independent classes are then tested
- •**Cluster Testing**

- **Validation Testing**
  - –Begins at the culmination of Integration Testing, when
  - Individual components are exercised,
  - Software is completely assembled as a package, &
  - Interfacing errors are uncovered and corrected
  - –Focus is on user-visible actions and user-recognizable output from the system
  - –Validation succeeds when software functions in a manner that can be reasonably acceptable by the customer
  - –How to determine the “reasonable expectations”?
  - –Validation Criteria
  - –Configuration Review
  - –Alpha and Beta Testing

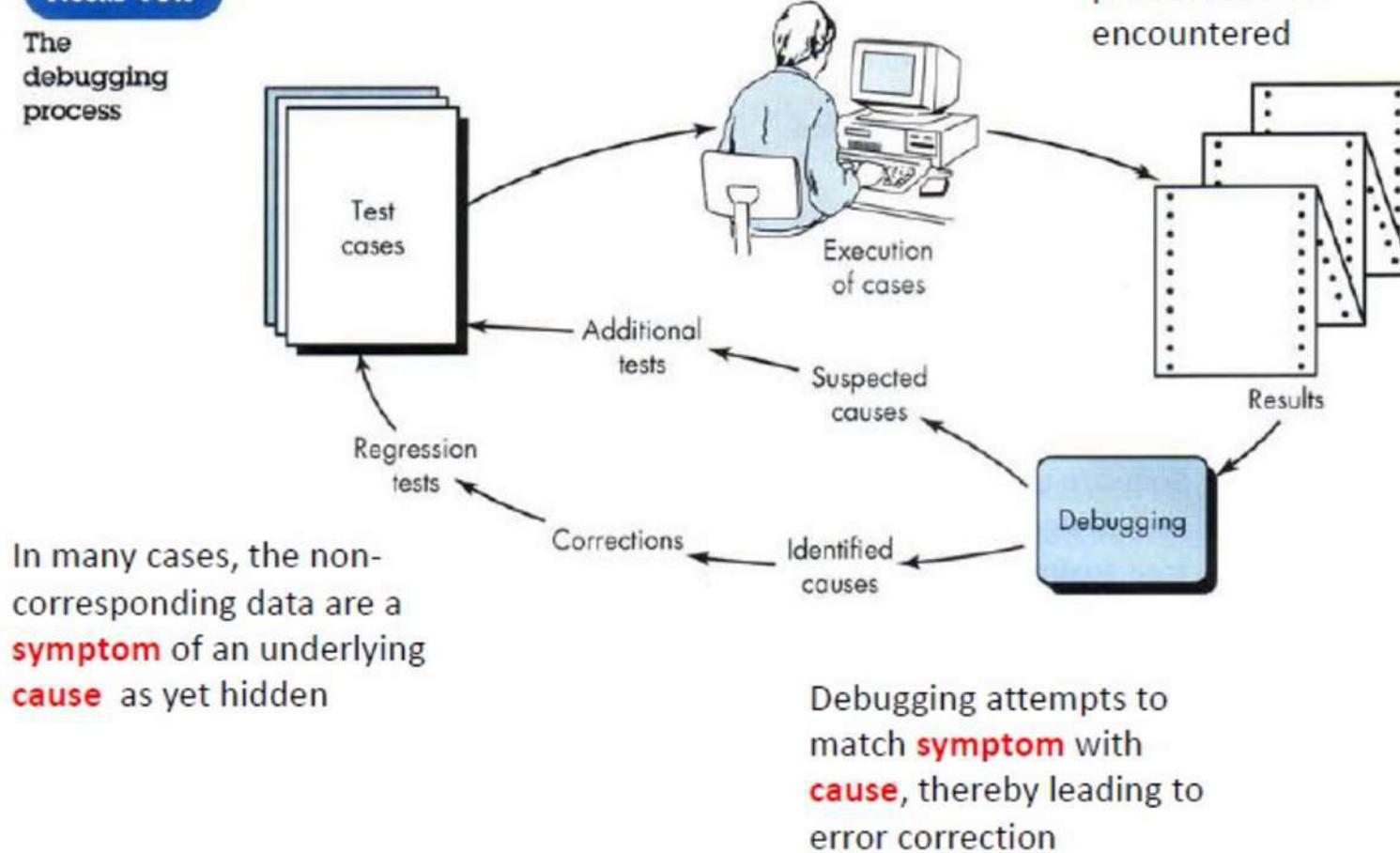
- **System Testing**
  - –A series of different tests whose primary purpose is to fully exercise the computer-based system
    - Types of system testing
      - **Recovery testing**
        - –System must be fault tolerant
        - –In most cases, a system failure corrected within a specific period of time or severe economic damage will occur
        - –Forces the system to fail in a variety of ways and verifies that recovery is properly performed
        - –For automatic recovery: reinitialization, checkpointing mechanisms, data recovery, and restart evaluated for correctness
        - –For manual recovery: MTTR evaluated to determine whether it is within acceptable limits

- **Security Testing**
  - –Any system are subjected to harmful penetration
  - –Penetration spans a broad range of activities:
    - »Hackers penetration to systems for stealing confidential data
    - »Annoyed employees for revenge
    - »For illicit personal gain
  - –verifies that protection mechanisms built into the system will protect it from improper penetration
  - –Testers penetrate the system through various means
  - –System designer to make penetration cost more than the value of the information that will be obtained
- **Stress Testing**
  - –Designed to confront programs with abnormal situations

- **Performance Testing**
  - –Is designed to test the run-time performance of software within the context of an integrated system
  - –Occurs throughout all the steps in the testing process
  - –Often coupled with stress testing and usually require both hardware and software instrumentation
  - –Execution intervals
  - –Log events
- **Debugging**
  - –Is not testing, but occurs as a consequence of successful testing
  - –When a test case uncovers an error, debugging is an action that results in the removal of the error

**FIGURE 13.7**

The debugging process



- Debugging always have one of 2 outcomes:
- •The cause will be found and corrected (OR)
- •The cause will not be found
- –In the latter case, the person performing debugging may suspect a cause,
  - design one or more test cases to help validate the suspicion, and Work toward error correction in an iterative fashion

# TESTING TACTICS

- Testing Tactics –Test Case Design
- •How to write an effective test case?
- •Test case design
- •Characteristics of “Testability”
  - –Software Testability is simply how easily a computer program can be tested
  - –Must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum effort
  - –Operability: “the better it works, the more efficiently it can be tested”
  - –Observability: “What you see is what you test”
  - –Controllability: “the better we can control the software, the more the testing can be automated and optimized”
  - –Decomposability:
  - –Simplicity:
  - –Stability:
  - –Understandability:

- Test characteristics
  - –A good test has a high probability of finding an error
  - –A good test is not redundant
  - –A good test should be “best of breed”
  - –A good test should be neither too simple nor too complex
  - •Black-box Testing and White-Box Testing
  - –BB Testing
  - The Black Box Test is a test that only considers the external behavior of the system; the internal workings of the software is not taken into account. It is a functional test of the software.
  - --WB testing
  - The White Box Test is a method used to test a software taking into consideration its internal functioning. It is a structural test of the software.

- **Black-box test design techniques-**
- Decision table testing
- Equivalence partitioning
- Boundary Value Analysis

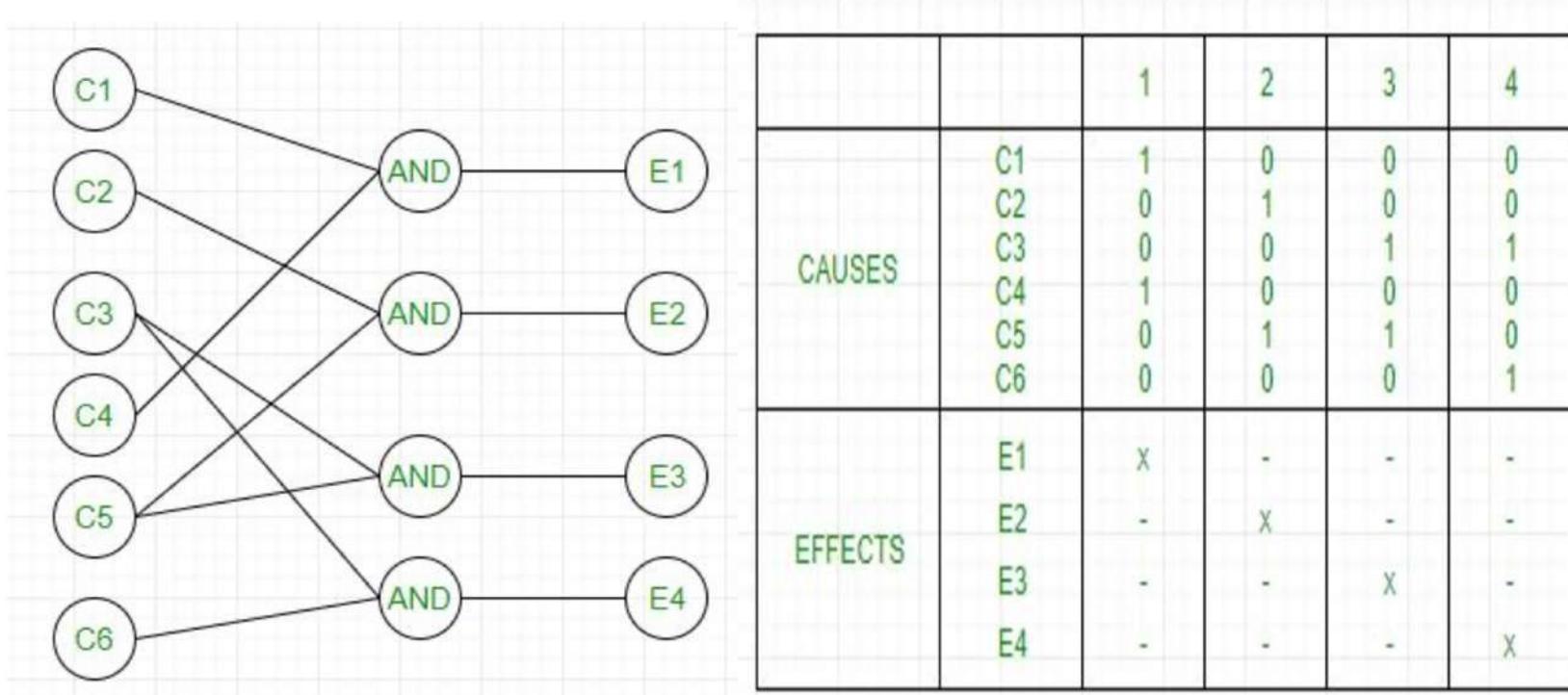
### **Types of Black Box Testing:**

- Functional Testing
- Non-functional testing
- Regression Testing

- **Equivalence partitioning** – It is often seen that many types of inputs work similarly so instead of giving all of them separately we can group them and test only one input of each group. The idea is to partition the input domain of the system into several equivalence classes such that each member of the class works similarly, i.e., if a test case in one class results in some error, other members of the class would also result in the same error.
- The technique involves two steps:
- **Identification of equivalence class** – Partition any input domain into a minimum of two sets: **valid values** and **invalid values**. For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.
- **Generating test cases** – (i) To each valid and invalid class of input assign a unique identification number. (ii) Write a test case covering all valid and invalid test cases considering that no two invalid inputs mask each other. To calculate the square root of a number, the equivalence classes will be **(a) Valid inputs:**
  - The whole number which is a perfect square- output will be an integer.
  - The whole number which is not a perfect square- output will be a decimal number.
  - Positive decimals
  - Negative numbers(integer or decimal).
  - Characters other than numbers like “a”, “!”, “;”, etc.

- **Boundary value analysis** – Boundaries are very good places for errors to occur. Hence if test cases are designed for boundary values of the input domain then the efficiency of testing improves and the probability of finding errors also increases. For example – If the valid range is 10 to 100 then test for 10,100 also apart from valid and invalid inputs.
- **Cause effect Graphing** – This technique establishes a relationship between logical input called causes with corresponding actions called the effect. The causes and effects are represented using Boolean graphs. The following steps are followed:
  - Identify inputs (causes) and outputs (effect).
  - Develop a cause-effect graph.
  - Transform the graph into a decision table.
  - Convert decision table rules to test cases.

- For example, in the following cause-effect graph:
- It can be converted into a decision table like:



- Each column corresponds to a rule which will become a test case for testing. So there will be 4 test cases.

- **Black Box Testing Type**
- The following are the several categories of black box testing:
- Functional Testing
- Regression Testing
- Nonfunctional Testing (NFT)
- **Functional Testing:** It determines the system's software functional requirements.

**Regression Testing:** It ensures that the newly added code is compatible with the existing code. In other words, a new software update has no impact on the functionality of the software. This is carried out after a system maintenance operation and upgrades.

**Nonfunctional Testing:** Nonfunctional testing is also known as NFT. This testing is not functional testing of software. It focuses on the software's performance, usability, and scalability.

- **Tools Used for Black Box Testing:**
  - Appium
  - Selenium
  - Microsoft Coded UI
  - AppliTools
  - HP QTP.

- **Features of black box testing:**
- **Independent testing:** Black box testing is performed by testers who are not involved in the development of the application, which helps to ensure that testing is unbiased and impartial.
- **Testing from a user's perspective:** Black box testing is conducted from the perspective of an end user, which helps to ensure that the application meets user requirements and is easy to use.
- **No knowledge of internal code:** Testers performing black box testing do not have access to the application's internal code, which allows them to focus on testing the application's external behavior and functionality.
- **Requirements-based testing:** Black box testing is typically based on the application's requirements, which helps to ensure that the application meets the required specifications.

- **Features of black box testing: -contd...**
- **Different testing techniques:** Black box testing can be performed using various testing techniques, such as functional testing, usability testing, acceptance testing, and regression testing.
- **Easy to automate:** Black box testing is easy to automate using various automation tools, which helps to reduce the overall testing time and effort.
- **Scalability:** Black box testing can be scaled up or down depending on the size and complexity of the application being tested.
- **Limited knowledge of application:** Testers performing black box testing have limited knowledge of the application being tested, which helps to ensure that testing is more representative of how the end users will interact with the application.

- **Advantages of Black Box Testing:**
- The tester does not need to have more functional knowledge or programming skills to implement the Black Box Testing.
- It is efficient for implementing the tests in the larger system.
- Tests are executed from the user's or client's point of view.
- Test cases are easily reproducible.
- It is used in finding the ambiguity and contradictions in the functional specifications.

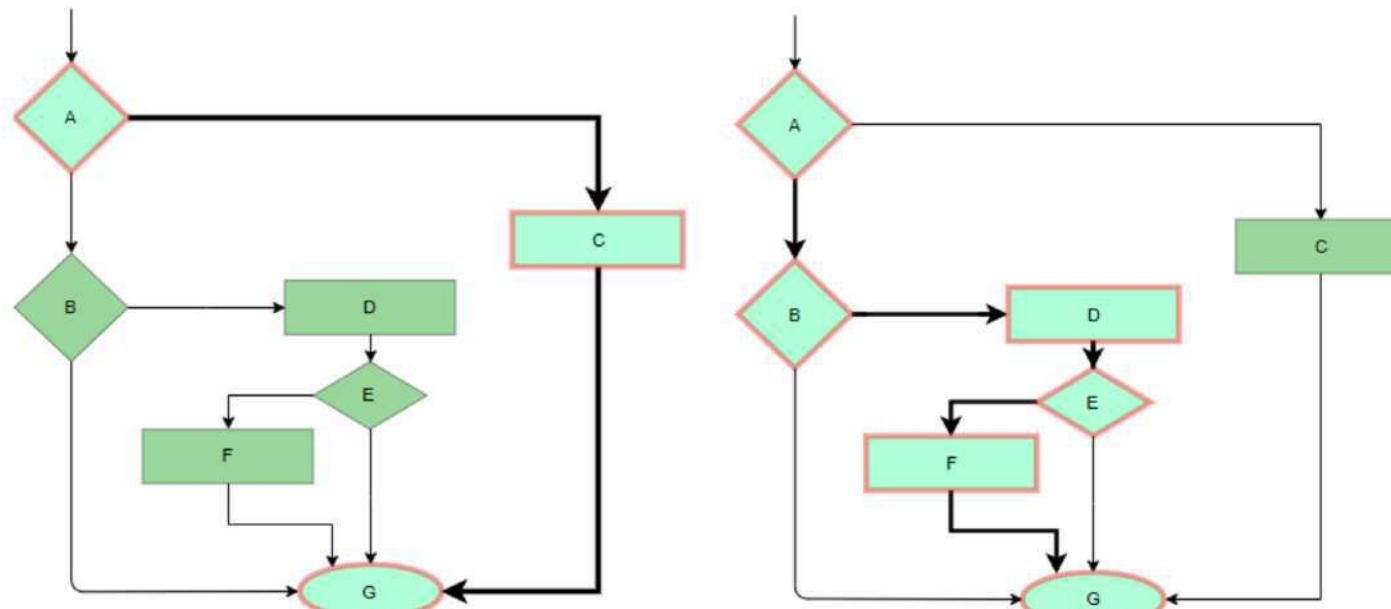
- **Disadvantages of Black Box Testing:**
- There is a possibility of repeating the same tests while implementing the testing process.
- Without clear functional specifications, test cases are difficult to implement.
- It is difficult to execute the test cases because of complex inputs at different stages of testing.
- Sometimes, the reason for the test failure cannot be detected.
- Some programs in the application are not tested.
- It does not reveal the errors in the control structure.
- Working with a large sample space of inputs can be exhaustive and consumes a lot of time

## White-Box Testing

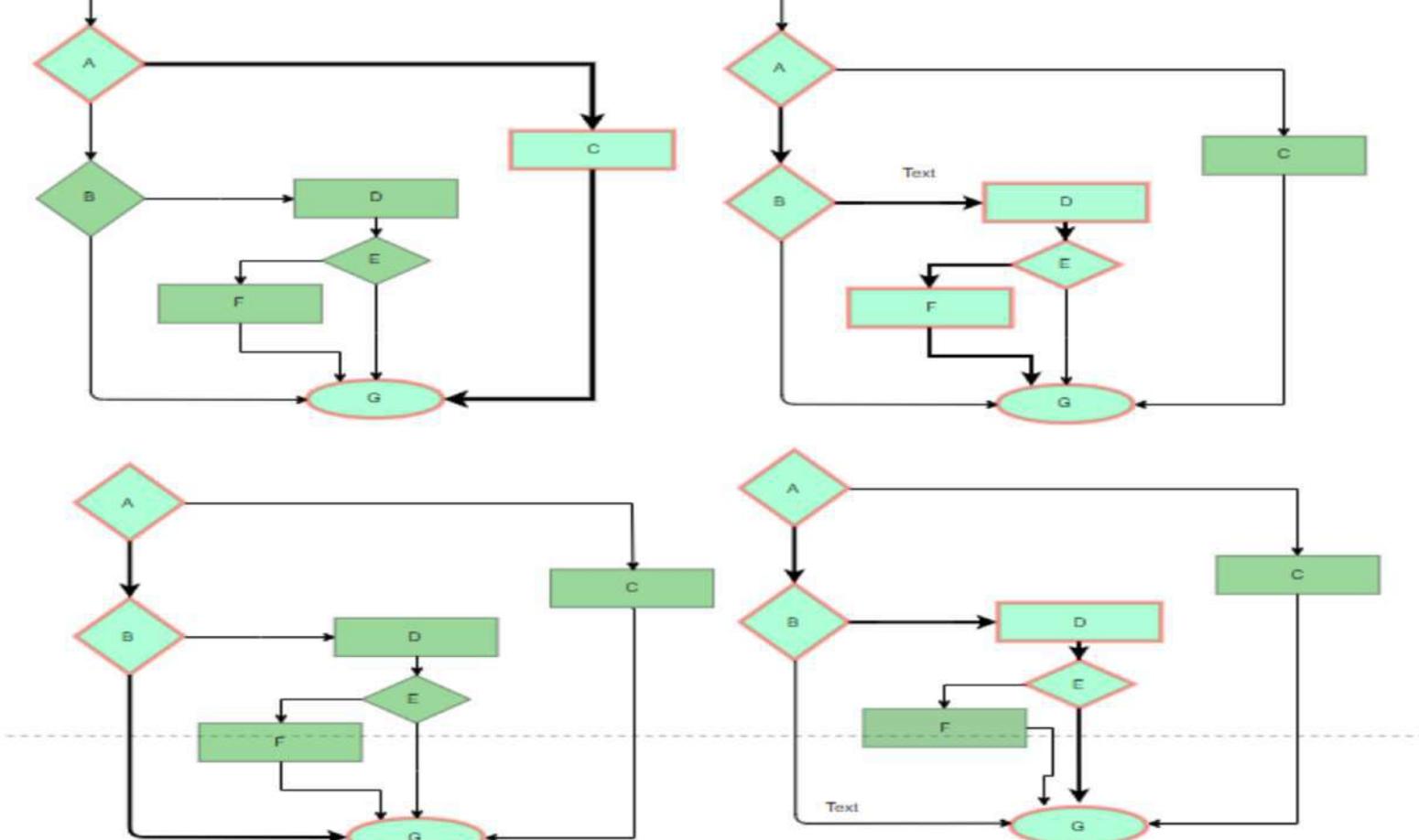
- –Also called as Glass-Box testing, is a test-case design philosophy that uses the control structure described as part of the component-level design to derive test cases
- •Guarantee that all independent paths within a module have been exercised at least once
- •Exercise all logical decisions on their true and false sides
- •Execute all loops at their boundaries and within their operational bounds
- •Exercise internal data structures to ensure their validity
- –Basic Path Testing
- •Enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths

- **White-box test design techniques-**
- Control flow testing
- Data flow testing
- Branch testing
- **Types of White Box Testing:**
- Path Testing
- Loop Testing
- Condition testing

- **Statement coverage:** In this technique, the aim is to traverse all statements at least once. Hence, each line of code is tested. In the case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



- **Branch Coverage:** In this technique, test cases are designed so that each branch from all decision points is traversed at least once. In a flowchart, all edges must be traversed at least once.



- **Condition Coverage:** In this technique, all individual conditions must be covered as shown in the following example:
  - READ X, Y
  - IF( $X == 0 \parallel Y == 0$ )
  - PRINT ‘0’
  - #TC1 – X = 0, Y = 55
  - #TC2 – X = 5, Y = 0
- **Multiple Condition Coverage:** In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let’s consider the following example:
  - READ X, Y
  - IF( $X == 0 \parallel Y == 0$ )
  - PRINT ‘0’
  - #TC1: X = 0, Y = 0
  - #TC2: X = 0, Y = 5
  - #TC3: X = 55, Y = 0
  - #TC4: X = 55, Y = 5

- **Basis Path Testing:** In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.
- **Steps:**
  - Make the corresponding control flow graph
  - Calculate the cyclomatic complexity(A software metric that provides the quantitative measure of the logical complexity of a program)
  - Find the independent paths
  - Design test cases corresponding to each independent path
  - $V(G) = P + 1$ , where P is the number of predicate nodes in the flow graph
  - $V(G) = E - N + 2$ , where E is the number of edges and N is the total number of nodes
  - $V(G) = \text{Number of non-overlapping regions in the graph}$

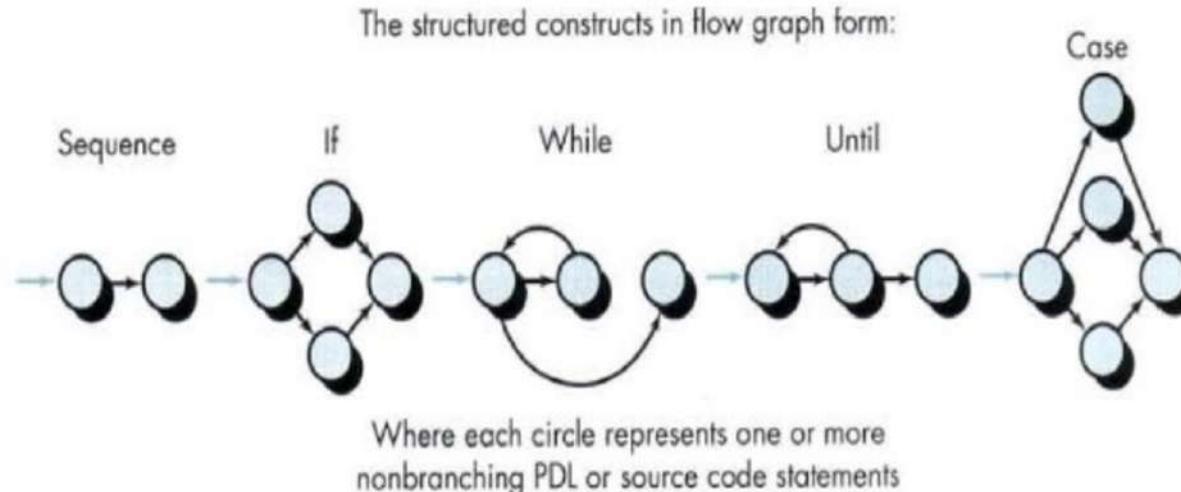
- Test cases derived to exercise the basis set are guaranteed to execute every statement at least one time during testing

## - Flow Graph notation

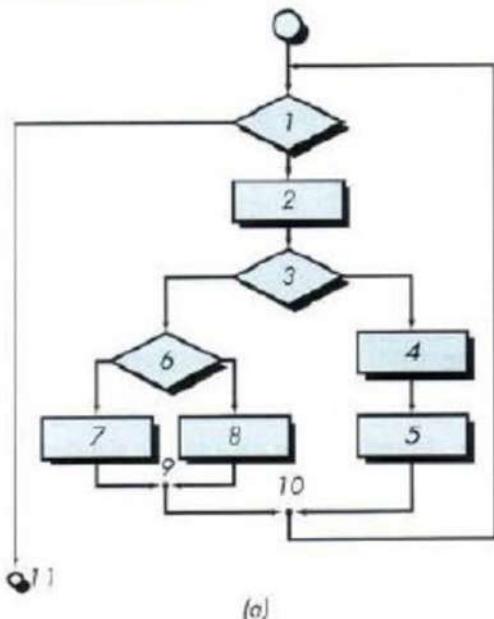
- Flow Graph depicts logical control flow

**FIGURE 14.1**

Flow graph  
notation

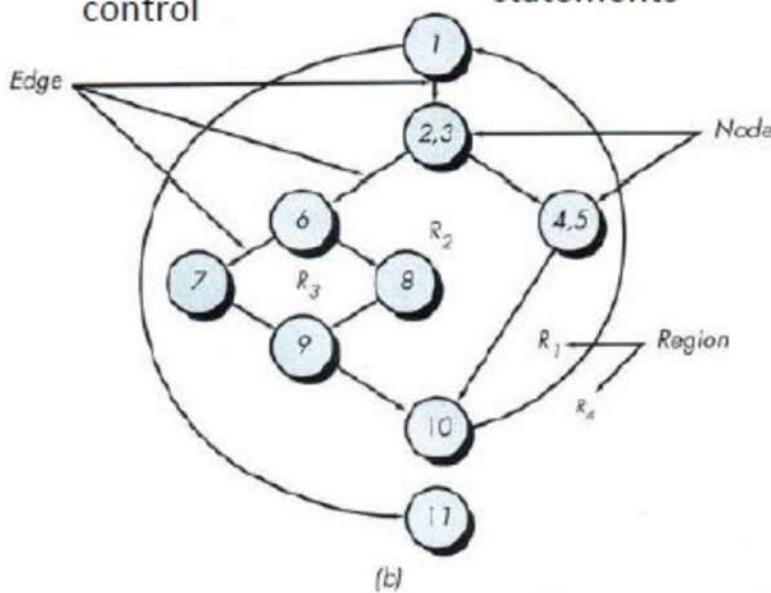


**FIGURE 14.1** (a) Flowchart and (b) flow graph



Maps the flowchart into a corresponding flowgraph

Or links represent flow of control  
 Each circle, called a Flowgraph node represents one or more procedural statements

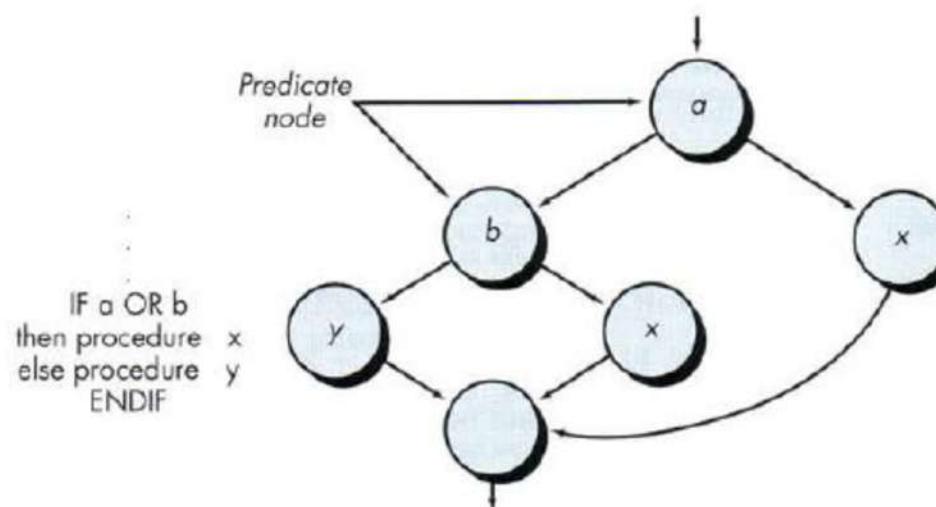


A sequence of process boxes and a decision diamond can map into a single node

- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated
- Each node that contains a condition is called a **Predicate Node** & is characterized by 2 or more edges deriving from it

**FIGURE 14.3**

Compound logic



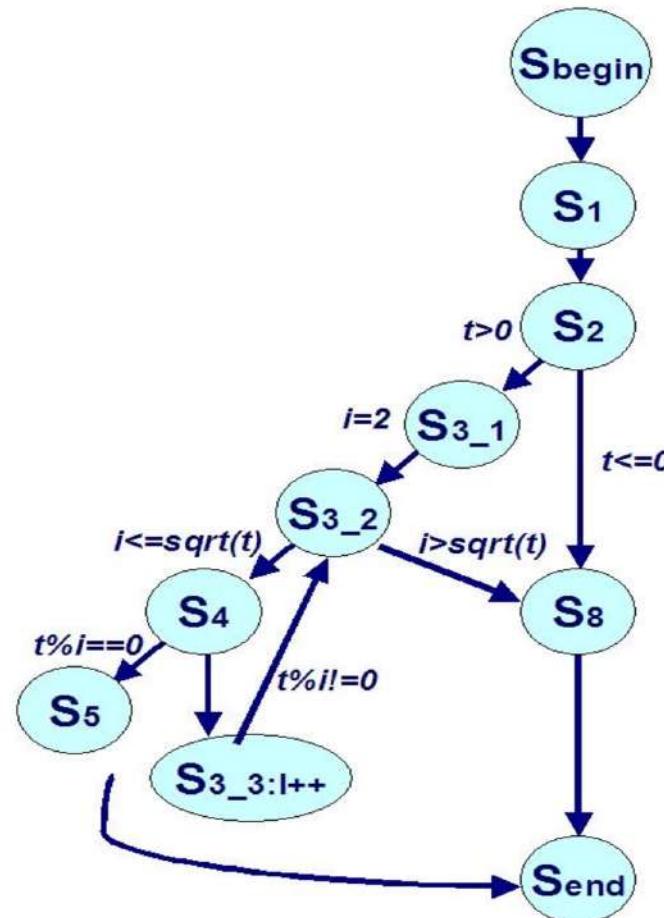
- Independent Program Paths
    - Any path through the program that introduces at least one new set of processing statements or a new condition
    - An independent path must move along at least one edge that has not been traversed before the path is defined
    - Example: from fig. 14.2 (b)
      - Path 1: 1-11
      - Path 2: 1-2-3-4-5-10-11
      - Path 3: 1-2-3-6-8-9-10-1-11
      - Path 4: 1-2-3-6-7-9-10-1-11
    - Path 1,2,3, and 4 constitute a basis set for the flow graph
    - This basis set is not unique
- } Each new path introduces a new edge

- 1.The no. of regions corresponds to the cyclomaticcomplexity – $V(G)$
- 2. $V(G)$  for a flow graph, G, is defined as
- $V(G) = E - N + 2$
- E –no. of flow graph edges
- N –no. of flow graph nodes
- 3.CyclomaticComplexity,  $V(G)$ , for a flow graph, G, is also defined as,  
 $V(G) = P + 1$
- P –no. of predicate nodes contained in the flow graph, G.
- 4.From Fig.14.2b, the cyclomaticcomplexity can be computed using each of the algorithms
- flow graph has 4 regions
- $V(G) = 11 \text{ edges} - 9 \text{ nodes} + 2 = 4$
- $V(G) = 3 \text{ predicate nodes} + 1 = 4$

```

S1 Boolean Composite (integer t)
S2 if (t > 0) {
S3   for (i=2; i<=sqrt(t); i++) {
S4     if (t%i==0) {
S5       return true;
S6     }
S7   }
S8 return false;

```



Pseudo Code to find out if an Integer is a composite Number

PROCEDURE average;

- \* This procedure computes the average of 100 or fewer numbers that lie between bounding values; it also computes the sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;

INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;

TYPE average, total.input, total.valid;

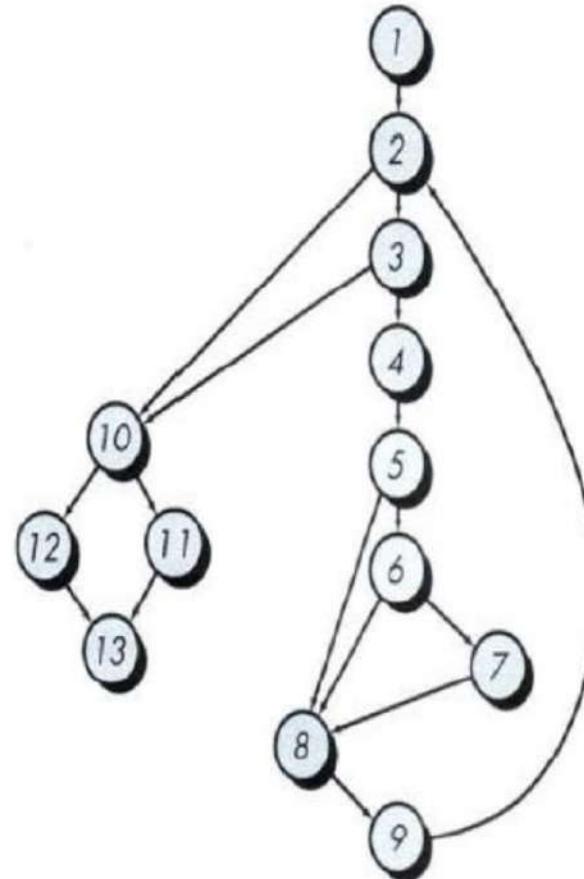
minimum, maximum, sum IS SCALAR;

TYPE i IS INTEGER;

```

1 { i = 1;
  total.input = total.valid = 0; 2
  sum = 0;
  DO WHILE value[i] <> -999 AND total.input < 100 3
    4 increment total.input by 1;
    IF value[i] >= minimum AND value[i] <= maximum 6
      5 { THEN increment total.valid by 1;
        sum = sum + value[i]
      ELSE skip
    8 { ENDIF
    increment i by 1;
  9 ENDDO
  IF total.valid > 0 10
    11 THEN average = sum / total.valid;
  12 ELSE average = -999;
  13 ENDIF
END average

```



- **Data Flow Testing** is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.

It is concerned with:

- Statements where variables receive values,
- Statements where these values are used or referenced.
- To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S-

$\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains the definition of } X\}$

$\text{USE}(S) = \{X \mid \text{statement } S \text{ contains the use of } X\}$

Variable occurrence can be one of the following three types

- **def** represents the definition of a variable. The variable on the left-hand side of an assignment statement is the one getting defined .
- **c-use** represents computational use of a variable. Any statement (e.g., read, write, an assignment) that uses the value of variables for computational purposes is said to be making c-use of the variables. In an assignment statement, all variables on the right-hand side have a c-use occurrence. In a read and a write statement, all variable occurrences are of this type.
- **p-use** represents predicate use. These are all the occurrences of the variables in a predicate (i.e., variables whose values are used for computing the value of the predicate), which is used for transfer of control.

- If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.
- Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program. Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:
  - A variable is defined but not used or referenced,
  - A variable is used but never defined,
  - A variable is defined twice before it is used

- **Advantages of Data Flow Testing:**

Data Flow Testing is used to find the following issues-

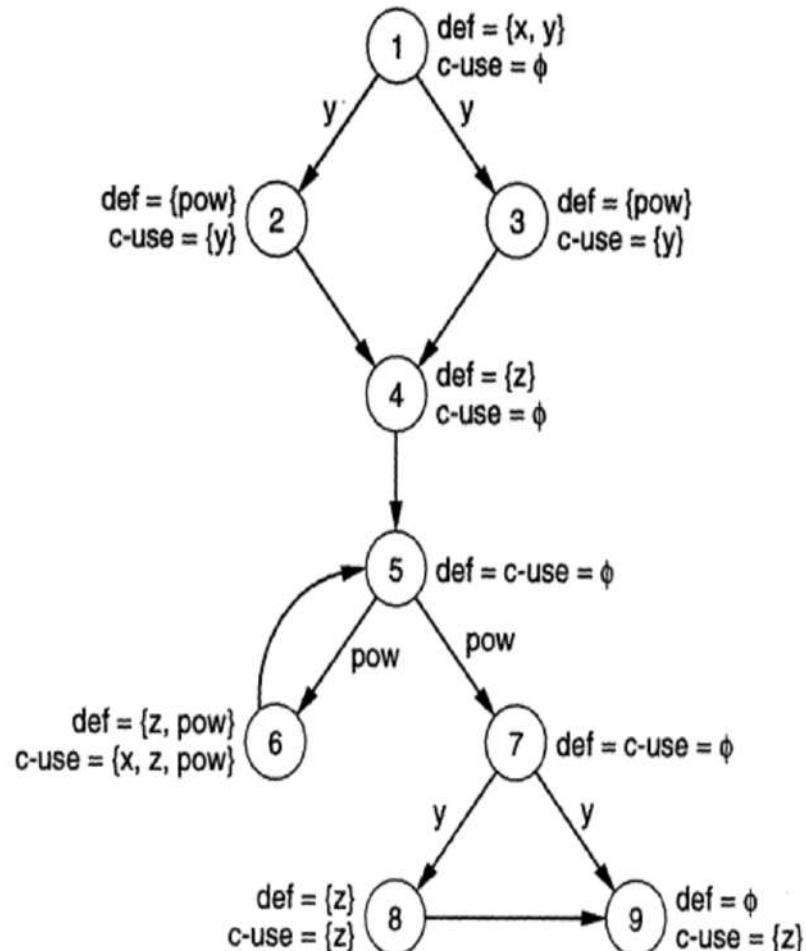
- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is used,
- Deallocating a variable before it is used.
- **Disadvantages of Data Flow Testing**
- Time consuming and costly process
- Requires knowledge of programming languages

```

1. scanf(x, y); if (y <= 0)
2.   pow = 0 - y;
3. else pow = y;
4. z = 1.0;
5. while (pow != 0)
6.   { z = z * x; pow = pow - 1; }
7. if (y <= 0)
8.   z = 1.0/z;
9. printf(z);

```

(node, var)	dcu	dpu
(1, x)	{6}	$\phi$
(1, y)	{2, 3}	$\{(1,2), (1,3), (7, 8), (7, 9)\}$
(2, pow)	{6}	$\{(5, 6), (5, 7)\}$
(3, pow)	{6}	$\{(5, 6), (5, 7)\}$
(4, z)	{6, 8, 9}	$\phi$
(6, z)	{6, 8, 9}	$\phi$
(6, pow)	{6}	$\{(5, 6), (5, 7)\}$
(8, z)	{9}	$\phi$



- **Loop Testing:** Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.
  - **Simple loops:** For simple loops of size  $n$ , test cases are designed that:
    - Skip the loop entirely
    - Only one pass through the loop
    - 2 passes
    - $m$  passes, where  $m < n$
    - $n-1$  and  $n+1$  passes
  - **Nested loops:** For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.
  - **Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

- **White Testing is Performed in 2 Steps:**
  - 1. Tester should understand the code well
  - 2. Tester should write some code for test cases and execute them
- **Tools required for White box Testing:**
  - PyUnit
  - Sqlmap
  - Nmap
  - Parasoft Jtest
  - Nunit
  - VeraUnit
  - CppUnit

- **Features of white box testing:**
- **Code coverage analysis:** White box testing helps to analyze the code coverage of an application, which helps to identify the areas of the code that are not being tested.
- **Access to the source code:** White box testing requires access to the application's source code, which makes it possible to test individual functions, methods, and modules.
- **Knowledge of programming languages:** Testers performing white box testing must have knowledge of programming languages like Java, C++, Python, and PHP to understand the code structure and write tests.
- **Identifying logical errors:** White box testing helps to identify logical errors in the code, such as infinite loops or incorrect conditional statements.

- **Features of white box testing: contd...**
- **Integration testing:** White box testing is useful for integration testing, as it allows testers to verify that the different components of an application are working together as expected.
- **Unit testing:** White box testing is also used for unit testing, which involves testing individual units of code to ensure that they are working correctly.
- **Optimization of code:** White box testing can help to optimize the code by identifying any performance issues, redundant code, or other areas that can be improved.
- **Security testing:** White box testing can also be used for security testing, as it allows testers to identify any vulnerabilities in the application's code.

- **Advantages:**
  - White box testing is thorough as the entire code and structures are tested.
  - It results in the optimization of code removing errors and helps in removing extra lines of code.
  - It can start at an earlier stage as it doesn't require any interface as in the case of black box testing.
  - Easy to automate.
  - White box testing can be easily started in Software Development Life Cycle.
  - Easy Code Optimization.
- **Some of the advantages of white box testing include:**
  - Testers can identify defects that cannot be detected through other testing techniques.
  - Testers can create more comprehensive and effective test cases that cover all code paths.
  - Testers can ensure that the code meets coding standards and is optimized for performance.

- **However, there are also some disadvantages to white box testing, such as:**
- Testers need to have programming knowledge and access to the source code to perform tests.
- Testers may focus too much on the internal workings of the software and may miss external issues.
- Testers may have a biased view of the software since they are familiar with its internal workings.  
Overall, white box testing is an important technique in software engineering, and it is useful for identifying defects and ensuring that software applications meet their requirements and specifications at the code level

- **Disadvantages:**
- It is very expensive.
- Redesigning code and rewriting code needs test cases to be written again.
- Testers are required to have in-depth knowledge of the code and programming language as opposed to black-box testing.
- Missing functionalities cannot be detected as the code that exists is tested.
- Very complex and at times not realistic.
- Much more chances of Errors in production.

# PRODUCT METRICS

- PRODUCT METRICS

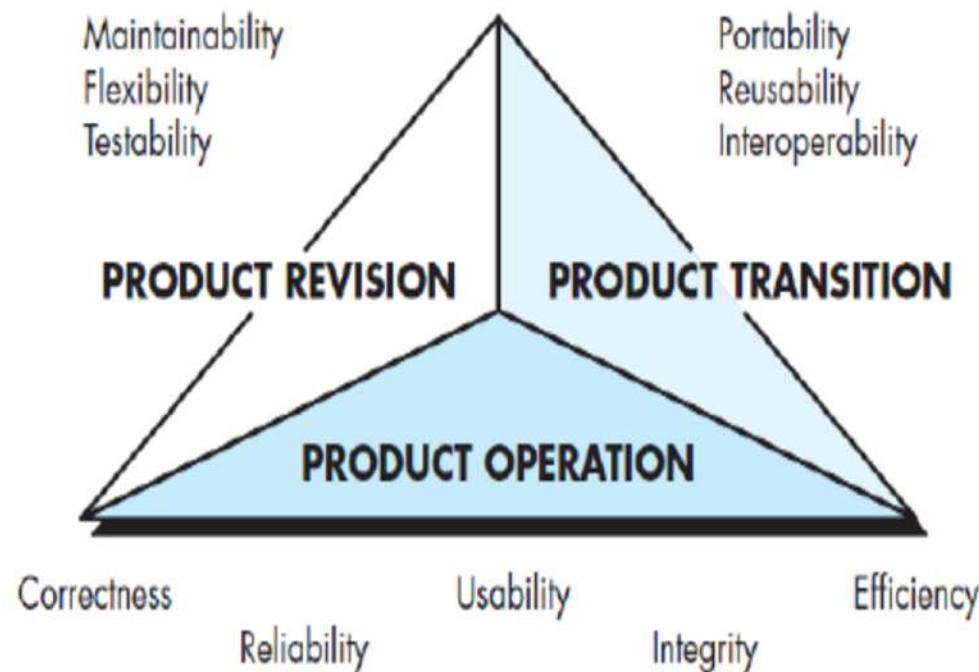
Direct measures are unusual in the software world

- Software metrics are indirect
- Measures used to assess the quality of the software product as it is being engineered
- Software Quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- Software requirements: foundation from which quality is measured; Lack of conformance to requirements is lack of quality
- Specified standards: define a set of development criteria that guide the manager in which the software is engineered
- If software conforms to its explicit requirements but fails to meet implicit requirements, quality is compromised

- McCall's Quality factors
  - propose a useful categorization of factors that affect software quality.
  - focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments

**FIGURE 19.1**

McCall's  
software  
quality factors



- McCall and his colleagues provide the following descriptions:
- *–Correctness* -The extent to which a program satisfies its specification and fulfills the customer's mission objectives
- *–Reliability*.
- *–Efficiency*.
- *–Integrity* -Extent to which access to software or data by unauthorized persons can be controlled.
- *–Usability*
- *–Maintainability*
- *–Flexibility*
- *–Testability*
- *–Portability*.
- *–Reusability*
- *–Interoperability*
- Difficult to develop direct measures of these quality factors

- ISO 9126 Quality Factors
  - Was developed to identify quality attributes for computer software
  - **Functionality:** degree to which the software satisfies stated needs as indicated by the following sub-attributes:
    - suitability, accuracy, interoperability, compliance, and security.
  - **Reliability:** amount of time that the software is available for use as indicated by the following sub-attributes
    - Maturity, fault tolerance, recoverability
  - **Usability:** degree to which the software is easy to use as indicated by the following sub-attributes
    - understandability, learnability, operability

- –**Efficiency:** degree to which the software makes optimal use of system resources as indicated by the following sub-attributes
  - Time behavior, resource behavior.
- –**Maintainability:**The ease with which repair may be made to the software as indicated by the following sub-attributes
  - analyzability, changeability, stability, testability.
- –**Portability:** The ease with which the software can be transposed from one environment to another as indicated by the following sub-attributes
  - adaptability, installability, conformance, replaceability
  - The transition to a Quantitative view
- –Quality is subjective, a more precise definition of software quality is needed as a way to derive quantitative measurements of software quality for objective analysis

- A framework for Product Metrics
- –Within the software engineering context,
- •**Measure**: provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- •**Measurement**: the act of determining a measure.
- •**Metric**: “a quantitative measure of the degree to which a system, component, or process possesses a given attribute”.
- –A software engineer collects measures and develops metrics so that indicators will be obtained
- –**Indicator**: a metric or combination of metrics that provides insight into the software process, a software project, or the product itself.
- –When a single data point has been collected (e.g., the number of errors uncovered within a single software component), a measure has been established

- Measurement occurs as the result of the collection of one or more data points
- (a number of component reviews and unit tests are investigated to collect measures of the number of errors for each).
- –Software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test)
- –What are the steps of an effective measurement process?
- •Formulation
- •Collection
- •Analysis
- •Interpretation
- •Feedback

- Software metrics are useful only if they are characterized effectively & validated so that their worth is proven:
- A metric should have desirable mathematical properties
- When a metric represents a software characteristic that increases when positive traits occur or decrease when undesirable traits are encountered, the value of metric should increase or decrease in the same manner

- Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions
- •The Product Metrics Landscape
- –**Metrics for the Analysis model**
- •Functionality delivered
- •System size
- •Specification quality
- –**Metrics for the Design model**
- •Architectural metrics
- •Component-level metrics
- •Interface design metrics
- •Specialized OO design metrics

- **-Metrics for Source code**

- Halstead metrics

- Complexity metrics

- Length metrics

- **-Metrics for testing**

- Statement and Branch coverage metrics

- Defect-related metrics

- Testing Effectiveness

- In-process metrics

- **-Metrics for Maintenance**

- SMI

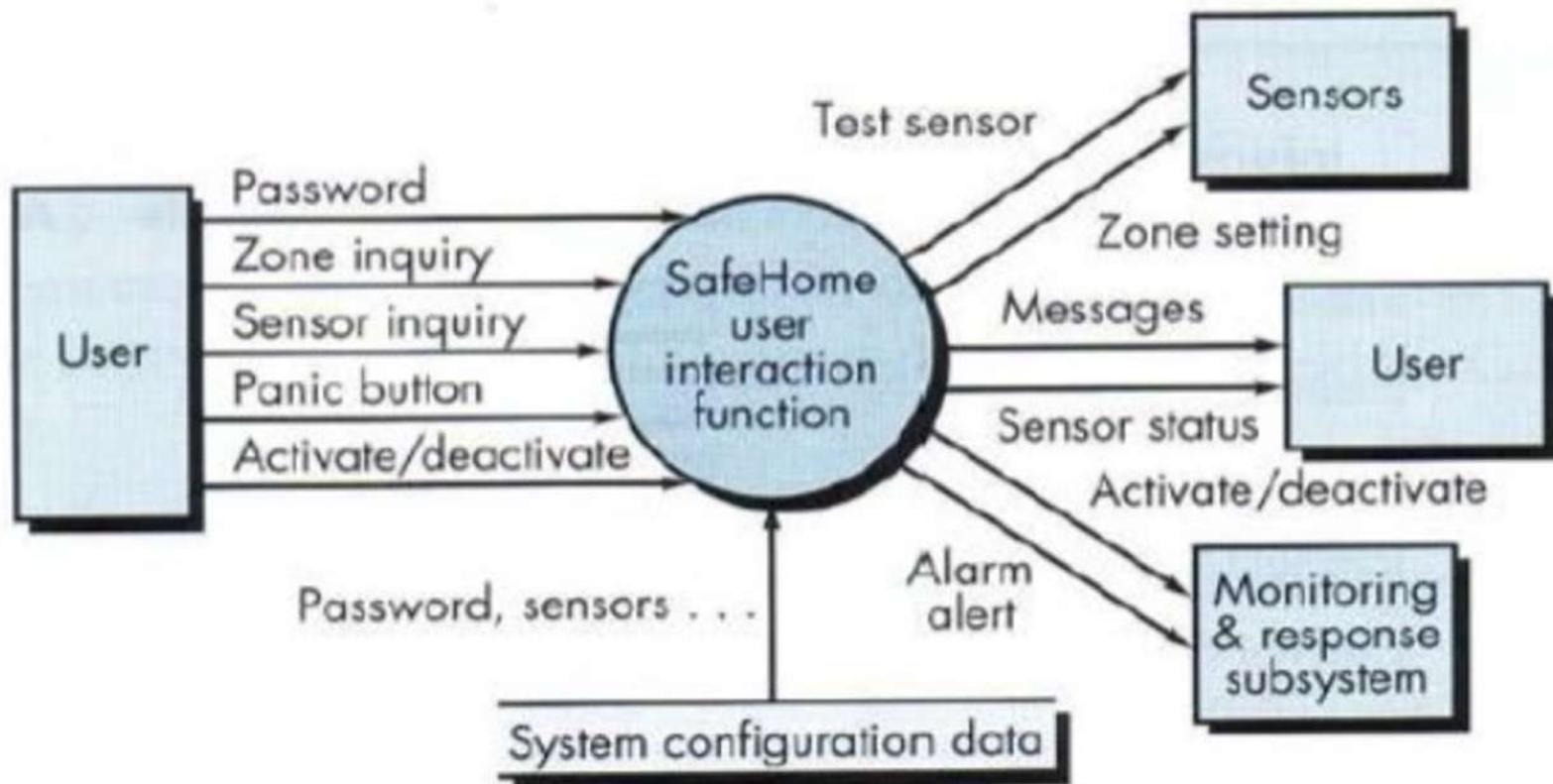
# METRICS FOR REQUIREMENT MODEL

- Function-based Metrics
- •FP metric
- •Function points derived using an empirical relationship based on direct measures of software's information domain & assessments of software complexity
- •Information domain values:
  - –No. of external inputs
  - –No. of external outputs
  - –No. of external queries
  - –No. of internal logic files (ILF)
  - –No. of external interface files (EIF)

- Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average or complex
- To compute FPs, the following relationship holds:
  - $FP = \text{count total} \times [0.65 + 0.01 \times \sum (F_i)]$
  - $F_i$  ( $i=1$  to  $14$ ) are value adjustment factors (VAF) based on responses to the following questions:

**FIGURE 15.2**

Computing function points	Information Domain Value	Count	Weighting factor			=
			Simple	Average	Complex	
	External Inputs (EIs)	×	3	4	6	=
	External Outputs (EOs)	×	4	5	7	=
	External Inquiries (EQs)	×	3	4	6	-
	Internal Logical Files (ILFs)	×	7	10	15	=
	External Interface Files (EIFs)	×	5	7	10	=
Count total			→			



- EI- 3
- EO -2
- EQ-2
- ILF-1
- EIF-3

**FIGURE 15.2**

 Computing  
 function points

Information Domain Value	Count	Weighting factor			=
		Simple	Average	Complex	
External Inputs (EIs)	X	3	4	6	=
External Outputs (EOs)	X	4	5	7	=
External Inquiries (EQs)	X	3	4	6	=
Internal Logical Files (ILFs)	X	7	10	15	=
External Interface Files (EIFs)	X	5	7	10	=
Count total					

- $FP = \text{count total} \times [0.65 + 0.01 \times \sum (F_i)]$
- $F_i$  ( $i=1$  to  $14$ ) are value adjustment factors (VAF) based on responses to the following questions:

- A system has 12 external inputs, 24 external outputs, fields 30 different external queries, manages 4 internal logical files, and interfaces with 6 different legacy systems (6 EIFs). All of these data are of average complexity and the overall system is relatively simple. Compute FP for the system.

- **Metrics for DESIGN**
  - Architectural Design Metrics
  - Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. These metrics are “black box” in the sense that they do not require any knowledge of the inner workings of a particular software component.
  - Card and Glass [Car90] define three software design complexity measures:
  - **structural complexity, data complexity, and system complexity.**
  - For hierarchical architectures (e.g., call-and-return architectures), *structural complexity* of a module  $i$  is defined in the following manner:

For hierarchical architectures (e.g., call-and-return architectures), *structural complexity* of a module  $i$  is defined in the following manner:

$$S(i) = f_{\text{out}}^2(i) \quad (30.2)$$

where  $f_{\text{out}}(i)$  is the fan-out<sup>6</sup> of module  $i$ .

*Data complexity* provides an indication of the complexity in the internal interface for a module  $i$  and is defined as

$$D(i) = \frac{v(i)}{|f_{\text{out}}(i) + 1|} \quad (30.3)$$

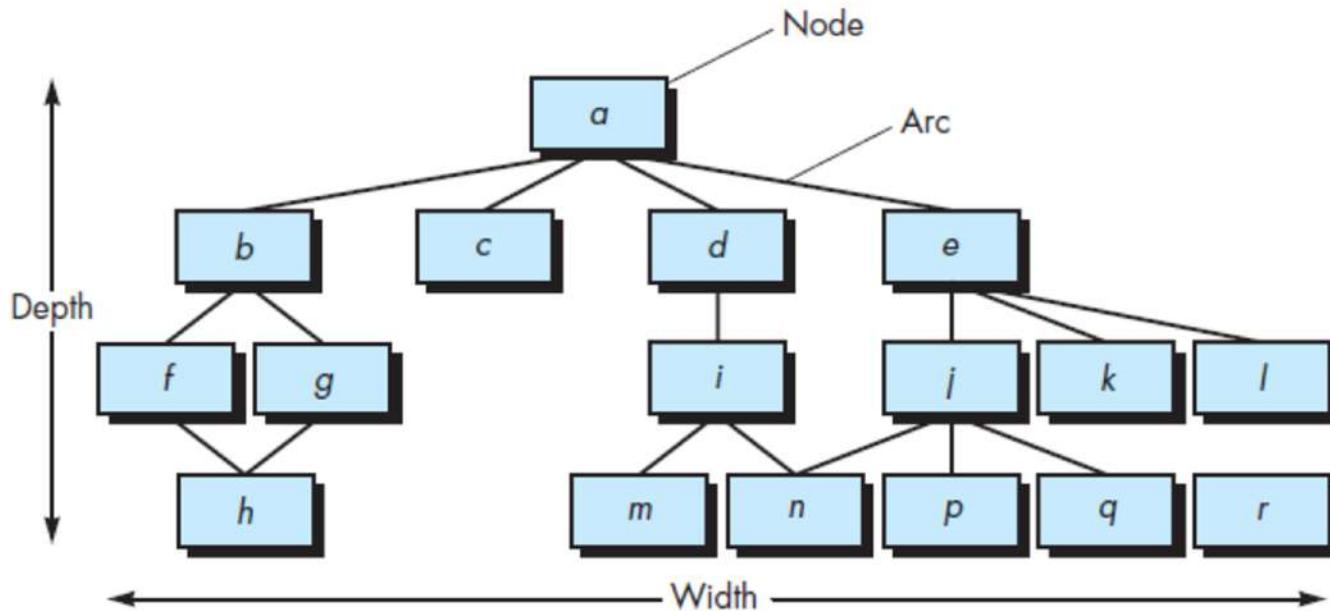
where  $v(i)$  is the number of input and output variables that are passed to and from module  $i$ .

Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i) \quad (30.4)$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

- Fenton [Fen91] suggests a number of simple morphology (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to the call-and-return architecture the following metrics can be defined:
- Size =  $n + a$
- where  $n$  is the number of nodes and  $a$  is the number of arcs.
- Size =  $17 + 18 = 35$
- Depth = longest path from the root (top) node to a leaf node. For the architecture shown in Figure , depth = 4.
- Width = maximum number of nodes at any one level of the architecture.
- For the architecture shown in Figure, width = 6.



The arc-to-node ratio,  $r = a/n$ , measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture.

For the architecture shown in Figure ,  $r = 18/17 = 1.06$ .

- The U.S. Air Force Systems Command [USA87] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program.
- Using concepts similar to those proposed in IEEE Std. 982.1-2005 [IEE05], the Air Force uses information obtained from data and architectural design to derive a *design structure quality index* (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI [Cha89]:

- $S_1$ = total number of modules defined in the program architecture
- $S_2$ = number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of  $S_2$ )
- $S_3$ = number of modules whose correct function depends on prior processing
- $S_4$ = number of database items (includes data objects and all attributes that define objects)
- $S_5$ = total number of unique database items
- $S_6$ = number of database segments (different records or individual objects)
- $S_7$ = number of modules with a single entry and exit (exception processing is not considered to be a multiple exit)

- Once values  $S_1$  through  $S_7$  are determined for a computer program, the following intermediate values can be computed:
- *Program structure*:  $D_1$ , where  $D_1$  is defined as follows:
  - If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then  $D_1 = 1$ , otherwise  $D_1 = 0$ .

*Module independence:*  $D_2 = 1 - \left( \frac{S_2}{S_1} \right)$

*Modules not dependent on prior processing:*  $D_3 = 1 - \left( \frac{S_3}{S_1} \right)$

*Database size:*  $D_4 = 1 - \left( \frac{S_4}{S_1} \right)$

*Database compartmentalization:*  $D_5 = 1 - \left( \frac{S_5}{S_4} \right)$

*Module entrance/exit characteristic:*  $D_6 = 1 - \left( \frac{S_6}{S_1} \right)$

With these intermediate values determined, the DSQI is computed in the following manner:

$$\text{DSQI} = \sum w_i D_i$$

- where  $i = 1$  to  $6$ ,  $w_i$  is the relative weighting of the importance of each of the intermediate values, and  $\sum w_i = 1$  (if all  $D_i$  are weighted equally, then  $w_i = 0.167$ ).
- The value of DSQI for past designs can be determined and compared to a design that is currently under development.
- If the DSQI is significantly lower than average, further design work and review are indicated.
- Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

- METRICS FOR SOURCE CODE
- Halstead Metrics:
- According to Halstead's "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operand.“
- Token Count
- In these metrics, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these basic symbols. These symbols are called as a token. The basic measures are
- $n_1$  = count of unique operators.
- $n_2$  = count of unique operands.
- $N_1$  = count of total occurrences of operators
- $N_2$  = count of total occurrence of operands.
- In terms of the total tokens used, the size of the program can be expressed as  
$$N = N_1 + N_2.$$

- Halstead metrics are:
- **Program Volume (V)**
- The unit of measurement of volume is the standard unit for size "bits." It is the actual size of a program if a uniform binary encoding for the vocabulary is used.
- $V=N \cdot \log_2 n$
- **Program Level (L)**
- The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size).
- $L = V/V^*$
- **Potential Minimum Volume**
- The potential minimum volume  $V^*$  is defined as the volume of the most short program in which a problem can be coded.
- $V^* = (2 + n2^*) \cdot \log_2 (2 + n2^*)$
- Here,  $n2^*$  is the count of unique input and output parameters

## Program Difficulty

The difficulty level or error-proneness (D) of the program is proportional to the number of the unique operator in the program.

$$D = (n_1/2) * (N_2/n_2)$$

## Programming Effort (E)

The unit of measurement of E is elementary mental discriminations.

$$E = V/L = D * V$$

## Estimated Program Length

- According to Halstead, The first Hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands.

- $N = N_1 + N_2$

And estimated program length is denoted by  $N^{\wedge}$

- $N^{\wedge} = n_1 \log_2 n_1 + n_2 \log_2 n_2$

# METRICS FOR MAINTENANCE

IEEE Std. 982.1-2005 [IEE05] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

$M_T$  = number of modules in the current release

$F_c$  = number of modules in the current release that have been changed

$F_a$  = number of modules in the current release that have been added

$F_d$  = number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = \frac{|M_T - (F_a + F_c + F_d)|}{M_T}$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

## Decision Table Technique

**Let's understand it by an example:**

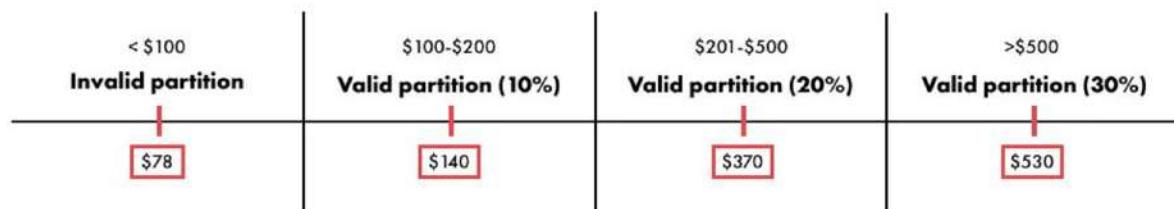
Most of us use an email account, and when you want to use an email account, for this you need to enter the email and its associated password.

If both email and password are correctly matched, the user will be directed to the email account's homepage; otherwise, it will come back to the login page with an error message specified with "Incorrect Email" or "Incorrect Password."

Now, let's see how a decision table is created for the login function in which we can log in by using email and password. Both the email and the password are the conditions, and the expected result is action.

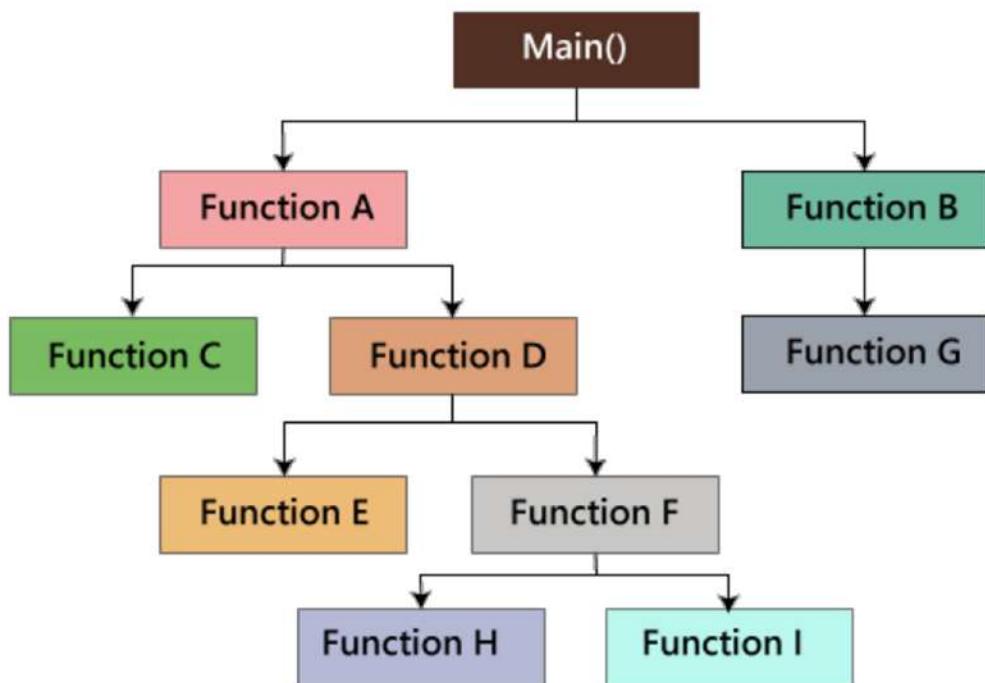
Email (condition1)	T	T	F	F
Password (condition2)	T	F	T	F
Expected Result (Action)	Account Page	Incorrect password	Incorrect email	Incorrect email

## Equivalence Partitioning



## White Box Testing

### 1. Path Testing



And test all the independent paths implies that suppose a path from main() to function G, first set the parameters and test if the program is correct in that particular path, and in the same way test all other paths and fix the bugs.

## 2. Loop Testing

Examples of types of loop tested are,

- Simple loop
- Nested loop
- Concatenated loop
- Unstructured loop

### Simple Loop

1. Skip the entire loop
2. Make 1 passes through the loop
3. Make 2 passes through the loop
4. Make  $a$  passes through the loop where  $a < b$ ,  $n$  is the maximum number of passes through the loop

5. Make  $b, b-1; b+1$  passes through the loop where “ $b$ ” is the maximum number of allowable passes through the loop.

#### Nested Loops

1. Set all the other loops to minimum value and start at the innermost loop
2. For the innermost loop, perform a simple loop test and hold the outer loops at their minimum iteration parameter value
3. Perform test for the next loop and work outward.
4. Continue until the outermost loop has been tested.

#### Concatenated Loops

In the concatenated loops, if two loops are independent of each other then they are tested using simple loops or else test them as nested loops.

#### Boundary Value Analysis(BVA)

Invalid partition	Valid partition (10%)	Valid partition (20%)	Valid partition (30%)
	\$99    \$100	\$200    \$201	\$500    \$501

#### Two Value Boundary analysis

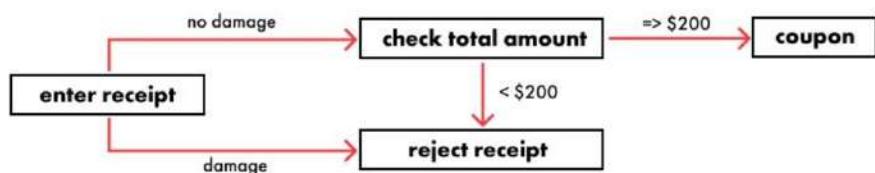
#### Three Value Boundary Analysis along with equivalence partitioning

Invalid partition	Valid partition (10%)	Valid partition (20%)	Valid partition (30%)
\$78	\$99    \$100	\$140    \$201	\$370    \$501

## Decision Table Testing

CONDITIONS	Rule 1	Rule 2	Rule 3	Rule 4
Total amount	Y	Y	N	N
Member	Y	N	Y	N
<b>ACTIONS/ OUTCOMES</b>				
Discount 10%	Y	N	N	N

## State Transition Testing



## White Box Testing

### Difference between statement/Branch/condition coverage

```
if(a || b)) {  
    test1 = true;  
}  
else {  
    if(c) {  
        test2 = true  
    }  
}
```

We have two statements here - if(a||b) and if(c), to fully explain those coverage differences:

1. **statement coverage** have to test each statement at least once, so we need just two tests:
  - a=true b=false - that gives us path if(a||b) true -> test1 = true
  - a=false, b=false and c=true - that gives us path: if(a||b) false -> else -> if(c) -> test2=true.

This way we executed each and every statement.

2. **branch/decision coverage** needs one more test:
  - a=false, b=false, c=false - that leads us to that second if but we are executing false branch from that statement, that wasn't executed in statement coverage

That way we have all the branches tested, meaning we went through all the paths.

3. **condition coverage** needs another test:
  - a=false, b=true - that leads through the same path as first test but executes the other decision in OR statement (a||b) to go through it.

#### 1. Path testing

Consider the following program, which determines whether an integer is prime or not. The following is the schedule for the following program –

- Create a Control Flow Diagram.
- Calculate the cyclic complexity using all of the methods available.

- Create a list of all the Independent Paths Design test cases.

```
int main () {
    int n, index;
    cout << "Enter a number: " << endl;
    cin >> n;
    index = 2;
    while (index <= n - 1) {
        if (n % index == 0) {
            cout << "It is not a prime number" << endl;
            break;
        }
        index++;
    }
    if (index == n)
        cout << "It is a prime number" << endl;
} // end main
```

### Solution: 1. Create a Control Flow Diagram

**Step 1** – After declaring the variables, begin numbering the statements (if no variables have been initialized in that statement). If a variable is initialized and declared on the same line, however, the numbering should begin on that line.

This is how numbering will be done for the supplied program –

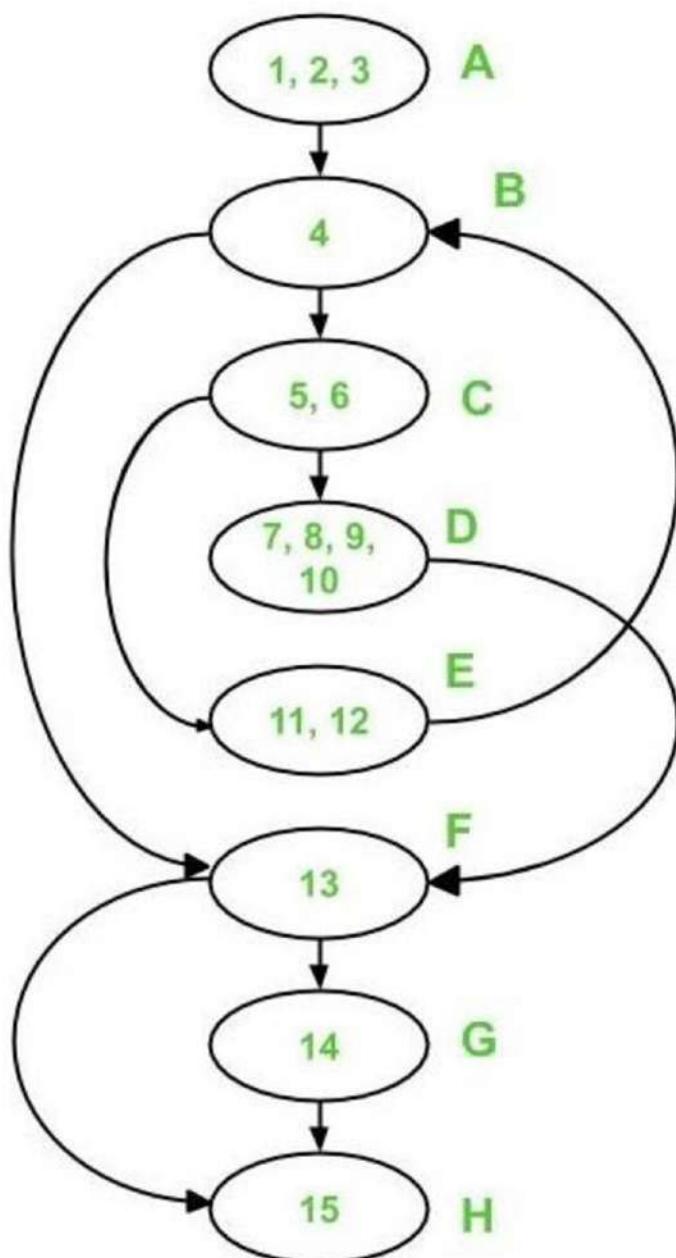
```
int main () {
    int n, index;
    cout << "Enter a number: " <> n;
    index = 2;
    while (index <= n - 1){
        if (n % index == 0){
            cout << "It is not a prime number" << endl;
            break;
        }
        index++;
    }
```

```
    }  
    if (index == n)  
        cout << "It is a prime number" << endl;  
} // end main
```

**Step 2** – Combine all of the sequential statements into one node. Statements 1, 2, and 3 are, for example, all consecutive statements that should be concatenated into a single node. For the rest of the statements, we'll use the notations described here.

**Note** – To keep things simple, number nodes in alphabetical order.

The resulting graph will look like this –



Calculate the cyclomatic complexity by using the following formula –

**Method 1:**

$$V(G) = e - n + 2*p$$

In the control flow diagram above,

where  $e = 10$ ,  $n = 8$ , and  $p = 1$

As a result,  $V(G) = 10 - 8 + 2 * 1 = 4$  Cyclomatic Complexity

**Method 2:**

$$V(G) = d + p$$

In the control flow diagram above,

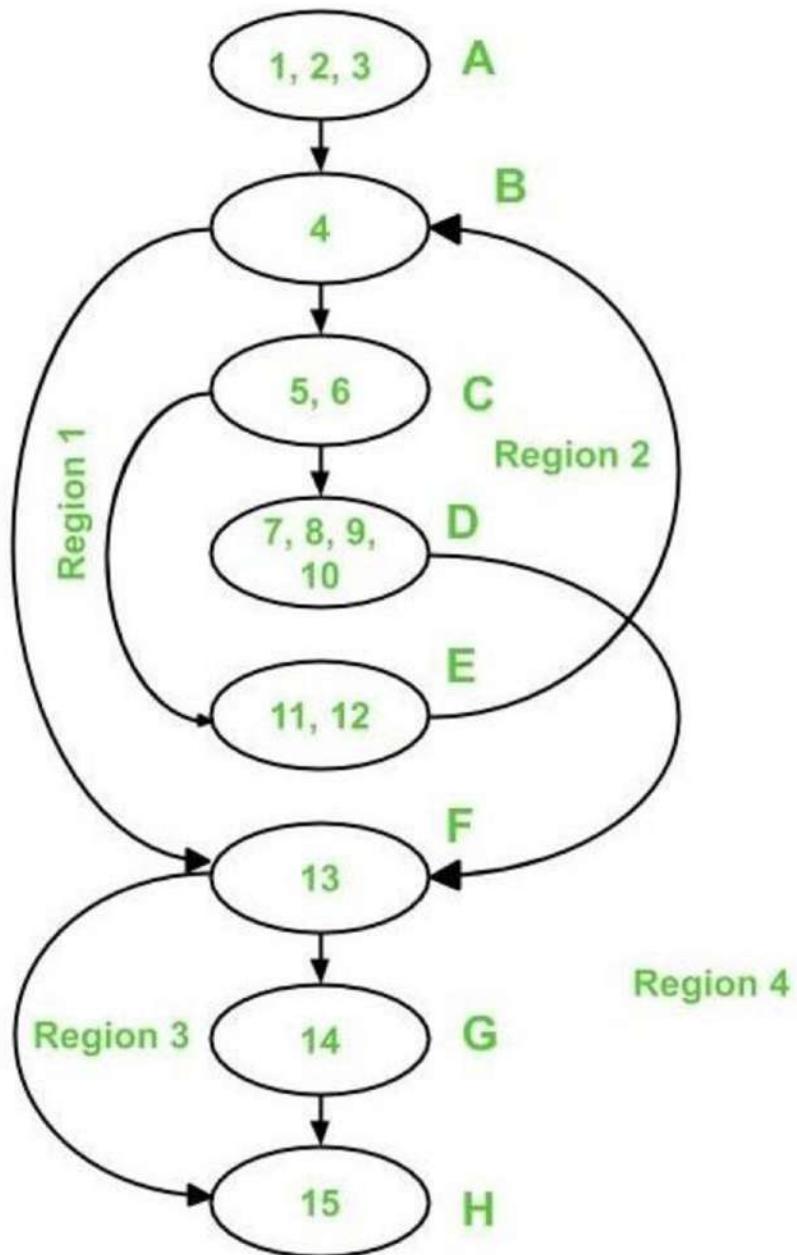
where  $d = 3$  (Node B, C, and F) and  $p = 1$

As a result, the cyclomatic complexity  $V(G) = 3 + 1 = 4$

**Method 3:**

$$V(G) = \text{Number of Regions (method 3)}$$

There are four zones in the aforementioned control flow graph, as indicated below –



As a result, there are four distinct regions: R1, R2, R3, and R4.

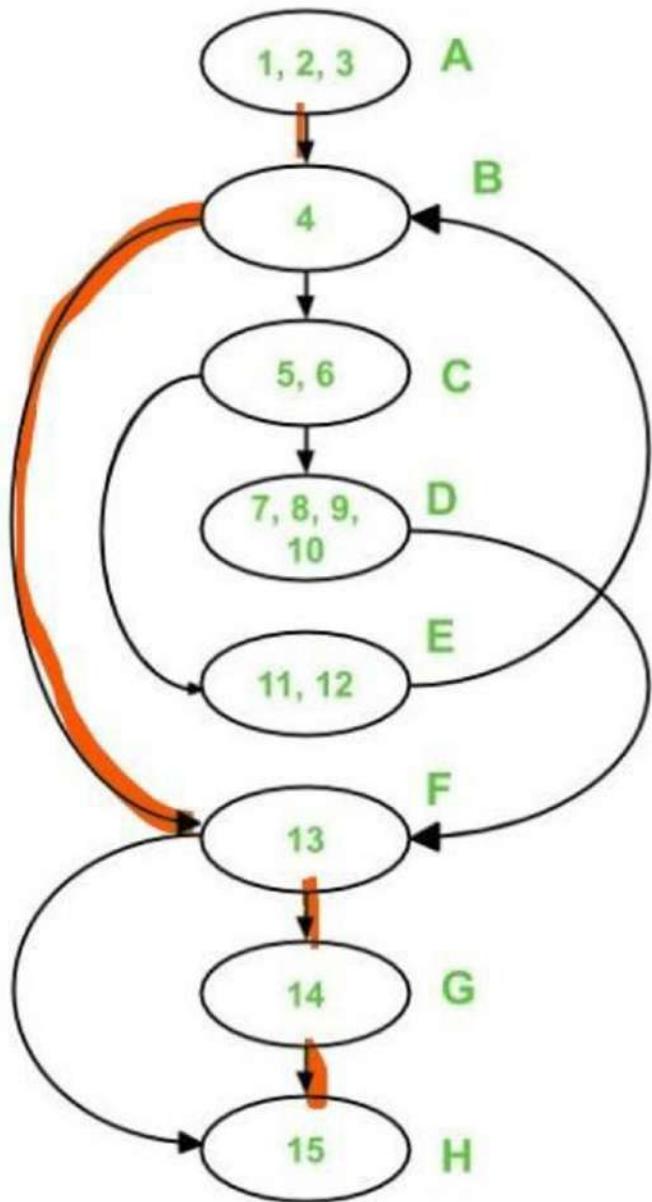
$$V(G) = 1 + 1 + 1 + 1 = 4 \text{ cyclomatic complexity}$$

It's worth noting that all three ways get the same cyclomatic complexity V value (G).

### Independent Paths

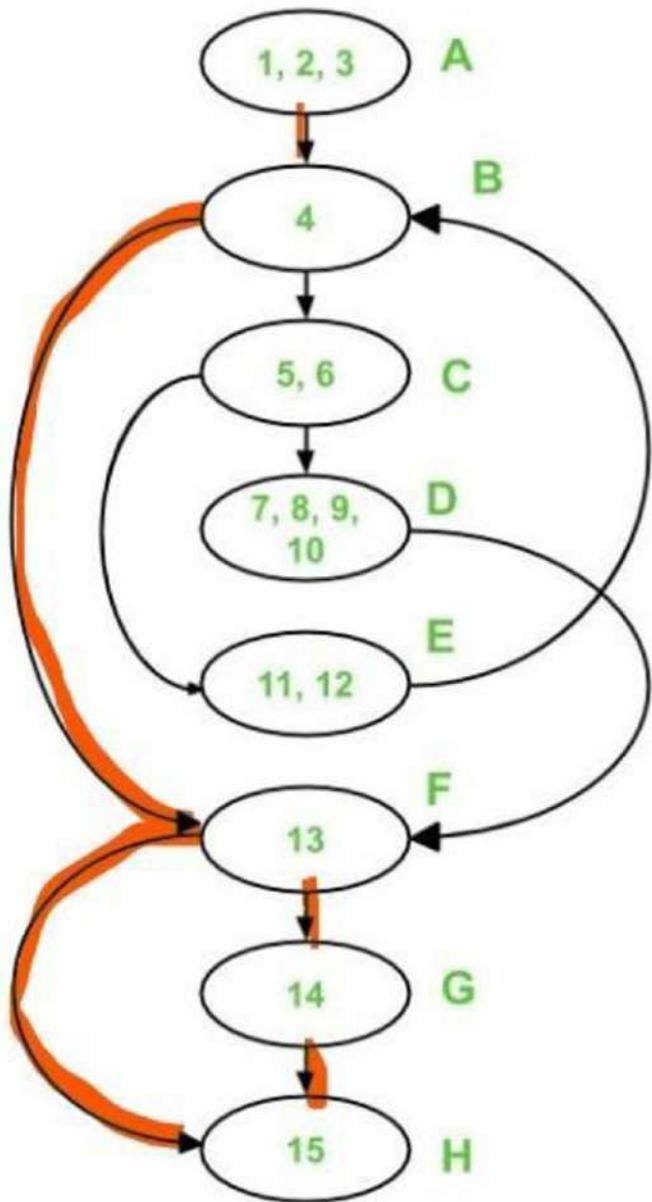
Because the cyclomatic complexity  $V(G)$  for the graph is 4, there are 4 independent paths.

Path 1 covers (in red) the following edges –



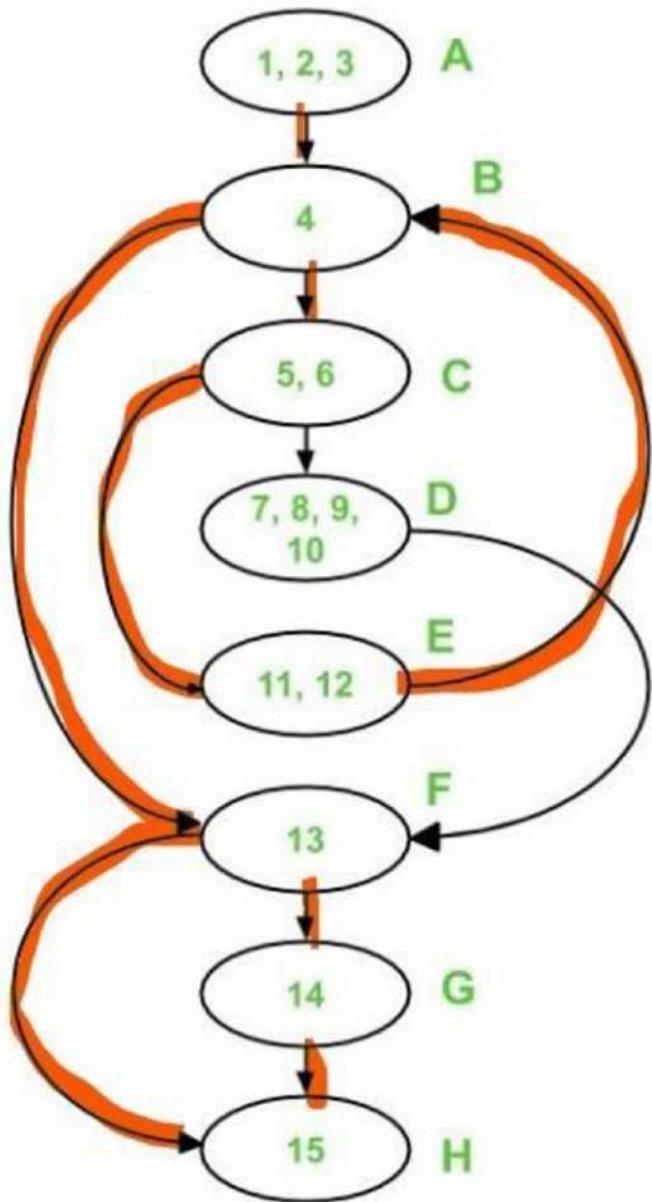
**A – B – F – G – H is the first path.**

The following are the edges that Path 1 and Path 2 cover –



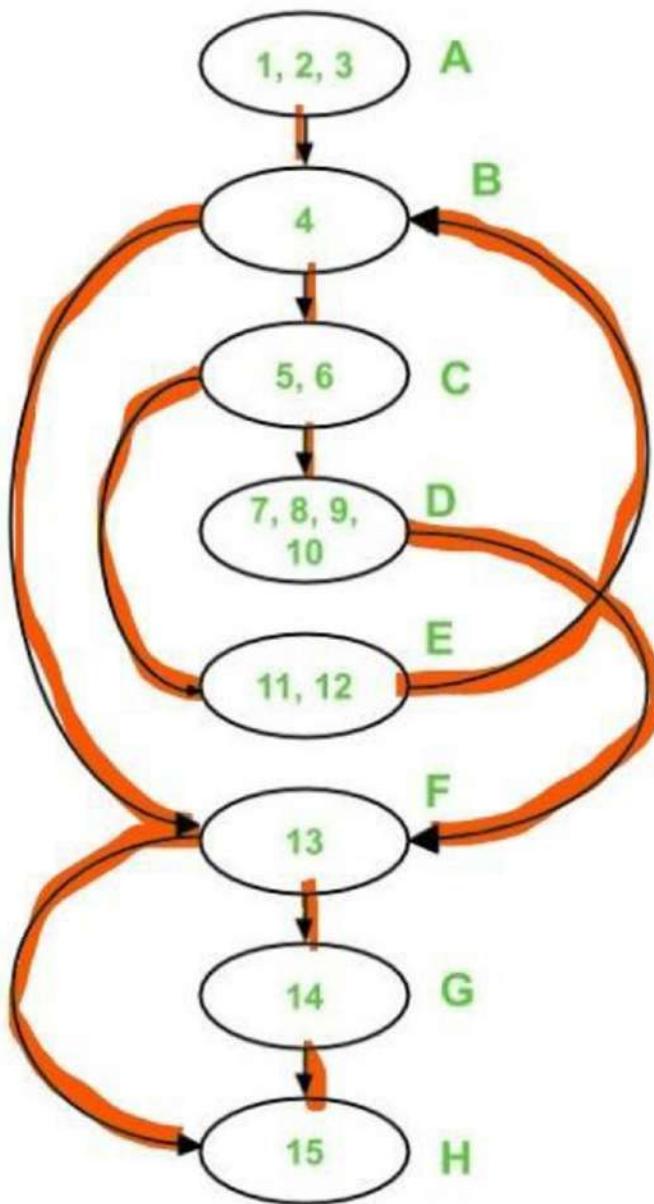
**A – B – F – H is the second path.**

Path 1, Path 2, and Path 3 cover the following edges –



**A - B - C - E - B - F - G - H is the third path**

Only two edges remain uncovered, namely edge C-D and edge D-F. As a result, these two edges must be included in Path 4.



**A - B - C - D - F - H is the fourth path.**

Each of these pathways has at least one new edge that has never been crossed previously.

Note that independent pathways aren't always unique.

### Test Cases

We must use the previously discovered independent paths to generate test cases. Provide input to the program in such a way that each independent path is executed to create a test case.

The following test cases will be obtained for the supplied program –

Test case ID	Input Number	Output	Independent Path covered
1	1	No output	A-B-F-H
2	2	It is a prime number	A-B-F-G-H
3	3	It is a prime number	A-B-C-E-B-F-G-H
4	4	It is not a prime number	

## Test Case Examples of Non-Functional Testing

Following are examples of Non-Functional Testing –

Test Case No.	Test Case	Domain
1	Up to 1000 users running the application at the same time, the application load time should not exceed 5 seconds.	Performance Testing
2	Software should be installable on all versions of Windows and Mac	Compatibility Testing
3	All web images should have alt tags	Accessibility testing

## 2. Data Flow Testing

See the code below:

```

1. read x;
2. If(x>0)           (1, (2, t), x), (1, (2, f), x)
3. a= x+1            (1, 3, x)
4. if (x<=0) {       (1, (4, t), x), (1, (4, f), x)
5.   if (x<1)        (1, (5, t), x), (1, (5, f), x)
6.     x=x+1; (go to 5) (1, 6, x)
else
7.   a=x+1            (1, 7, x)
8.   print a;          (6,(5, f)x), (6,(5,t)x)
                      (6, 6, x)

```

## Associations

(1, (2, f), x), (1, (2, t), x), (1, 3, x), (1, (4, t), x), (1, (4, f), x), (1, (5, t), x), (1, (5, f), x), (1, 6, x),  
 (1, 7, x), (6,(5, f)x), (6,(5,t)x), (6, 6, x), (3, 8, a), (7, 8, a), (3, 8, a), (7, 8, a)

## Definition

Definition of a variable is the occurrence of a variable when the value is bound to the variable. In the above code, the value gets bound in the first statement and then start to flow.

- If( $x > 0$ ) is statement 2 in which value of  $x$  is bound with it.  
 Association of statement 2 is (1, (2, f), x), (1, (2, t), x)
- $a = x + 1$  is statement 3 bounded with the value of  $x$   
 Association of statement 3 is (1, 3, x)

## All definitions coverage

(1, (2, f), x), (6, (5, f) x), (3, 8, a), (7, 8, a).

## Predicate use (p-use)

If the value of a variable is used to decide an execution path is considered as predicate use (p-use). In control flow statements there are two

Statement 4 if ( $x \leq 0$ ) is predicate use because it can be predicate as true or false. If it is true then if ( $x < 1$ ),  $6x = x + 1$ ; execution path will be executed otherwise, else path will be executed.

## Computation use (c-use)

If the value of a variable is used to compute a value for output or for defining another variable.

<b>Statement 3</b>	$a = x + 1$	(1,	3,	x)
<b>Statement 7</b>	$a = x + 1$	(1,	7,	x)
<b>Statement 8</b>	print a	(3, 8, a), (7, 8, a).		

These are **Computation use** because the value of x is used to compute and value of a is used for output.

### 3. Control Flow Testing

**Control Flow Graph** is formed from the node, edge, decision node, junction node to specify all possible execution path.

## Notations used for Control Flow Graph

1. Node
2. Edge
3. Decision Node
4. Junction node

### Node

Nodes in the control flow graph are used to create a path of procedures. Basically, it represents the sequence of procedures which procedure is next to come so, the tester can determine the sequence of occurrence of procedures.

We can see below in example the first node represent the start procedure and the next procedure is to assign the value of n after assigning the value there is decision node to decide next node of procedure as per the value of n if it is 18 or more than 18 so Eligible procedure will execute otherwise if it is less than 18 Not Eligible procedure executes. The next node is the junction node, and the last node is stop node to stop the procedure.

## Edge

Edge in control flow graph is used to link the direction of nodes.

We can see below in example all arrows are used to link the nodes in an appropriate direction.

## Decision node

Decision node in the control flow graph is used to decide next node of procedure as per the value.

We can see below in example decision node decide next node of procedure as per the value of n if it is 18 or more than 18 so Eligible procedure will execute otherwise if it is less than 18, Not Eligible procedure executes.

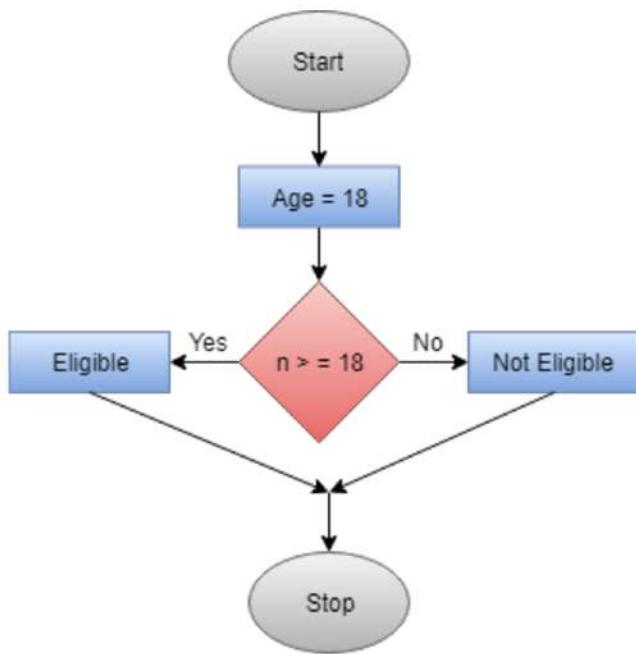
## Junction node

Junction node in control flow graph is the point where at least three links meet.

## Example

```
1. public class VoteEligibilityAge{  
2.  
3.     public static void main(String []args){  
4.         int n=45;  
5.         if(n>=18)  
6.         {  
7.             System.out.println("You are eligible for voting");  
8.         } else  
9.         {  
10.            System.out.println("You are not eligible for voting");  
11.        }  
12.    }  
13.}
```

## Diagram - control flow graph



The above example shows eligibility criteria of age for voting where if age is 18 or more than 18 so print message "You are eligible for voting" if it is less than 18 then print "You are not eligible for voting."

Program for this scenario is written above, and the control flow graph is designed for the testing purpose.

# Classification of Software Metrics

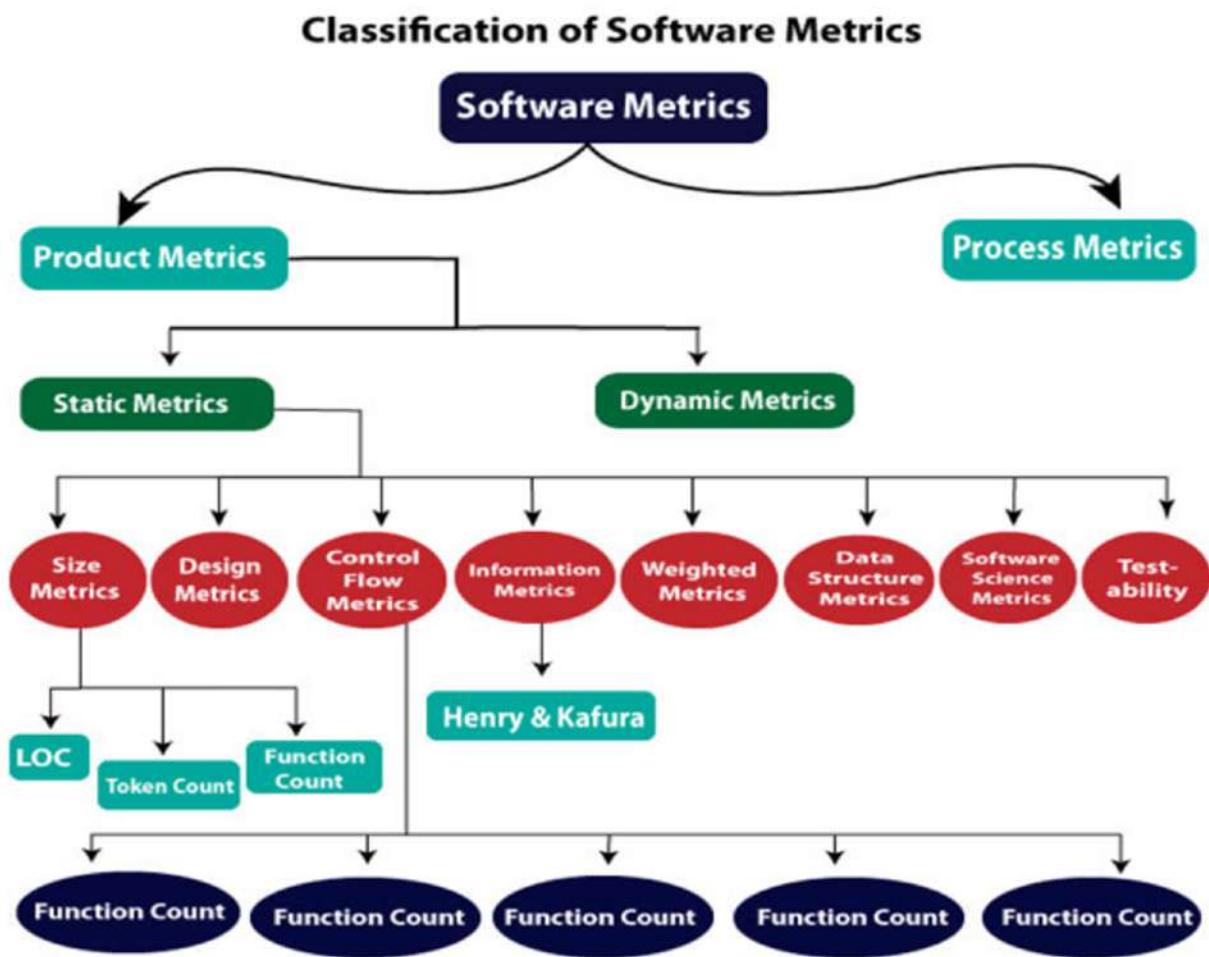
**Software metrics can be classified into two types as follows:**

**1. Product Metrics:** These are the measures of various characteristics of the software product. The two important software characteristics are:

1. Size and complexity of software.
2. Quality and reliability of software.

These metrics can be computed for different stages of SDLC.

**2. Process Metrics:** These are the measures of various characteristics of the software development process. For example, the efficiency of fault detection. They are used to measure the characteristics of methods, techniques, and tools that are used for developing software.



## Halstead's Software Metrics

According to Halstead's "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operand."

### a) Token Count

In these metrics, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these basic symbols. These symbols are called as a token.

The basic measures are

$n_1 = \text{count of unique operators.}$   
 $n_2 = \text{count of unique operands.}$   
 $N_1 = \text{count of total occurrences of operators.}$   
 $N_2 = \text{count of total occurrence of operands.}$

In terms of the total tokens used, the size of the program can be expressed as  $N = N_1 + N_2$ .

b) <https://www.youtube.com/watch?v=NtWmO9Cgii8>

## 2. Metrics for Requirement Model

### Objectives of FPA

The basic and primary purpose of the functional point analysis is to measure and provide the software application functional size to the client, customer, and the stakeholder on their request. Further, it is used to measure the software project development along with its maintenance, consistently throughout the project irrespective of the tools and the technologies.

**Following are the points regarding FPs**

1. FPs of an application is found out by counting the number and types of functions used in the applications. Various functions used in an application can be put under five types, as shown in Table:

**Types of FP Attributes**

Measurements Parameters	Examples
1. Number of External Inputs (EI)	Input screen and tables
2. Number of External Output (EO)	Output screens and reports
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories
5. Number of external interfaces (EIF)	Shared databases and shared routines.

All these parameters are then individually assessed for complexity.

2. FP characterizes the complexity of the software system and hence can be used to depict the project time and the manpower requirement.

3. The effort required to develop the project depends on what the software does.
4. FP is programming language independent.
5. FP method is used for data processing systems, business systems like information systems.
6. The five parameters mentioned above are also known as information domain characteristics.
7. All the parameters mentioned above are assigned some weights that have been experimentally determined and are shown in Table

### **Weights of 5-FP Attributes**

<b>Measurement Parameter</b>	<b>Low</b>	<b>Average</b>	<b>High</b>
1. Number of external inputs (EI)	7	10	15
2. Number of external outputs (EO)	5	7	10
3. Number of external inquiries (EQ)	3	4	6
4. Number of internal files (ILF)	4	5	7
5. Number of external interfaces (EIF)	3	4	6

Example <https://www.youtube.com/watch?v=CXgeSLbx7RA>

#### **3. Metrics for Design**

<https://www.youtube.com/watch?v=2iwQ-ZVN7gg>  
<https://slideplayer.com/slide/4889662/>

#### **4. Metrics for Maintenance**

<https://www.youtube.com/watch?v=09TFJEaSc74>