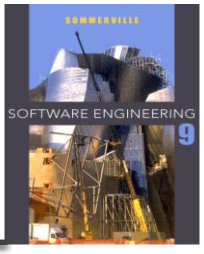
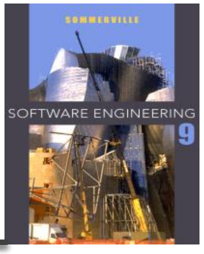


Midterm Prep Fall 2013

2 kinds of software products



Generic Products

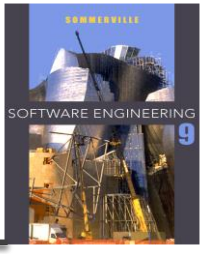
Organization that develops software controls specification

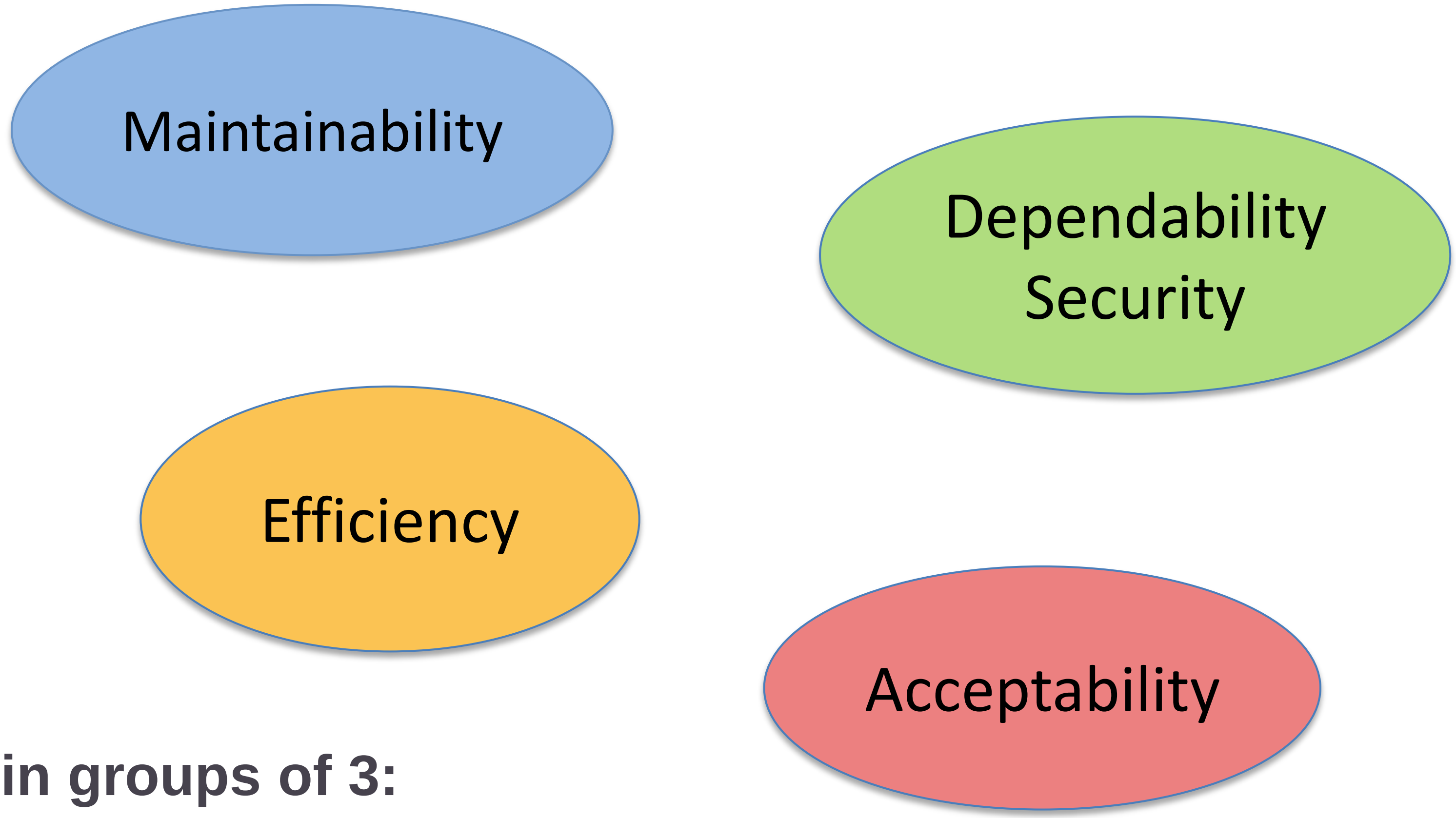
Customized Products

Customer provides specification

With software reuse this distinction gets blurred

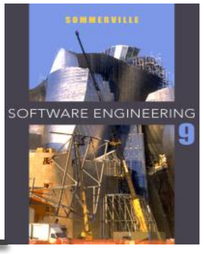
Essential attributes of good software Fig.1.2





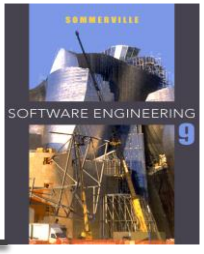
in groups of 3:
brain-storm examples

Software engineering

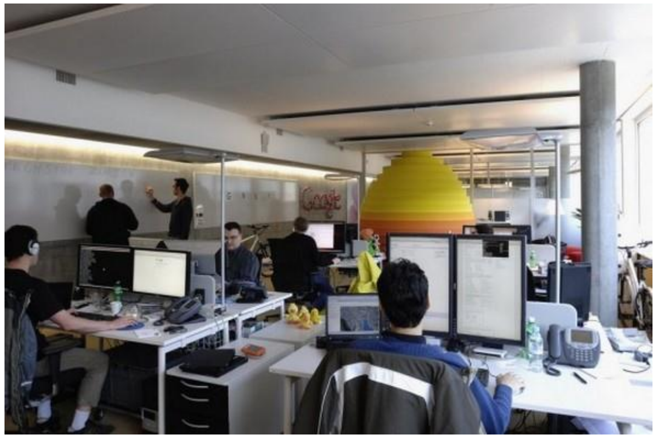


- ✧ Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- ✧ Engineering discipline
  - Select most appropriate way for given circumstance - pragmatic
- ✧ All aspects of software production
  - Incl. project management, documentation, software configuration …

Software engineering



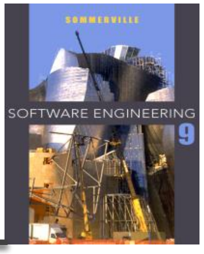
✧ Software engineering is intended to support professional development rather than individual programming



professional programming

individual programming

Software engineering

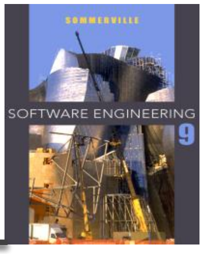


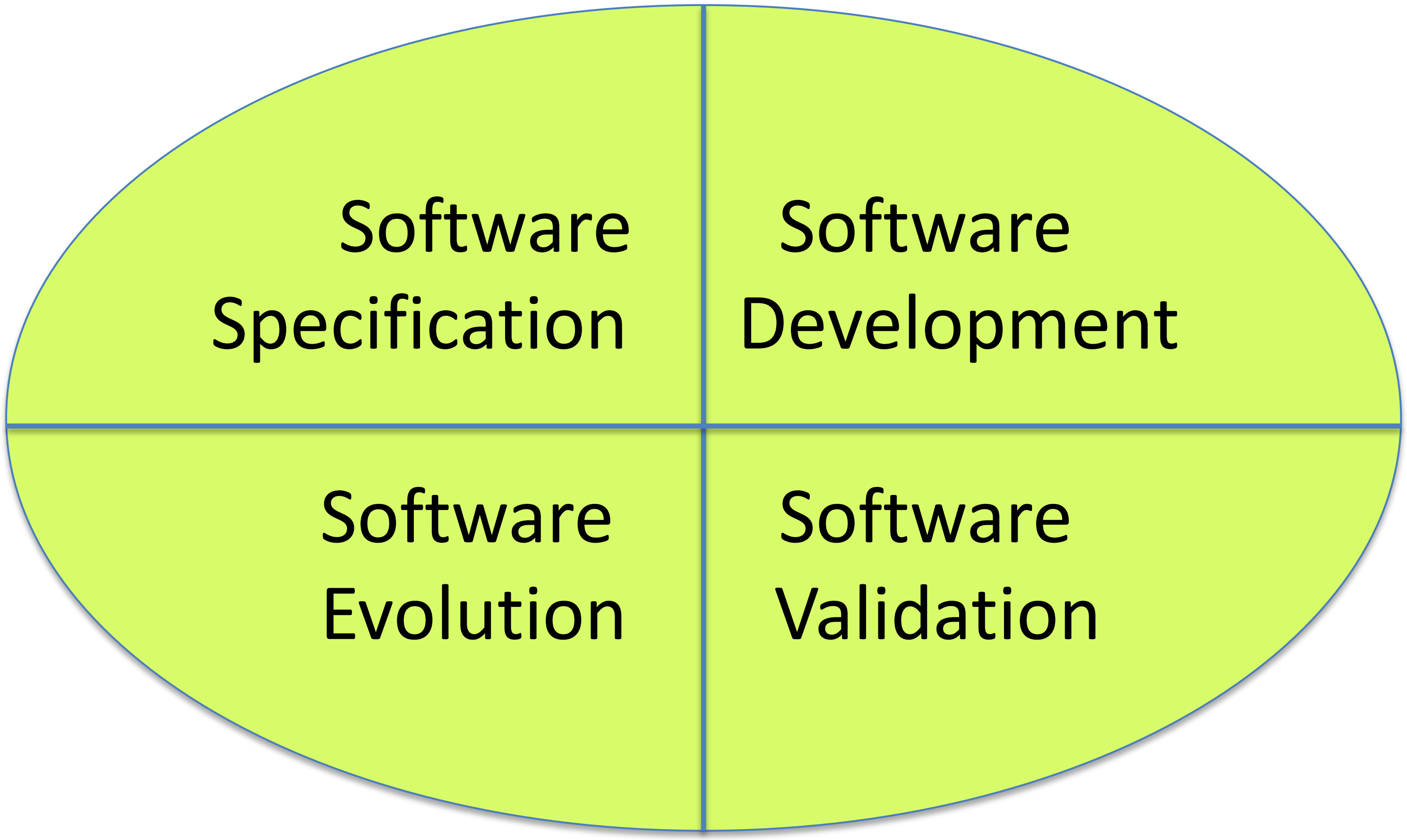
✧ .. About getting required results of required quality within schedule and budget

✧ A systematic, organized approach is often most effective to produce high-quality software.

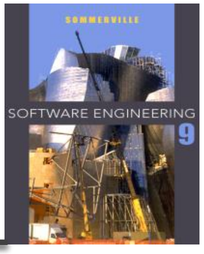
✧ Less formal development used where appropriate (e.g. web development)

Software process activities





Software development processes



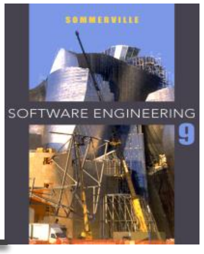
✧ Different types of systems need different development processes

✧ e.g.

real–time system in aircraft	vs	e–commerce software
completely specified		specification and program
before development starts		developed together

✧ No one software engineering method or technique applies to all software

3 General issues that affect most software



Heterogeneity

Need for dependable software that can cope with heterogeneity

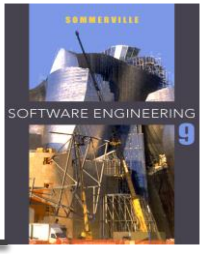
Business and Social Change

Need for rapid implementation and delivery

Security and Trust

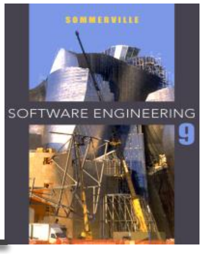
Need for secure and trustworthy software

Changes brought by the web



Before the web	Now
Applications mostly run on single computers	Applications run on one or more web server(s)
Typically pay for software	Software often free
Communication local	Communication global
Limited software reuse	Extensive software reuse

Web software engineering

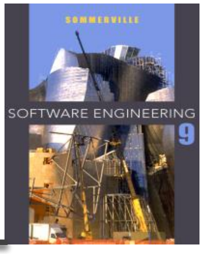


- ✧ Software reuse is the dominant approach for constructing web-based systems.
- ✧ Developed and Delivered incrementally
- ✧ User interfaces are constrained by the capabilities of web browsers.

Issues of professional responsibility

- Confidentiality
- Competence
- Intellectual Property Rights
- Computer Misuse

There is no 'ideal' software process



✧ Which process should we choose?

Consider type of application:

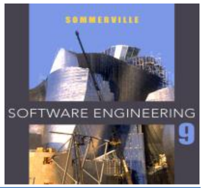
- Critical system vs business system

Consider organization:

- Process standardization

Consider developer team:

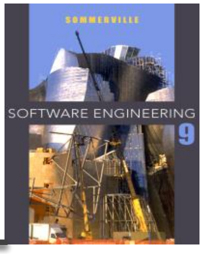
2 types of processes



Plan-driven process	Agile Process
Process activities are planned in advance	Planning is incremental
Progress is measured against this plan.	Easier to change
Process concludes with the delivery (except maintenance)	Delivery is incremental

In practice, most practical processes include elements of both plan-driven and agile approaches.

2.1 Software process models



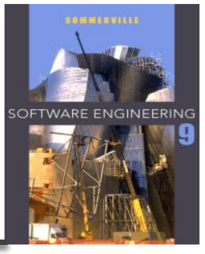
✧ "Process Paradigms":

a) The waterfall model
   - Plan-driven and document-driven

b) Incremental development
   - Specification, development, validation interleaved
   - Plan-driven or agile

c) Reuse-oriented software engineering
   - plan-driven or agile.

Waterfall model



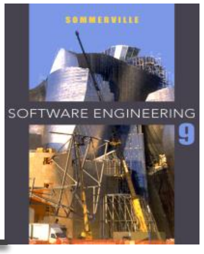
- Complete a phase > produce document(s) > next phase.
- Pro:

  Managers can monitor process

  Structure and documents help coordinate work
- Con:

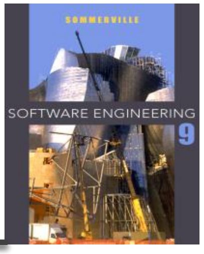
  Inflexible, cost of change is high
- Mostly used for
  - Large distributed projects
  - Safety and security critical systems / many regulations

Incremental development



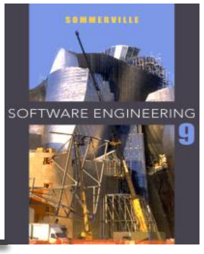
- Pro:
  - More flexible, cost of change reduced
  - Easier to get customer feedback
  - More rapid delivery and deployment of useful software
- Con:
  - Process not visible to managers
  - System structure tends to degrade (refactoring needed)
- Used for business, e-commerce, web applications, ..

Pro / Con of incremental delivery



- Pro:
  - Customer gains value early on
  - Early increments help elicit requirements
  - Highest priority components tend to get most testing
  - Less likely to fail, easier to adapt to change
- Con:
  - Harder to identify common facilities used by all increments
  - Replacement systems => difficulties with users (want all)
  - No complete requirements up front => difficulties w. managers

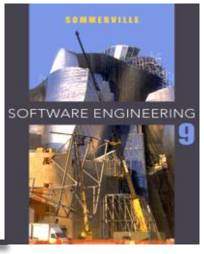
Pro / Con of Software Reuse



- Pro:
  - Faster development at lower cost
  - Components are already tested
- Con:
  - Requirements compromises are inevitable
  - Control over system evolution reduced (new versions)

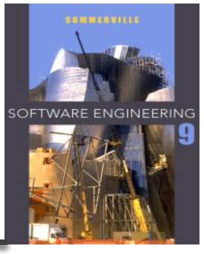
- Reuse is now the standard approach for building many types of business system

23.2 Plan-driven development



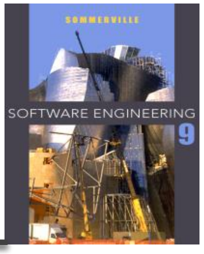
- ✧ Development process planned in detail.
- ✧ ‘Traditional’ way of managing large software projects.
- ✧ Project plan records
  - **What** work needs to be done
  - **Who** will do it
  - **When** should it be done

Plan-driven development – pros and cons



- Pro:
  - organizational issues (availability of staff, other projects, etc.) can be closely taken into account
  - potential problems and dependencies can be discovered before the project starts
  - Managers can use plan to support project decision making and to measure progress
- Con:
  - Early decisions might have to be revised
    => rework

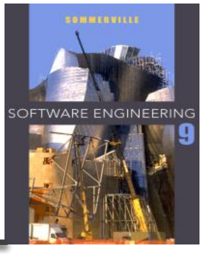
Project plan



- ✧ Principal project plan should focus on
  - Risks
  - Schedule
- ✧ There may be supplementary plans

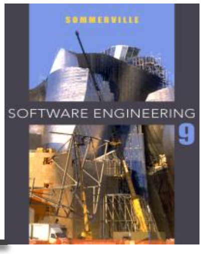
  Quality plan, Configuration management plan, Staff development plan, …

23.2.2 The planning process



- ✧ Iterative process
- ✧ Starts with initial project plan during the project startup phase
- ✧ More information becomes available during the project => regularly revise plan to reflect requirements, schedule, and risk changes

23.3 Project scheduling



The project schedule shows:

✧ When tasks will be executed

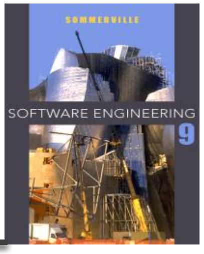
✧ Dependencies between tasks

✧ Estimated time required

✧ Allocation of people

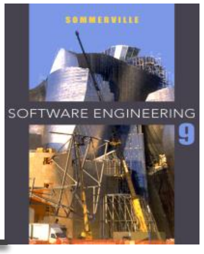
✧ Resources needed

Project schedule



- ✧ Iterative process
- ✧ Initial project schedule: during project startup
  - **Plan-driven** development:
    complete schedule developed up front
  - **Agile** development: (less detailed)
    identifies when major phases will be completed
    iterative approach to plan each phase
- ✧ Schedule is then refined and modified during development planning

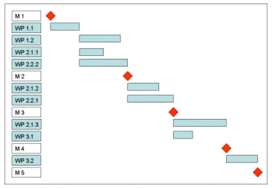
Schedule representation



✧ Table / Spreadsheet

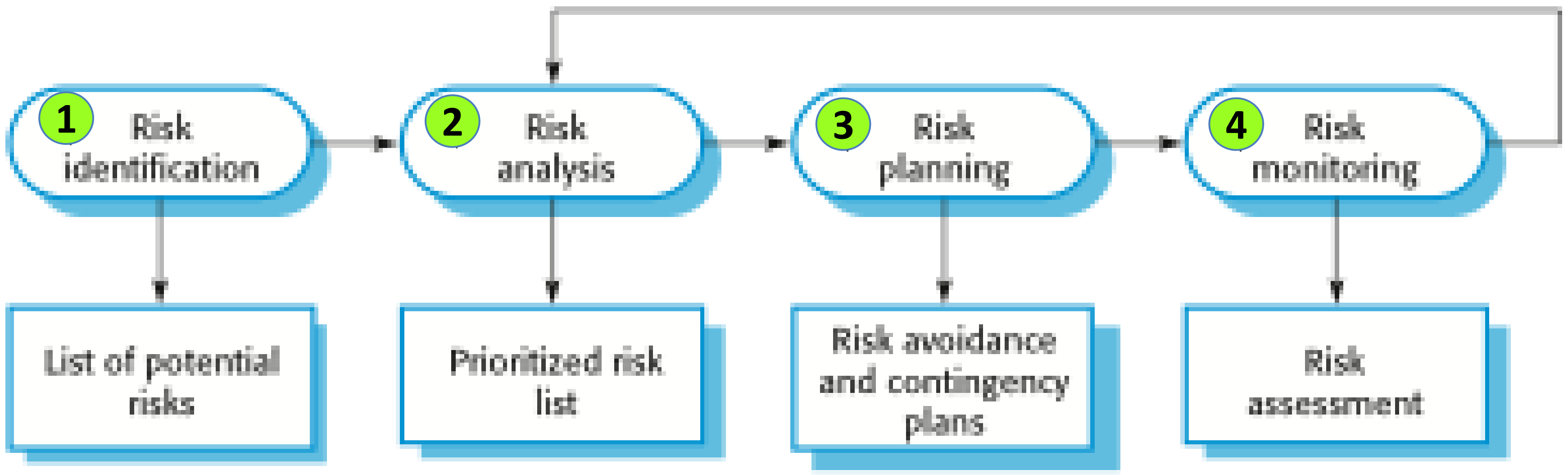
can be difficult to see relationships and dependencies

✧ Bar Chart (Gantt Chart)



- calendar based
- show who is responsible for the tasks
- begin and end of task / milestone
- Can show dependencies between tasks
- Can show progress accomplished

Risk Management Process



✧ Document outcome of risk management process in risk management plan

2 Risk Analysis

- ✧ Rank risks by probability and seriousness

Probability:

< 10 %	10 – 25 %	25 – 50 %	50 – 75 %	> 75 %

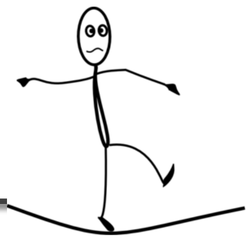
Seriousness:

Insignificant	Tolerable	Serious	Catastrophic

- ✧ Tabulate results ordered by seriousness

Update table during each iteration of the risk process

3 Risk Planning



For each of the key risks:

1. Collect information to anticipate problem
2. Find strategies to minimize project disruption

Risk management strategies:

a) Avoidance strategy

b) Minimization strategy

c) Contingency plan

People management factors

vs

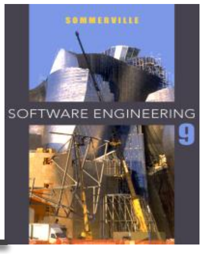
If you want to land together you have to make sure that everyone in on board when you take off.

22.3 Teamwork

- ✧ Group interaction is key determinant of group performance.
- ✧ Successful groups are more than individuals with right skills
- ✧ Characteristics of good group:
  1. Cohesive
  2. Team spirit

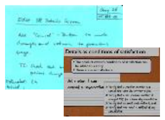
How to encourage group cohesiveness

- ✧ Establish sense of group identity
- ✧ Treat members as responsible and trustworthy
- ✧ Freely share information
- ✧ Organize social events


SOMMERVILLE
SOFTWARE ENGINEERING
9

change
software
Individuals
Working
processes
and
collaboration
contract
Customer
a
tools
negotiation
Responding
plan
following
interactions
documentation
comprehensive
to

Extreme programming practices (Fig 3.4)



Incremental planning

Continuous integration

Sustainable pace

days or weeks

Small releases

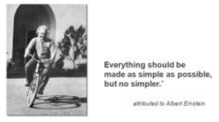
User Involve-ment

Test-first development

On-site customer

Refactoring

Simple design


Everything should be made as simple as possible, but no simpler."
attributed to Albert Einstein

Collective ownership

Pair programming

3.3.1 Testing in XP

✧ XP testing features:

- **Test-first** development.
- Incremental test development from scenarios.
- User involvement in test development and validation.
- Automated test harnesses used
- Acceptance testing (with user data) incremental

3.3.2 Pair programming

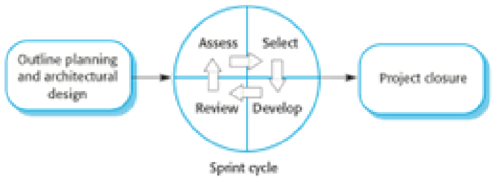
✧ programmers work in pairs, sitting side by side to develop code. (pairs created dynamically)

✧ collective ownership of code spreads knowledge across the team.

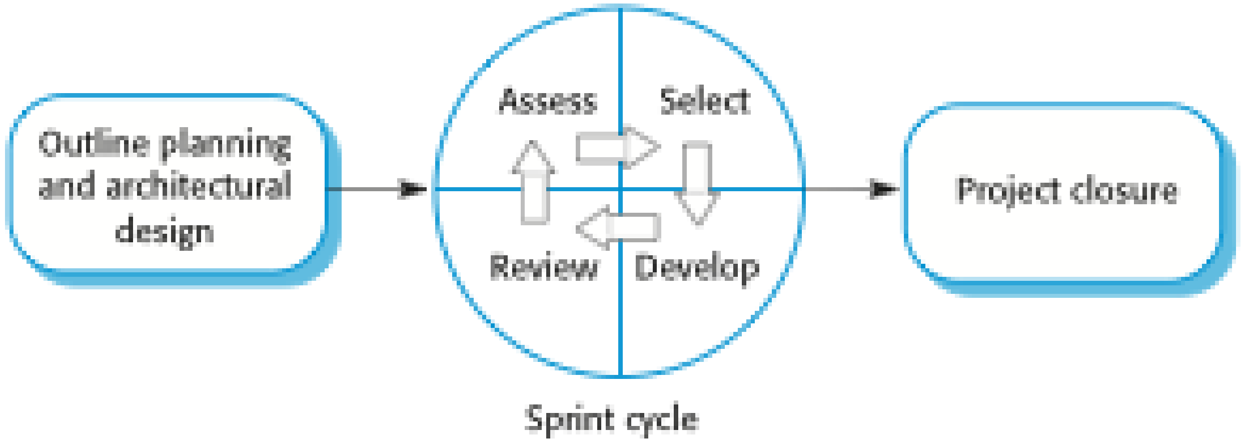
✧ informal code review

✧ encourages refactoring

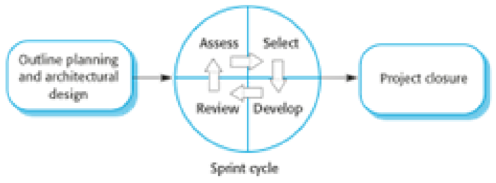
3.4 Scrum



- **Scrum** approach is a general agile method
- No prescribed programming practices like pair programming
- Provides management framework for iterative development
- Sprint .. Planning unit of fixed length; (few weeks)

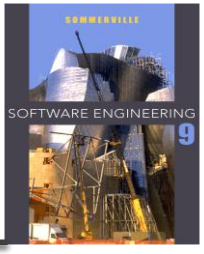


3.4 Sprint



- ✧ Assess: review product backlog, priorities, risks
- ✧ Select: features and functionality to be developed
- ✧ Develop: team organizes themselves
  short daily meeting
  Scrum master protects from external distractions
- ✧ Review: work reviewed and presented to stakeholders

3.5 Scaling agile methods



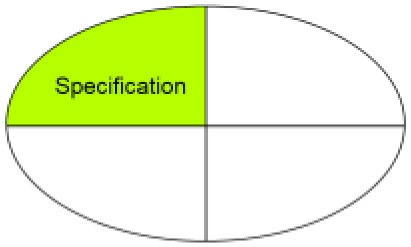
- **Scaling up** (to large software systems)

  Distributed / diverse stakeholders / multiple systems / configuration / rules regulations

- **Scaling out** (across large organization)

  reluctant to accept risk / company culture / diverse skills quality procedures / mandated tools

Requirement Documents



high-level, abstract . . . . detailed
statement specification

Dual function of requirements document:

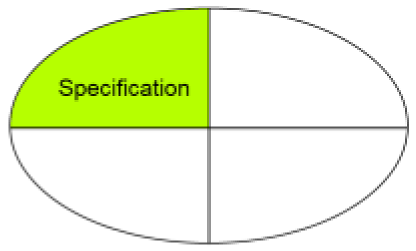
- basis for a bid for a contract

  => open to interpretation;

- basis for implementation of software

  => defined in detail;

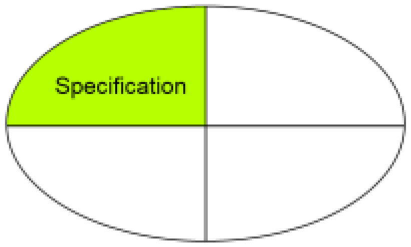
2 types of requirement



- ✧ User requirements
  - natural language + diagrams
  - Less detailed
  - e.g. For managers

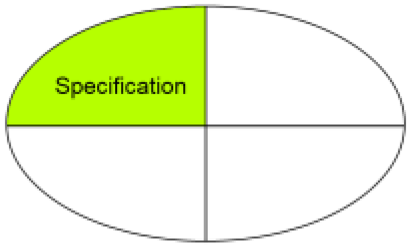
- ✧ System requirements
  - Structured, detailed document
  - e.g. For developers

4.1.1 Functional requirements



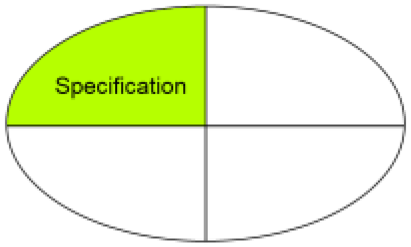
- Specific facilities provided by system
- Should be complete and consistent

4.1.2 Non-Functional requirements



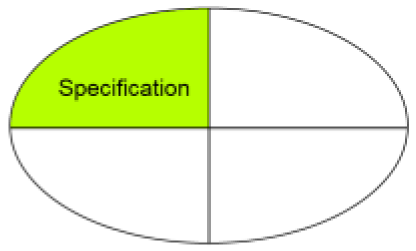
- Not directly related to specific facilities
- Usually affect system as a whole
  - Emergent system properties (e.g. Reliability, Performance)
  - Constraints on system (e.g. Regulations)

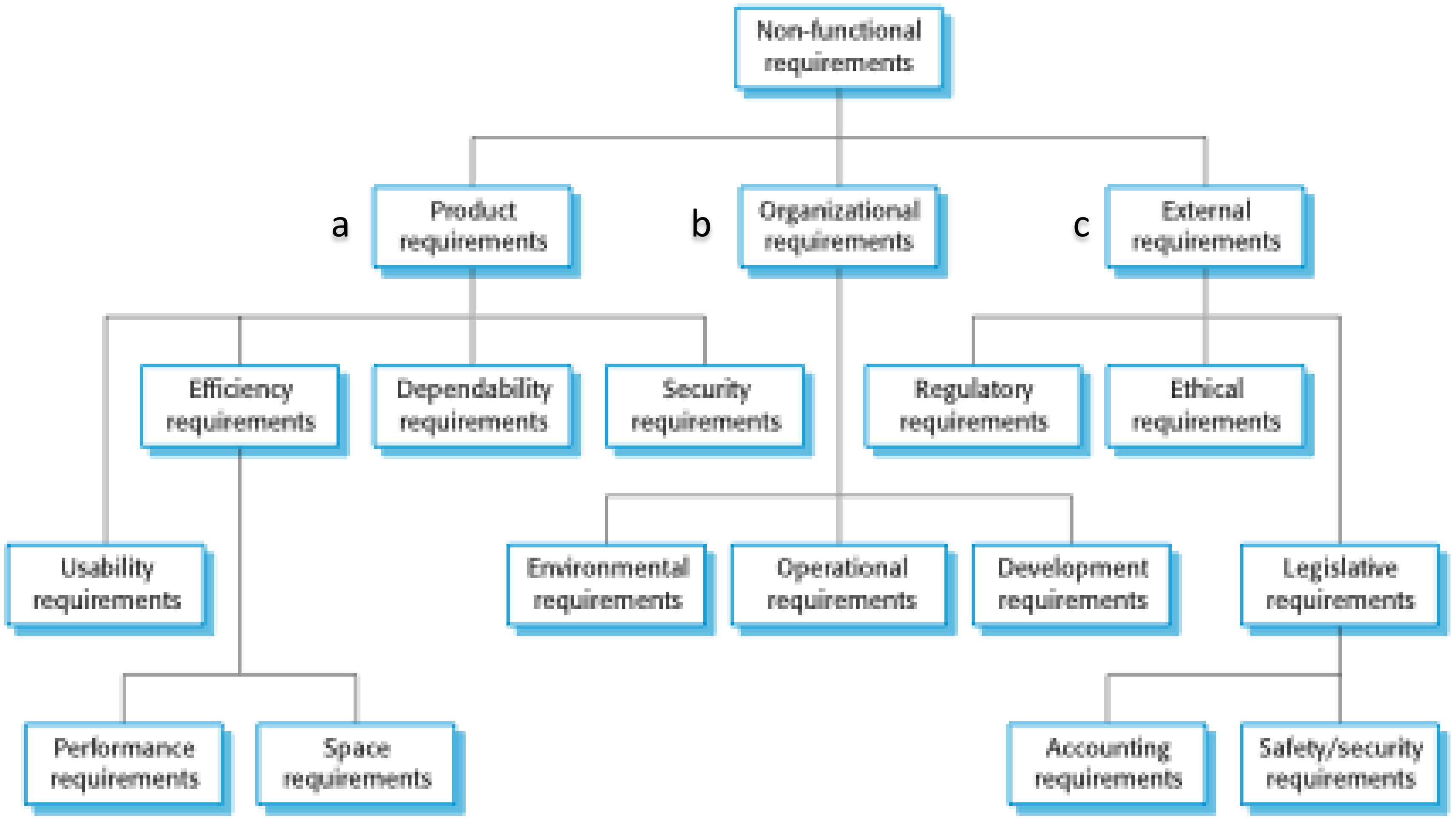
Non-Functional requirements



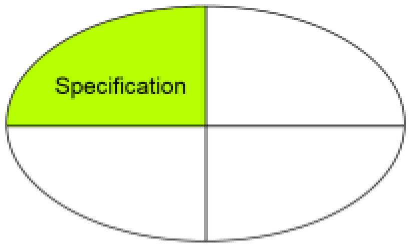
- May affect system architecture (e.g. Performance)
- A single non-functional requirement may generate multiple functional requirements (e.g. Security)
- Often more critical than functional requirements

Types of nonfunctional requirement



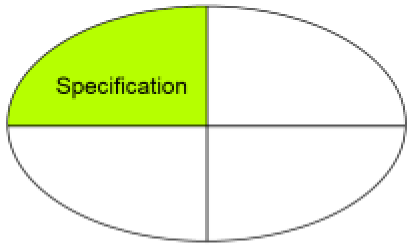


Domain requirements



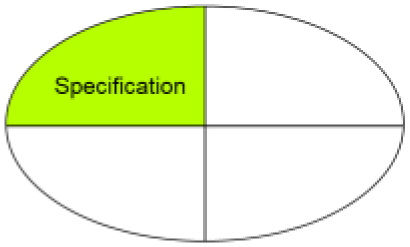
- ✧ Requirements imposed by system’s operational domain
  - For example, a train control system has to take into account the braking characteristics in different weather conditions.
- ✧ If domain requirements are not satisfied, system may be unworkable.

4.2 The software requirements document



- Official statement of what the system developers should implement.
- Should include
  - user requirements
  - system requirements
- NOT a design document
  - **WHAT** the system should do not **HOW** it should do it.

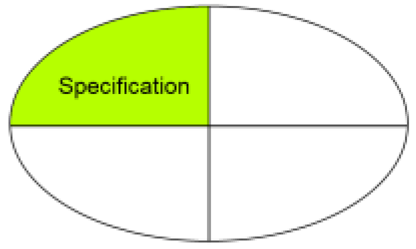
Requirements document



✧ Level of detail depends on

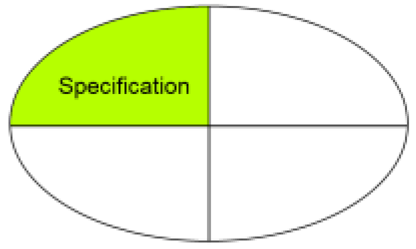
	Less detail	More detail
type of system	e.g. Games	e.g. Critical System
approach to development	Agile development	Traditional development
Location	local	distributed
Project Size	Small to medium	Large

4.3 Requirements specification



- The process of writing down the user and system requirements in a requirements document.
- User requirements have to be understood by managers, end-users and customers who have no technical background.
    - natural language
    - simple tables / forms
    - intuitive diagrams

4.3 Requirements specification

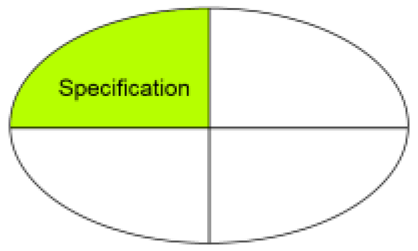


✧ System requirements:

- add detail and explain how user requirements should be provided by the system
- Should describe external behavior not design information

In practice, requirements and design can’t be completely separated.

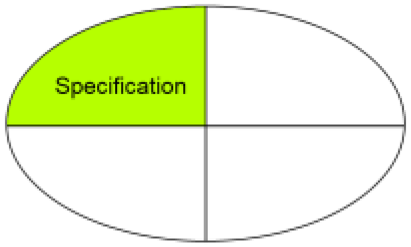
What not how …



✧ Some initial design is needed to

- to structure requirements specification (important if reuse of software component is planned)
- To interoperate with existing system
- Specific architecture to satisfy non-functional requirements

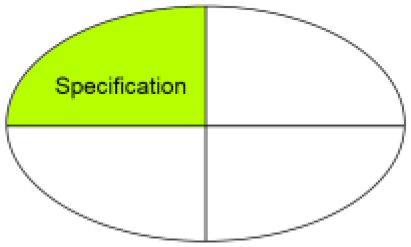
Guidelines for writing requirements



- ✧ Invent a standard format and use it for all requirements.
- ✧ Use language in a consistent way.

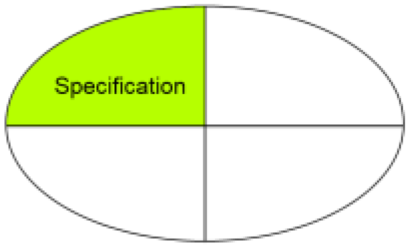
  e.g. shall for mandatory, should for desirable requirements
- ✧ Text highlighting to identify key parts
- ✧ Avoid jargon, acronyms, ..
- ✧ Include rationale why requirement is necessary.

4.4 Requirements engineering processes



- ✧ Vary widely depending on application domain, people involved and the organisation developing the requirements.
- ✧ Still, certain generic activities are common to all processes
  - Requirements feasibility
  - Requirements elicitation
  - Requirements analysis + specification
  - Requirements validation
  - Requirements management
- ✧ RE is an iterative activity in which these processes are interleaved.

1) Requirements discovery (elicitation)



✧ The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.

Sources:

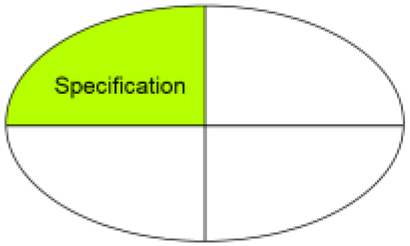
✧ Stakeholders

🟨 Interview 🟧 Scenario 🟧 Use Case 🟧 Ethnography

✧ Documentation, spec of similar systems

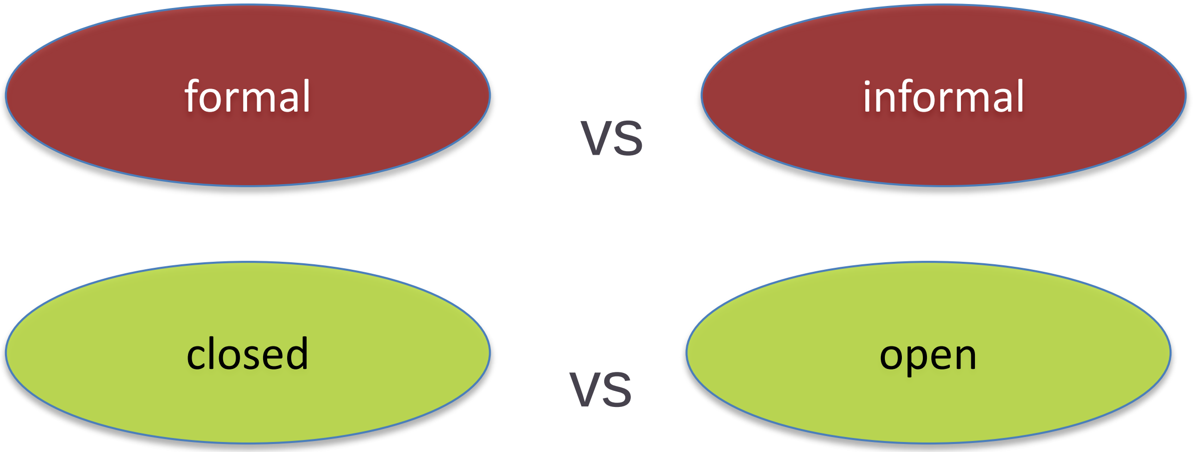
✧ Domain requirements, . .

Interviewing



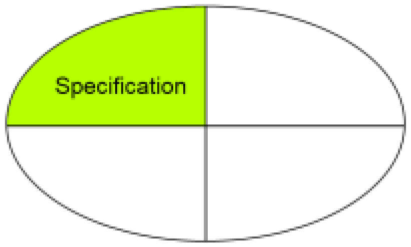
- Part of most RE processes

- Types of interview:



Typically a mix of closed and open-ended interviewing.

Interviewing



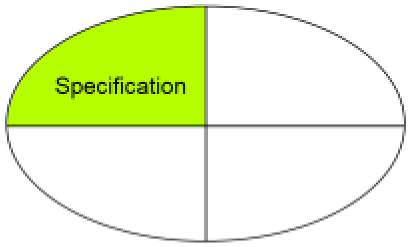
✧ Effective interviewing

- Be open-minded
- Avoid pre-conceived ideas
- Willing to listen

✧ Ways to get discussions going:

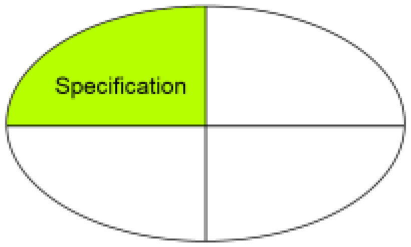
- springboard question
- requirements proposal
- working together on a prototype system.

Interviews in practice



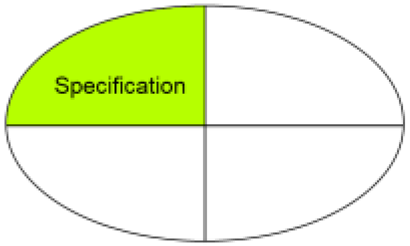
- Interviews are good for:
  - overall understanding what stakeholders do
  - how they might interact with system
  - identify difficulties with current system
- Interviews are not good for
  - understanding domain requirements
  - Organizational requirements

Scenarios

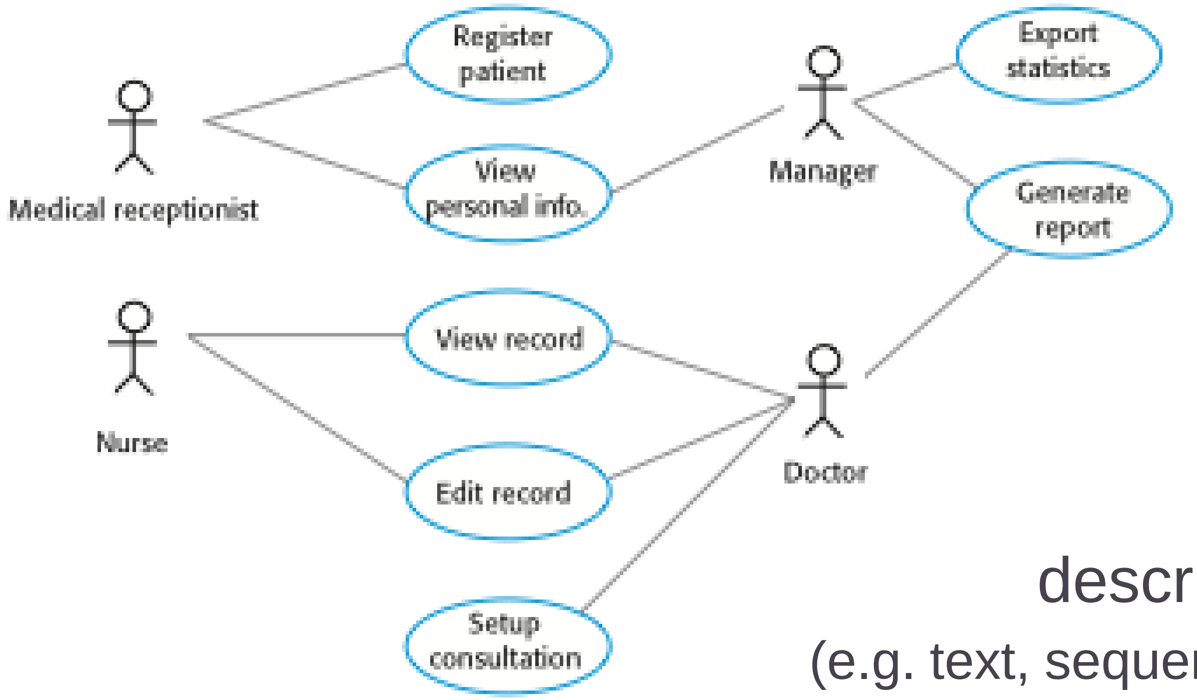


- Scenarios are real-life examples of how a system can be used.
- Pro: easy to relate to
- They should include
  - starting situation;
  - normal flow of events;
  - what can go wrong;
  - Information about other concurrent activities;
  - state when the scenario finishes.

Use cases



✧ Use case diagram (UML);

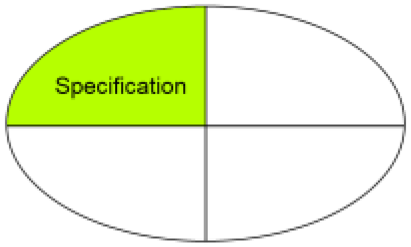


description of use-case

(e.g. text, sequence / state diagram..)

✧ Set of use cases should describe all possible interactions

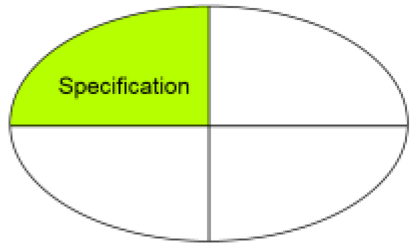
Scenarios and use cases



+ Effective for eliciting requirements

− Not as effective for eliciting constraints, high-level business requirements, non-functional requirements, or domain requirements

Ethnography

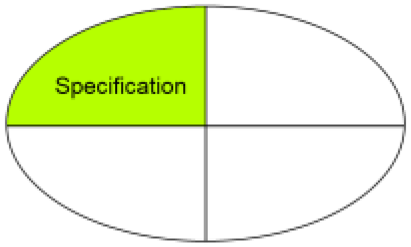


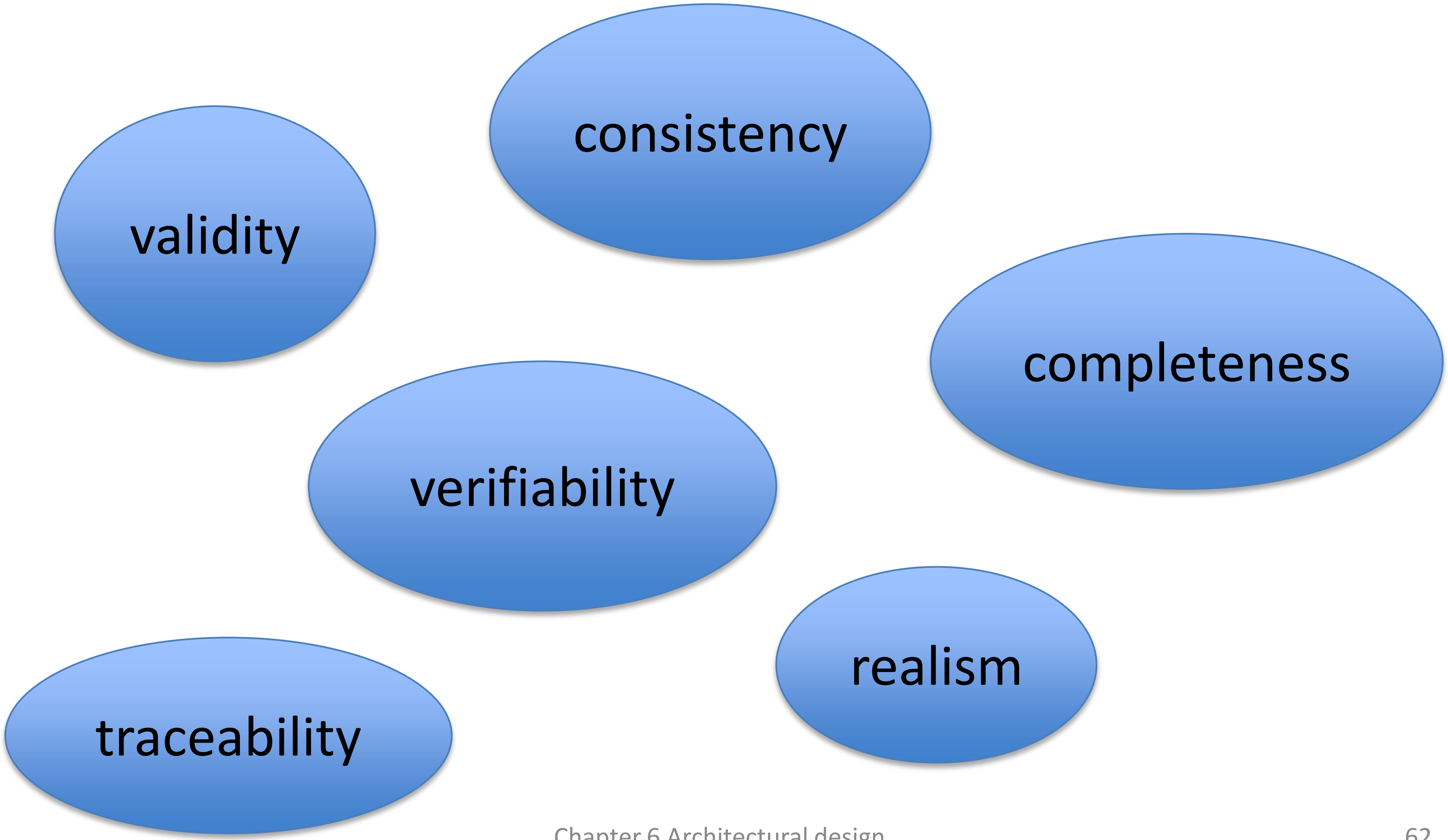
- Social scientist spends considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.

+ Effective in discovering requirements that are derived
  - from way in which people actually work (e.g. alarm)
  - From cooperation / awareness of other people's activities (e.g. screen)

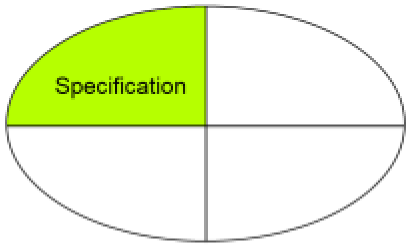
− Cannot identify new features
Might study existing but outdated practices

Requirements checks



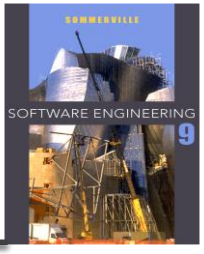


Requirements validation techniques



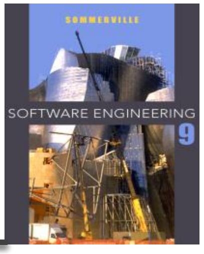
- ✧ Requirements reviews
- ✧ Prototyping
- ✧ Test-case generation

4.7 Requirements management



✧ Requirements management is the process of managing changing requirements during the requirements engineering process and system development.

Requirements change management



3 stages to a change management process:

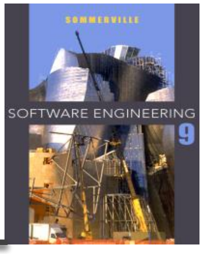
- *Problem analysis and change specification*
  - Check validity ; Analysis is fed back to change requestor
- *Change analysis and costing*
  - Use traceability information and general knowledge of the system requirements to analyze cost
    => decision is made whether or not to proceed
- Change implementation

System Model

A system model is an abstraction of the system

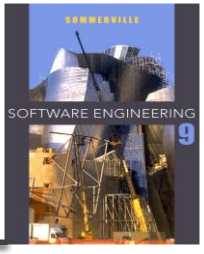
Alternative Representation	vs	**Abstraction:**
Should maintain all		Deliberately simplifies
information of system		Leaves out detail

Perspectives used for system modelling



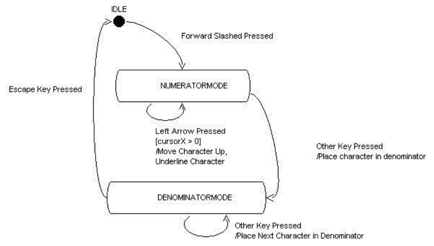
- External perspective
- Interaction perspective
- Structural perspective
- A behavioral perspective

Graphical Notation (UML)

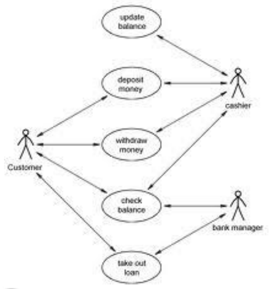


✧ 5 types of UML diagrams that can represent the essentials of a system:

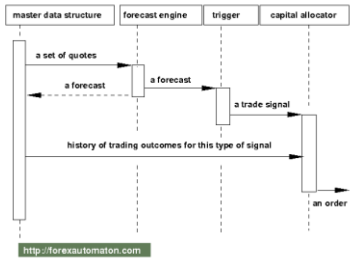
State diagram



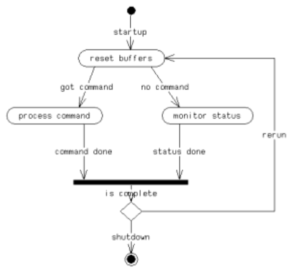
Use case diagram



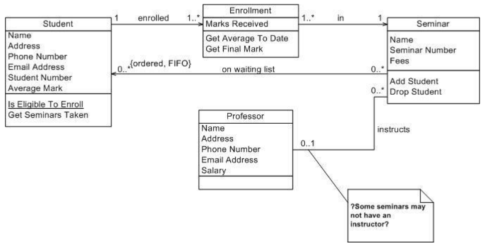
Sequence diagram



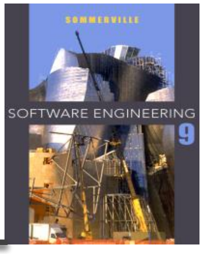
Activity diagram



Class diagram



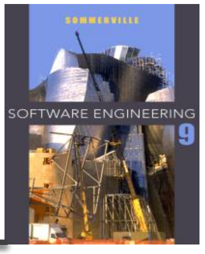
5.1 Context models

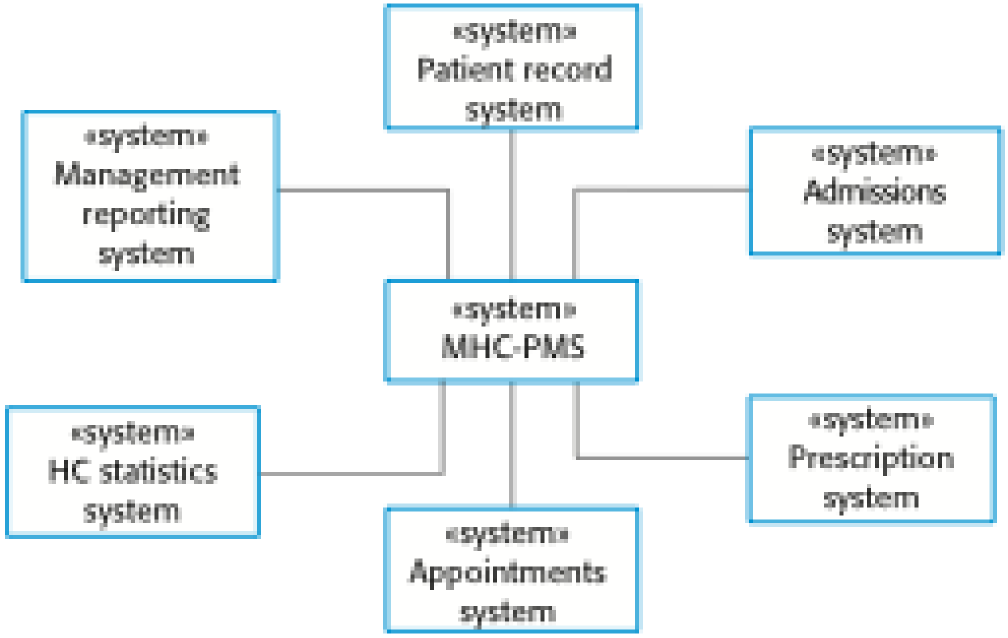


✧ Define

- context (what is outside)
- dependencies
- boundaries

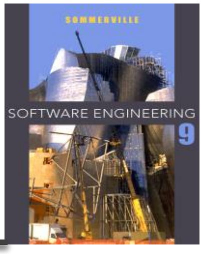
The context of the MHC-PMS




«system» Patient record system
«system» Management reporting system
«system» Admissions system
«system» MHC-PMS
«system» HC statistics system
«system» Prescription system
«system» Appointments system

shows other systems in environment

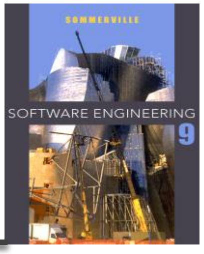
5.2 Interaction models



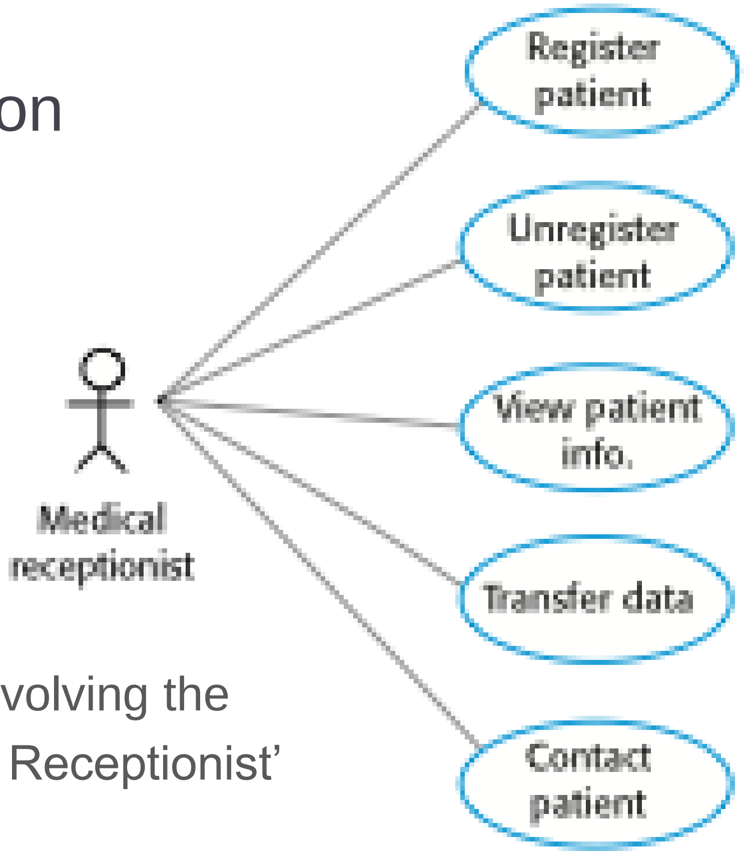
- User-system:
  - Identify user requirements.
- System-to-system:
  - Detect communication problems that may arise.
- Component-to-component
  - Detect performance and dependability problems.

- UML:
  - Use case diagrams
  - sequence diagrams

Use case modeling

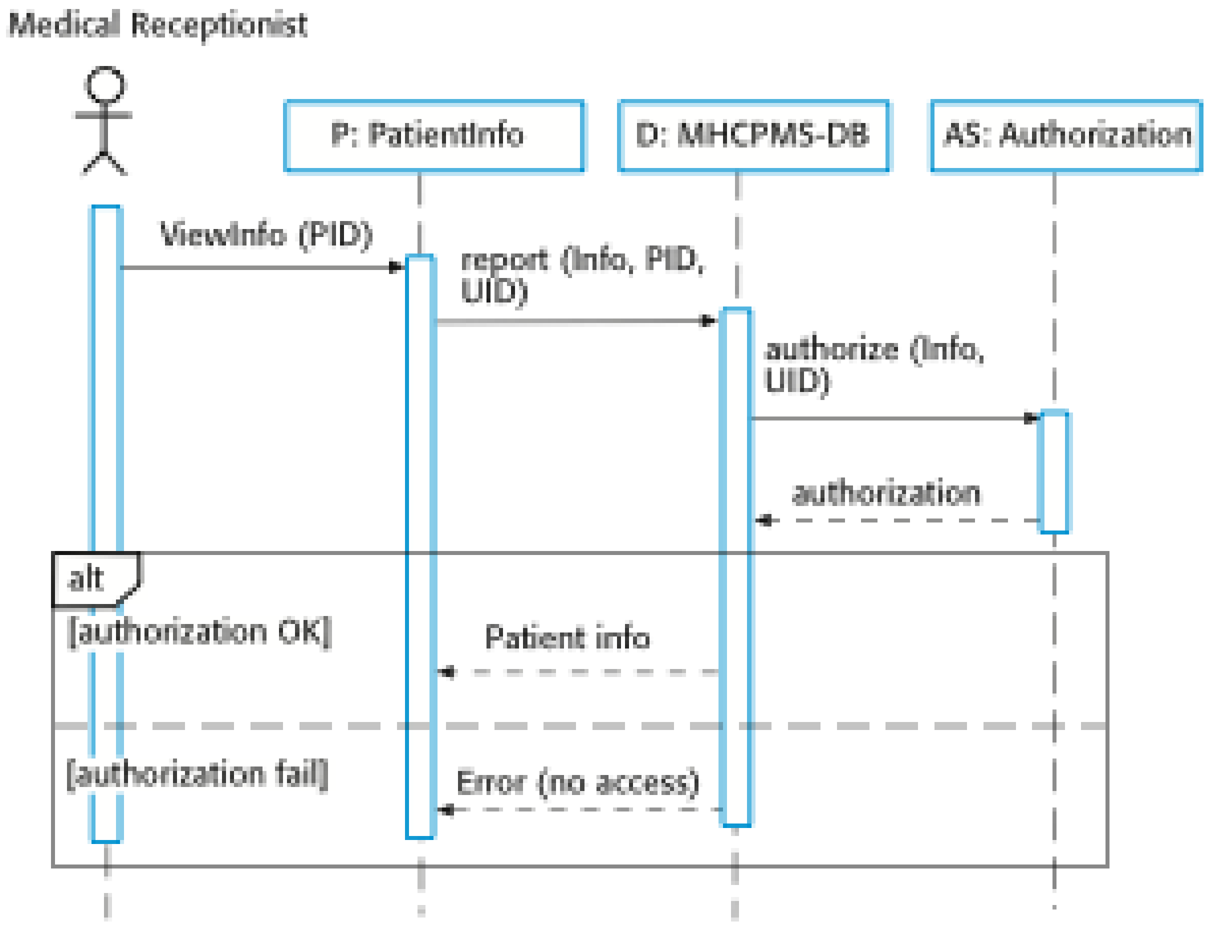


- ✧ Interaction between system and external actors
- ✧ Often used during requirements elicitation
- ✧ Purpose: provide overview

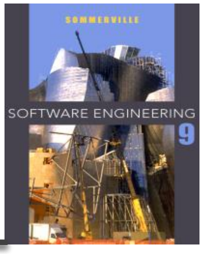


Use cases involving the role ‘Medical Receptionist’

Sequence diagram for View patient information



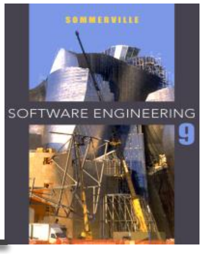
5.3 Structural models



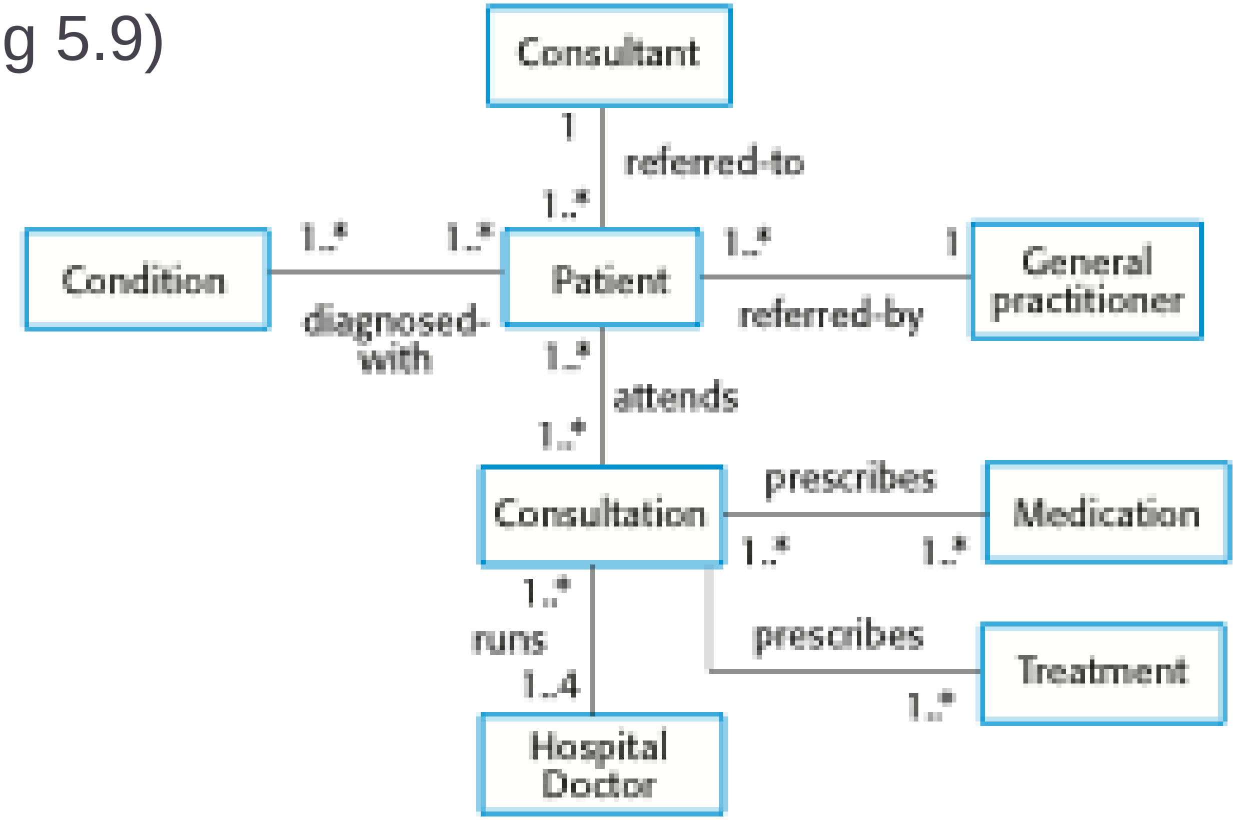
- ✧ Organization of a system in terms of components and their relationships.
- ✧ Used when discussing /designing system architecture.

static structural models	vs	**dynamic** structural models
structure of the system design		organization of the system when executing (set of interacting threads)

Classes and associations in the MHC-PMS

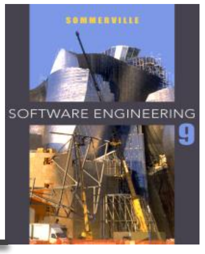


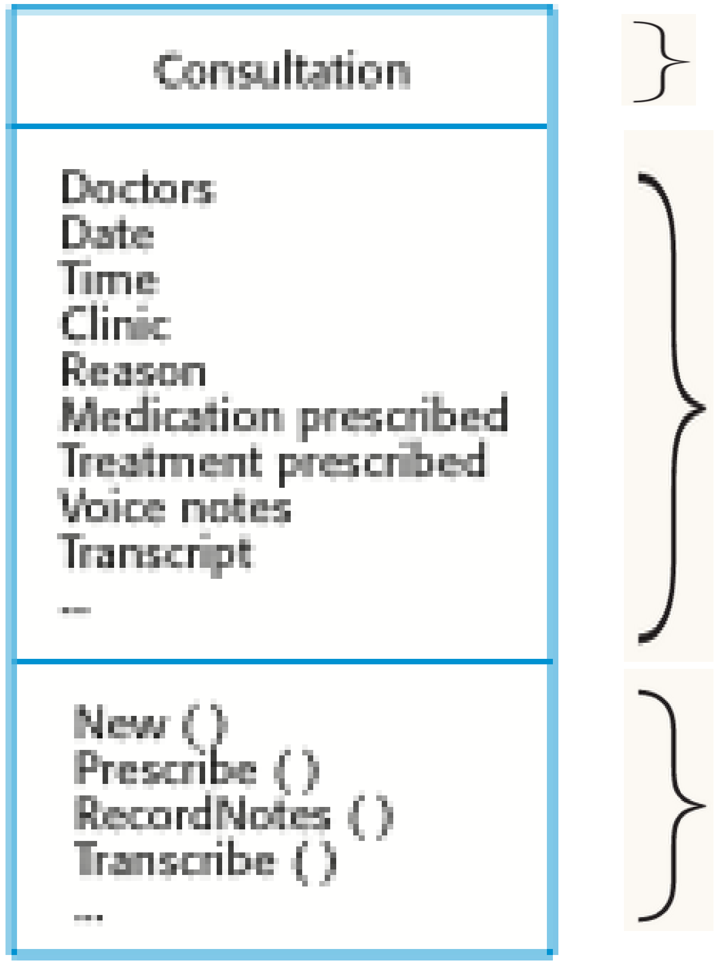
(Fig 5.9)



✧ UML (like in Fig. 5.9) can be used in database design when only attributes are considered (no operations)

More detailed class diagram: (Fig. 5.10)



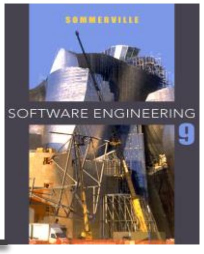


name

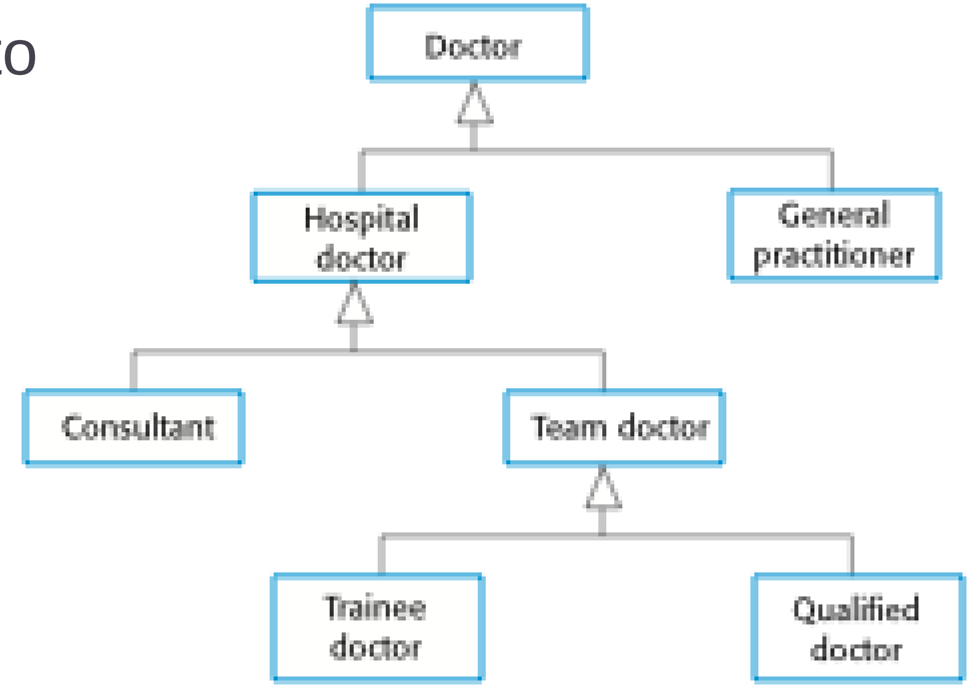
attributes

operations

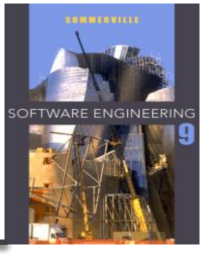
5.3.2 **Generalization** (Fig. 5.11)



- ✧ Inheritance (△)
- ✧ Good design practice to avoid code repetition



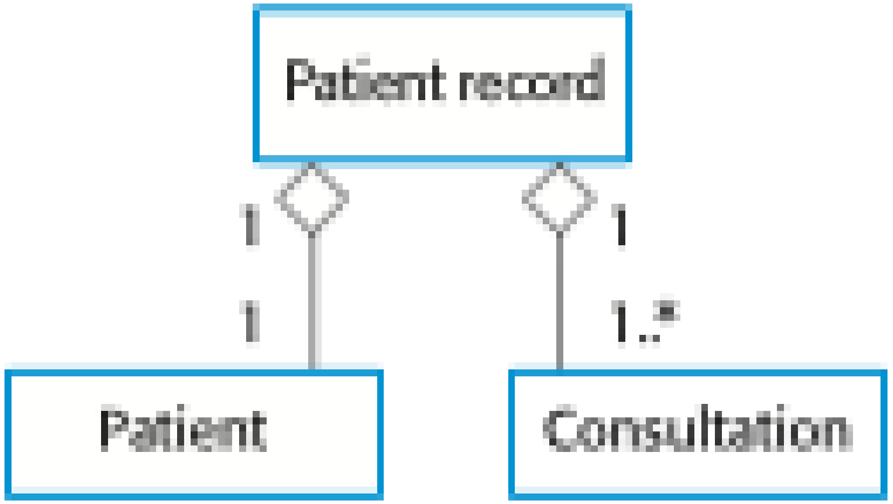
5.3.3 The aggregation association (Fig. 5.13)



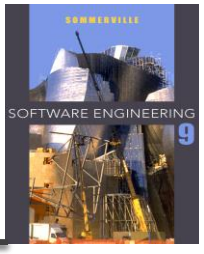
✧ Composition:

Object (whole) is composed of other objects (parts)

✧ ◇ shape next to the whole



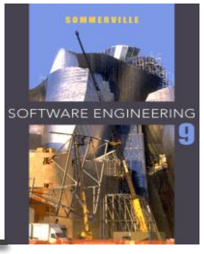
5.4 Behavioral models



- Model **dynamic** behavior of system as it is executing.
- Show what happens when system responds to stimulus from environment.
- 2 types of stimuli:

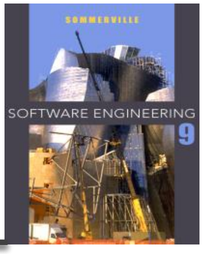
Data	**Events**
Some data arrives that has to be processed	Some event happens that triggers system processing. Events may have associated data, although this is not always the case

5.4.1 Data-driven modeling

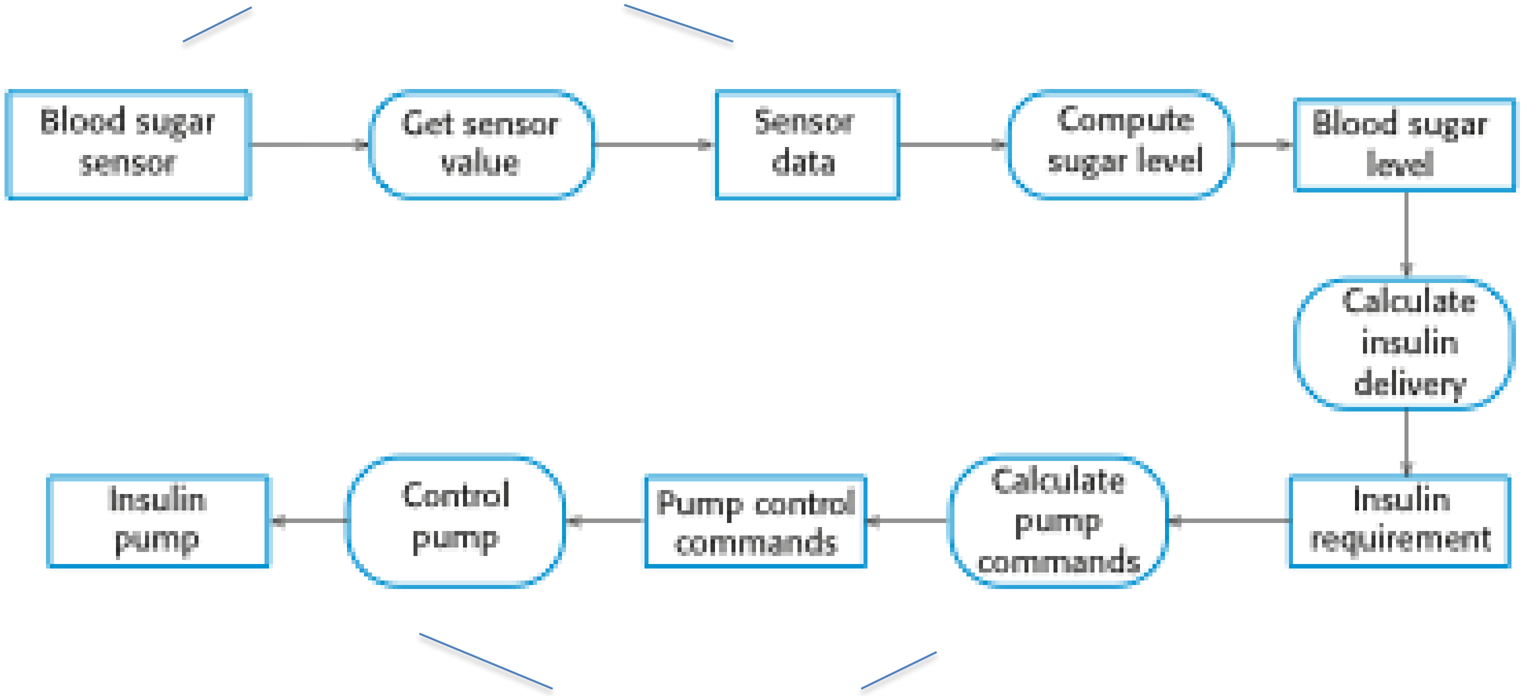


- ✧ Often used for business systems
- ✧ Show sequence of actions involved in processing input data and generating output.
- ✧ Can show end-to-end processing
- ✧ Useful during analysis of requirements

An activity model of the insulin pump's operation

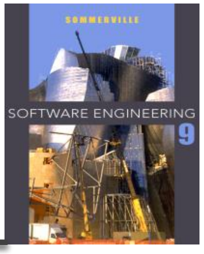


Data represented as objects



Processing steps represented as activities

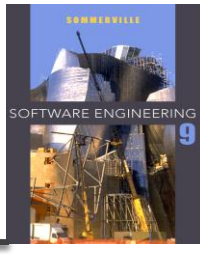
Event-driven modeling

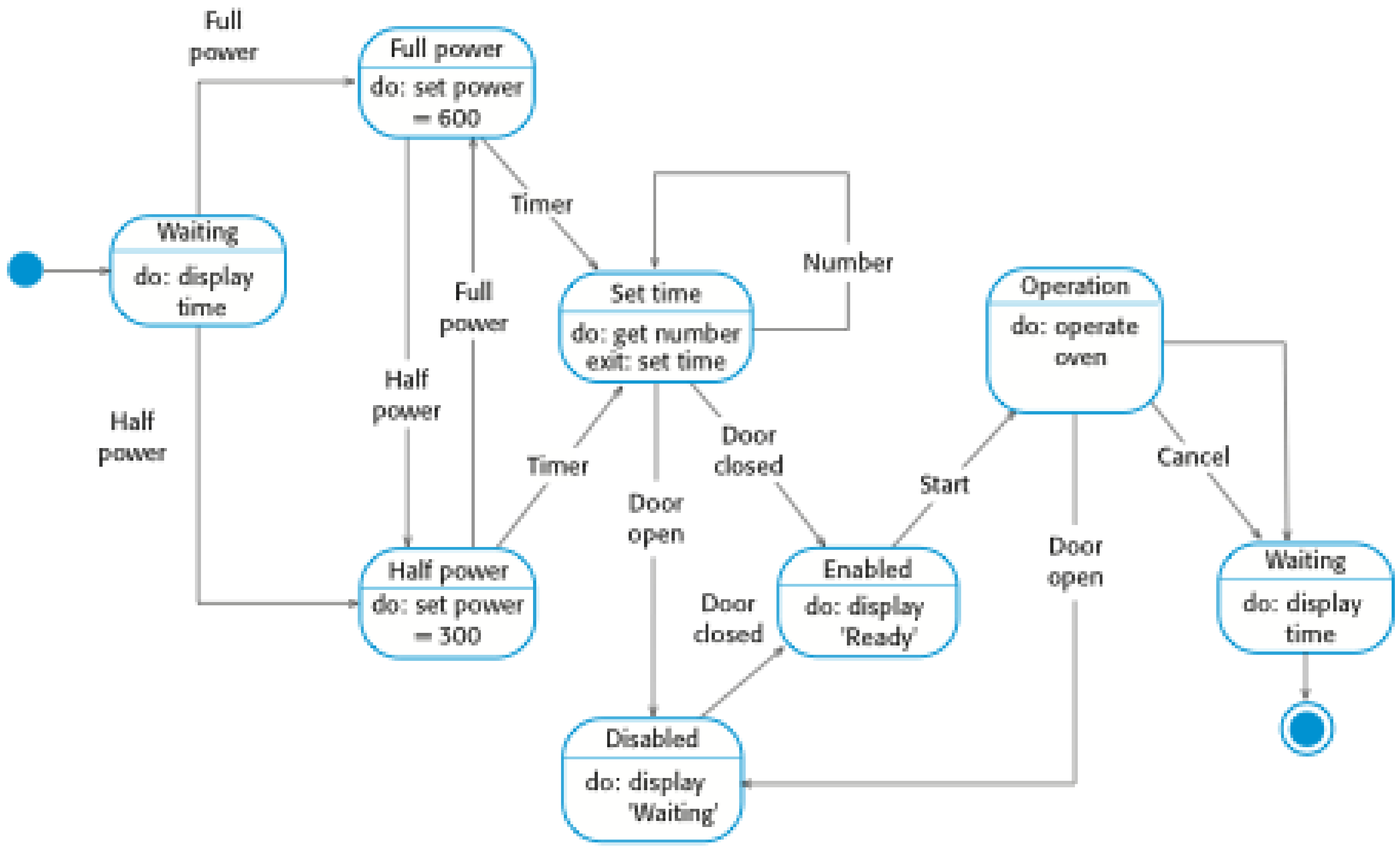


- Real-time systems are often event-driven, with minimal data processing.
- How a system responds to events (external and internal)
- Assumption:
  - Finite number of states
  - Events (stimuli) cause transition from one state to another.

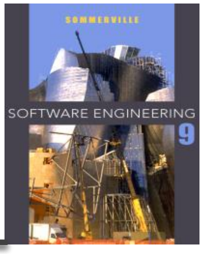
node .. System state **arc** .. event

State diagram of a microwave oven





5.5 Model-driven engineering (MDE)



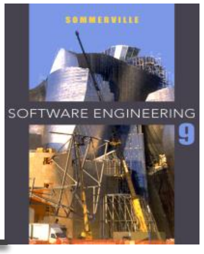
✧ Models rather than programs are principal output of development process

✧ Programs are generated automatically from the models.

✧ **MDA** .. Model-driven Architecture
focus on design and implementation

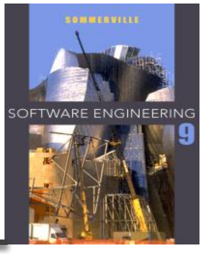
✧ **MDE** .. Model-driven Engineering
wider scope; all aspects of software engineering
(incl. model-based requirements engineering ..)

Executable UML



- ✧ Fundamental notion behind model-driven engineering: possible to completely automate transformation of models to code
- ✧ UML was designed for supporting / documenting not as programming language
- ✧ => subset of UML 2, called Executable UML or xUML.

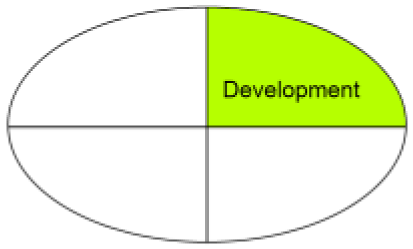
Usage of model-driven engineering



✧ Cons

- Model's abstraction not necessarily right for implementation. (e.g. reuse)
- In large, long-lifetime systems implementation is not the major problem.
  Requirements engineering, dependability, maintainability.. more important than fast development

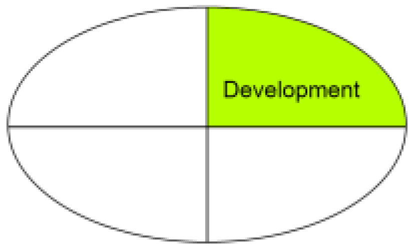
Architectural Design



✧ Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system.

- Identifying major software components (sub-systems)
- how they interoperate (communication / control)

Architectural representations



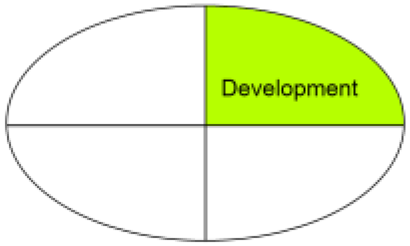
✧ Most often used: Simple, informal block diagrams showing entities and relationships

Describe vs Document

e.g. to facilitate

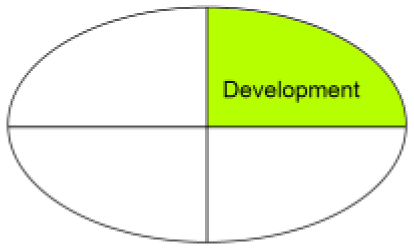
a discussion

Choosing an architectural style / pattern



- ✧ Performance
  - Minimize communication; small number of components local
- ✧ Security
  - Layered architecture, most critical assets in innermost layer
- ✧ Safety
  - Keep safety-critical features in small number of components
- ✧ Availability
  - Redundant components
- ✧ Maintainability
  - Fine-grained, self-contained, replaceable components

TODO:

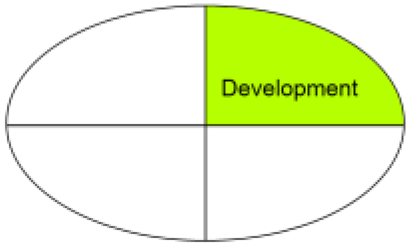


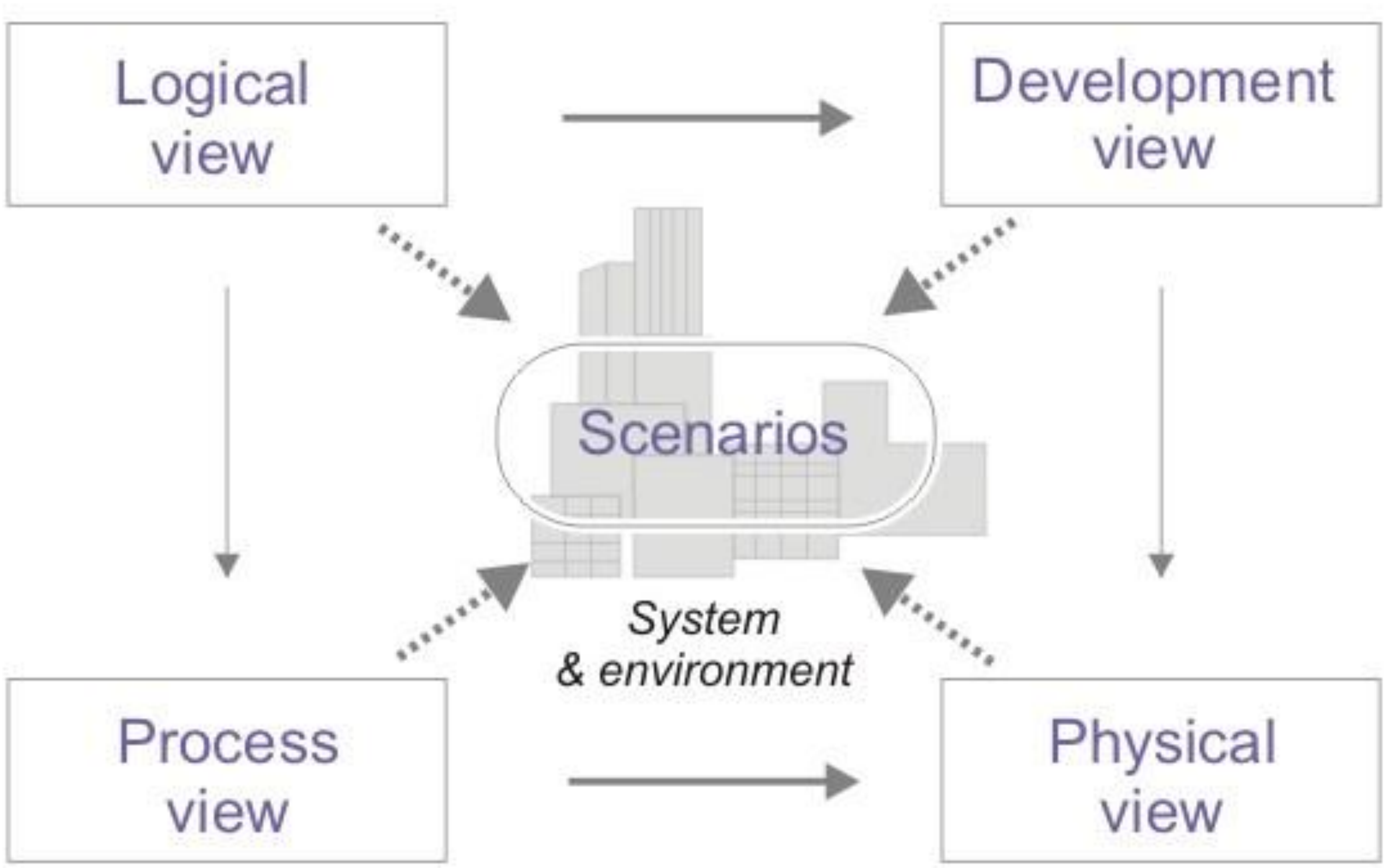
Explain why design conflicts might arise when designing an architecture for which both

a) performance and maintainability
b) availability and security

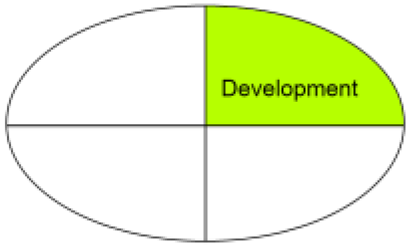
are the most important non-functional requirements.

4 + 1 view model of software architecture



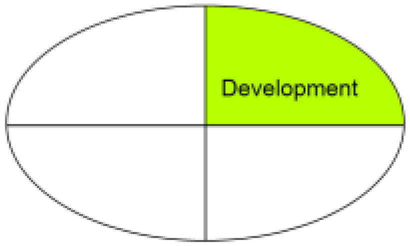


4 + 1 view model of software architecture



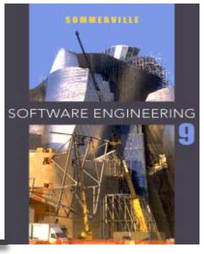
- ✧ Logical view (functionality provided to end-user)
  objects / object classes matched to functional requirements
- ✧ Process view (dynamic aspects)
  system processes and their interactions
- ✧ Development view (for software management)
  how software is decomposed for development.
- ✧ Physical view (system engineer’s point of view)
  deployment, how distributed across processors, ..
- ✧ Scenarios: (+1)
  small set of use cases / scenarios to illustrate architecture

6.3 Architectural patterns



- ✧ **Architectural pattern** … stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Purpose: representing, sharing and reusing knowledge.
- ✧ Can use tabular and graphical descriptions.
- ✧ Patterns should include information about when they are useful and when not

Architectural Patterns / Styles:



a) Layered architecture

b) Repository architecture

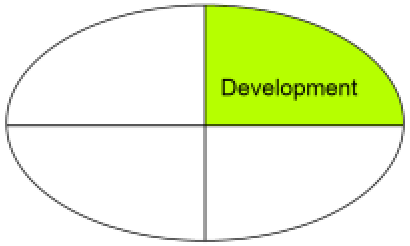
c) Client-server architecture

d) Pipe and filter architecture

e) MVC

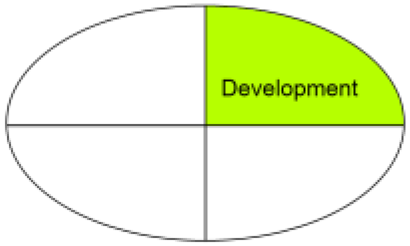
✧ . . .

a) Layered architecture



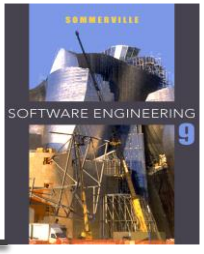
- ✧ Models interfacing of sub-systems.
- ✧ Each layer provides set of services.
- ✧ Pro:
  - Supports incremental development of sub-systems in different layers.
  - When layer interface changes, only the adjacent layer is affected.
- ✧ Con:
  - Can be artificial to structure systems in this way.

The Layered architecture pattern



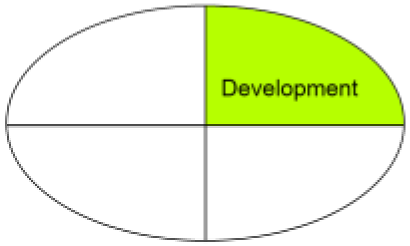
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Example: ISO OSI



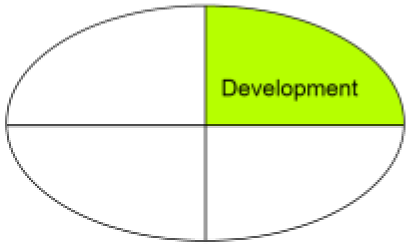
http://www.youtube.com/watch?v=Kb4hVvlCx40&feature=related

b) Repository architecture



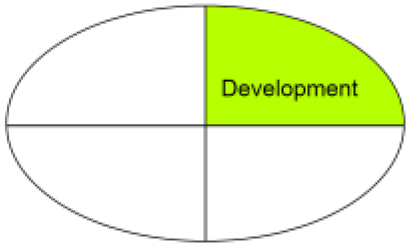
✧ All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.

The Repository pattern



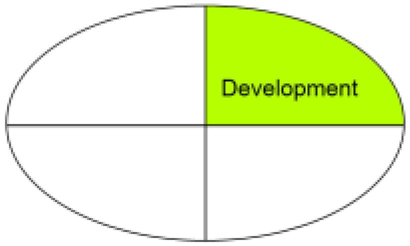
Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that have to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

c) Client-server architecture



- ✧ Distributed system model
- ✧ Shows how data and processing is distributed across a range of components.
  - ▪ Can be implemented on a single computer.

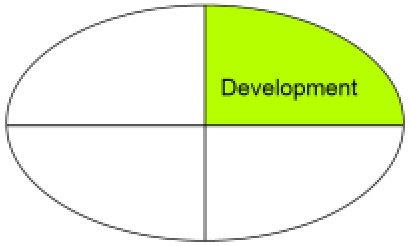
c) Client-server architecture



Requires:

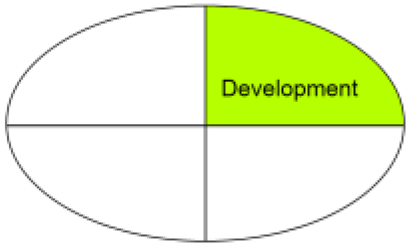
- ✧ Set of stand-alone servers which provide specific services such as printing, data management, etc.
- ✧ Set of clients which call on these services.
- ✧ Network which allows clients to access servers.

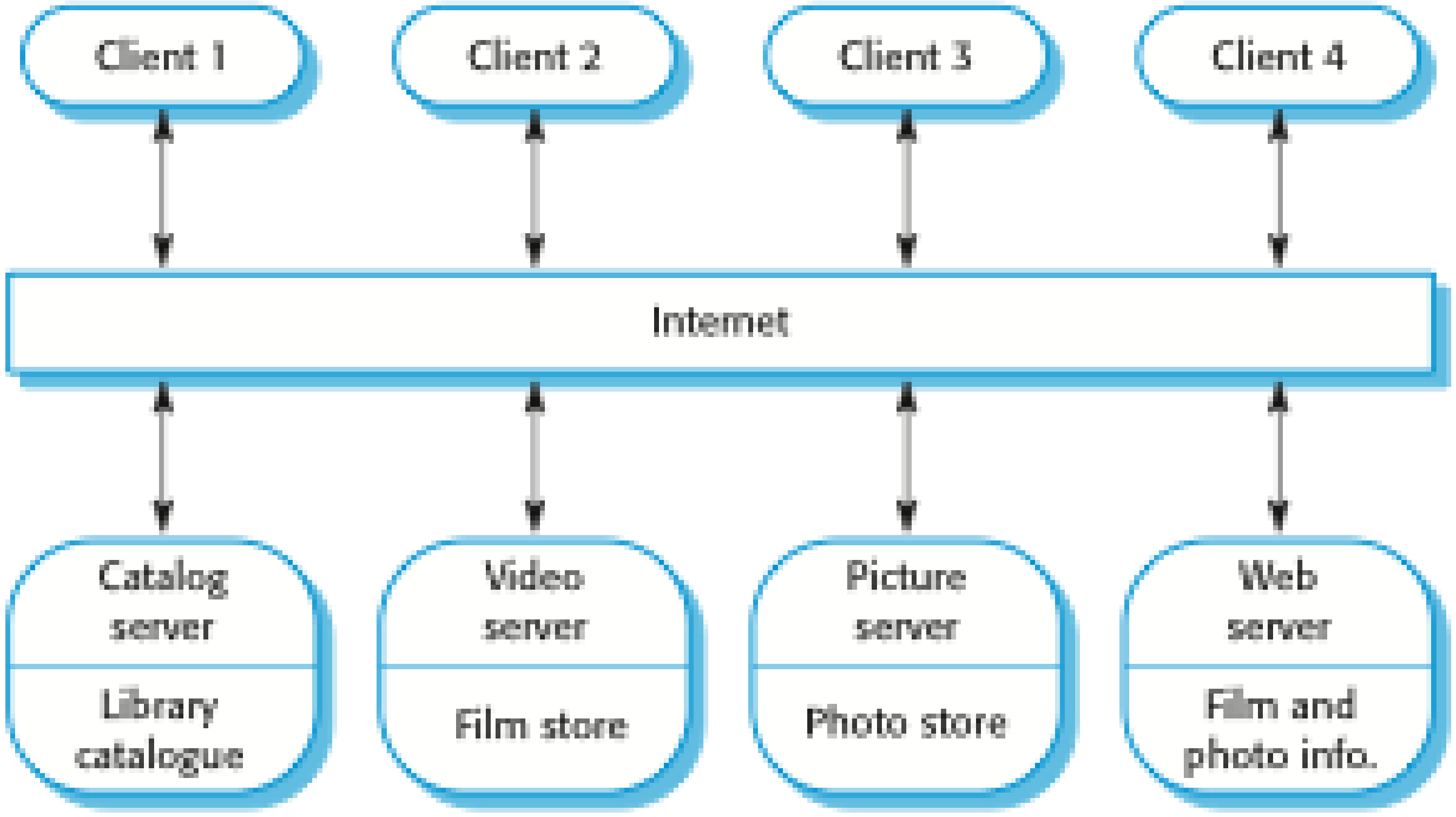
The Client–server pattern



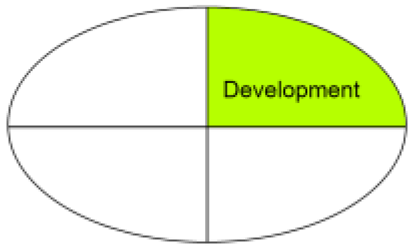
Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

A client–server architecture for a film librai





d) Pipe and filter architecture



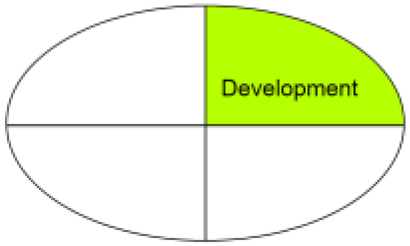
- ✧ Functional transformations

  process inputs to produce outputs.

- ✧ Common for patch processing systems (data processing systems)

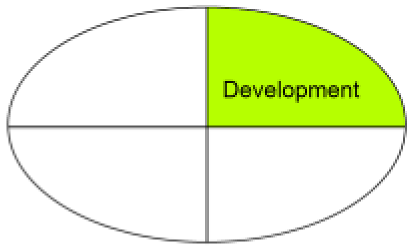
- ✧ Not really suitable for interactive systems.

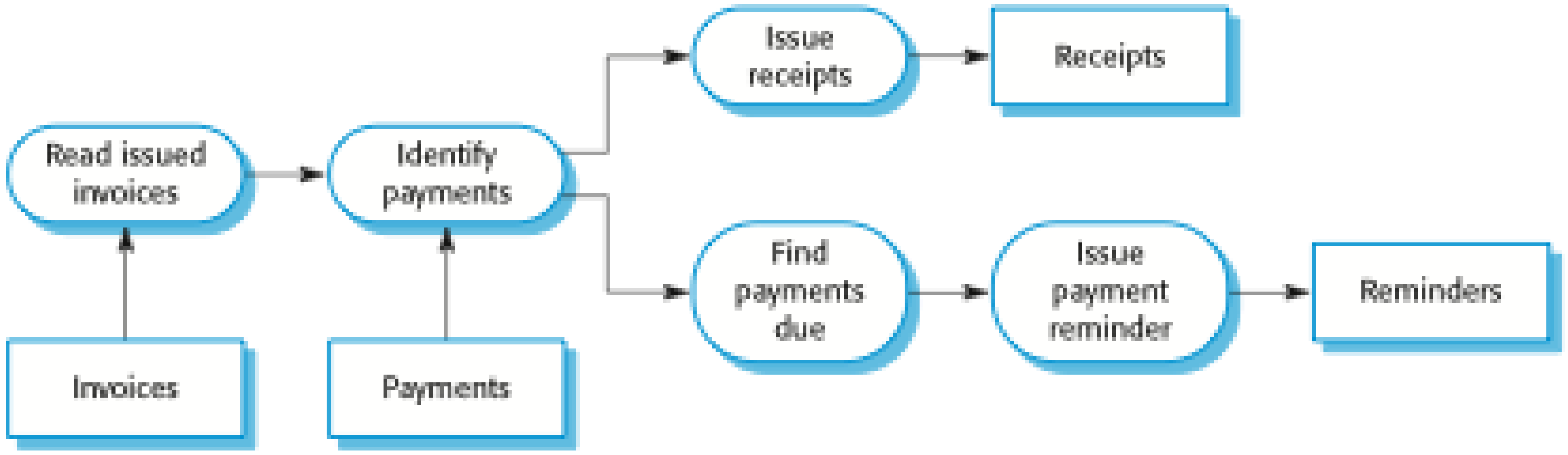
The pipe and filter pattern



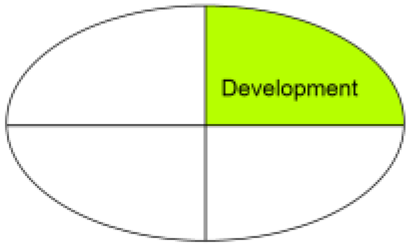
Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and provide its output in the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

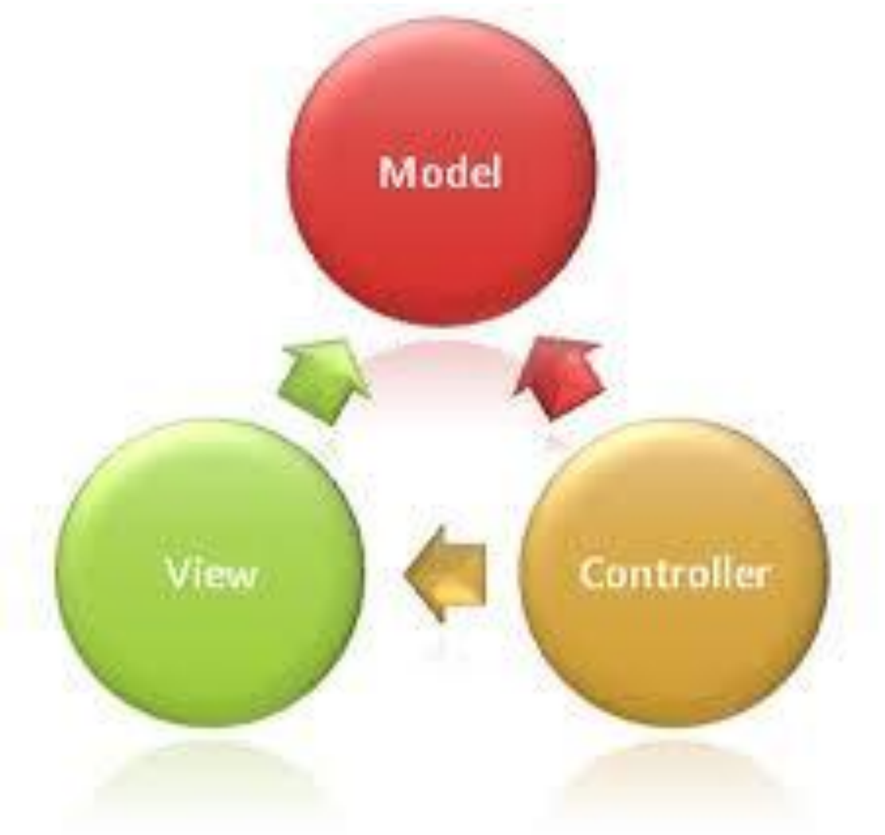
An example of the pipe and filter architectu



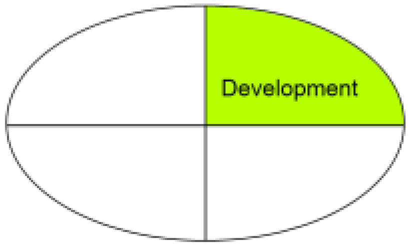


MVC (Model-View-Controller)



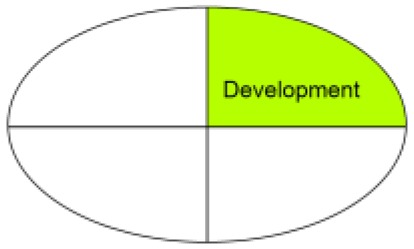


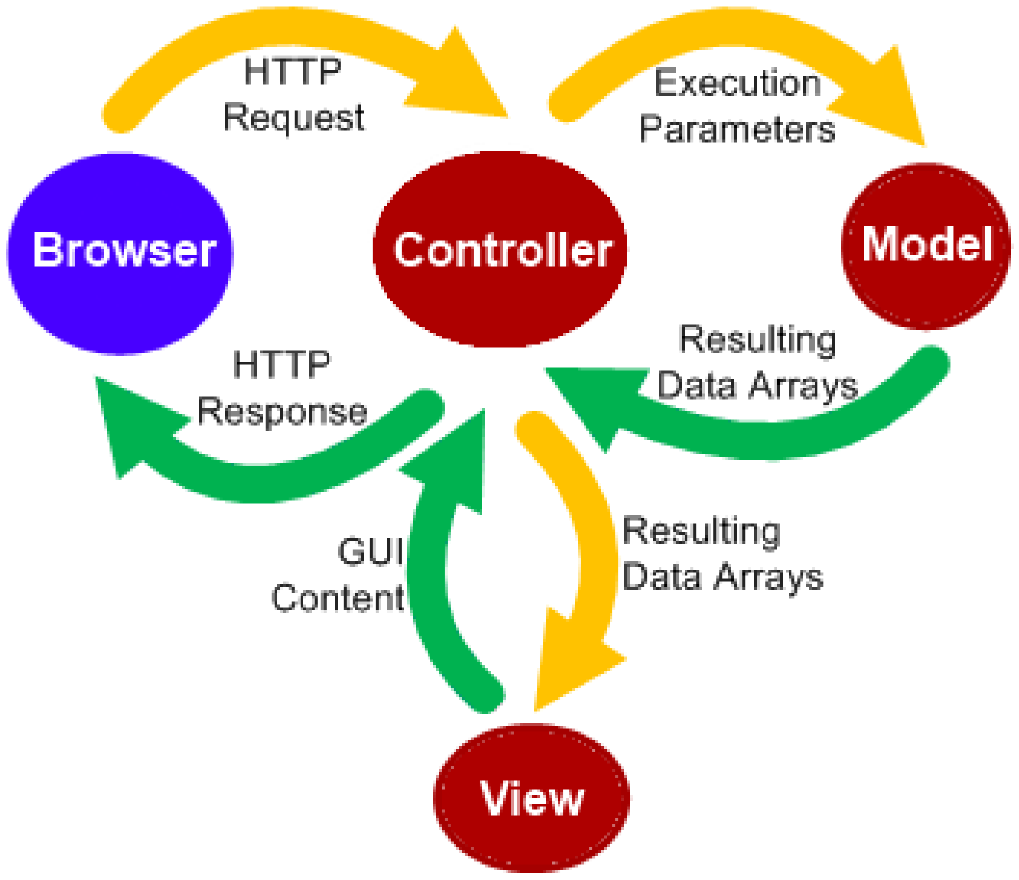
MVC (Model-View-Controller)



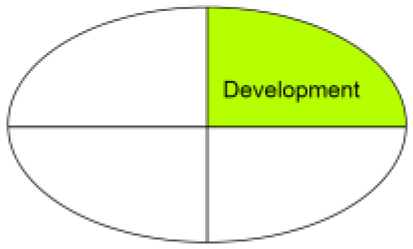
Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Web application architecture using the MVC pattern





6.4 Application architectures



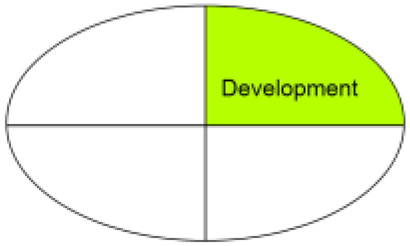
✧ Application systems are designed to meet business or organizational needs.

✧ Businesses have much in common:

- Hire people
- Issue invoices
- Keep accounts . .

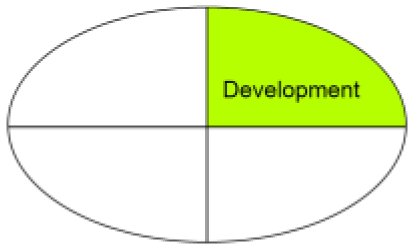
✧ ERP (Enterprise Resource Planning)

Examples of application types



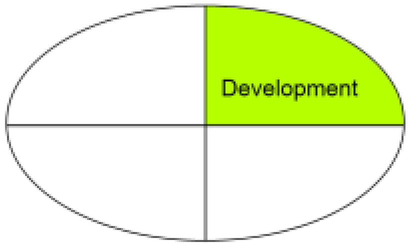
- Data processing applications
  - Data driven, process data in batches
- Transaction processing applications
  - Data-centered, user requests and updates information in db
- Event processing systems
  - System actions depend on interpreting events environment.
- Language processing systems
  - users' intentions specified in formal language that is processed and interpreted

Transaction processing systems



- Most common type of interactive business system
- Database centered
- Process user requests for information and update database
- Transaction used to maintain integrity of data
- Examples:
  banking system, e-commerce, booking system

What is a transaction?



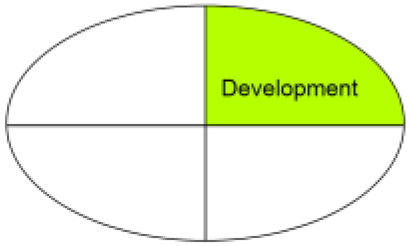
- **Transaction** (database transaction)

  sequence of operations that succeeds or fails as a unit

- **Transaction** (from user perspective)

  coherent sequence of operations that satisfies a goal (e.g. find flight from London to Paris, buy a book, withdraw money from ATM ..)

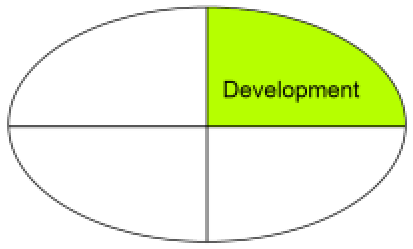
6. 4. 2 Information systems architecture

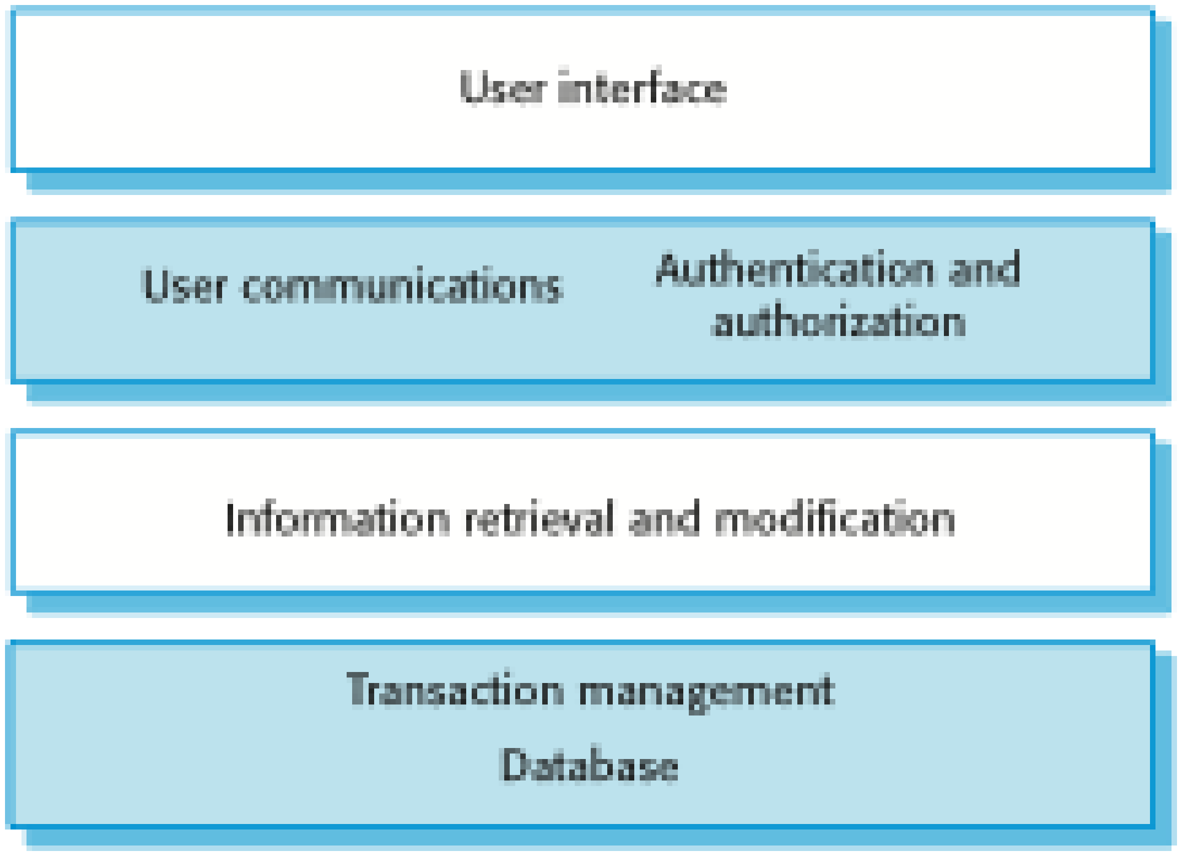


- ✧ Information systems allow controlled access to a large base of information
- ✧ Examples:

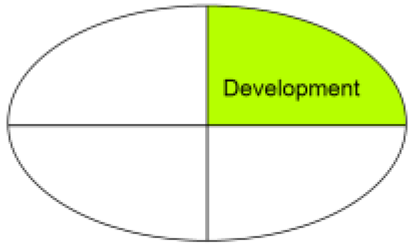
  Library catalog, flight timetable, drug information system

Layered Information system architecture



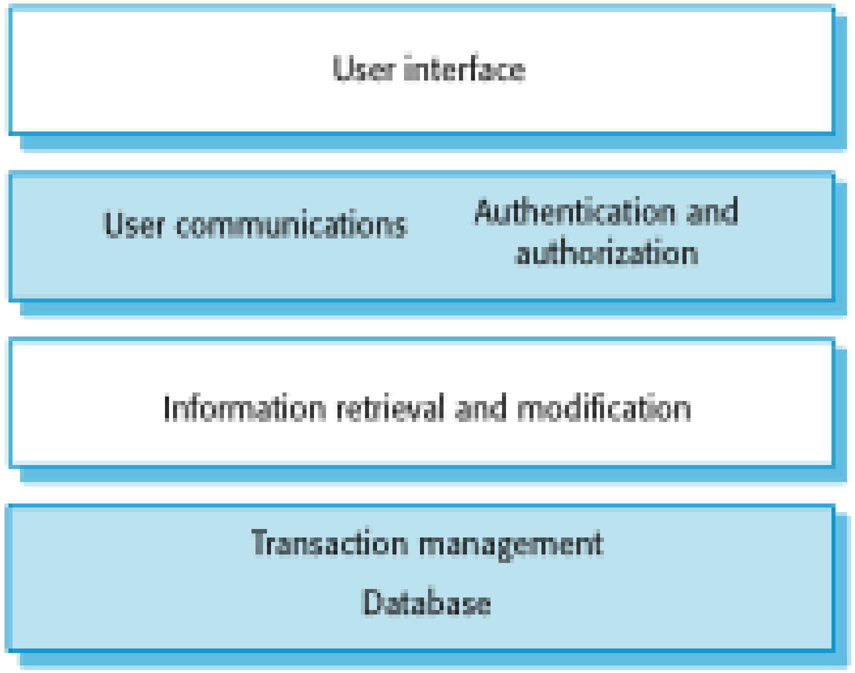

User interface
User communications
Authentication and authorization
Information retrieval and modification
Transaction management
Database

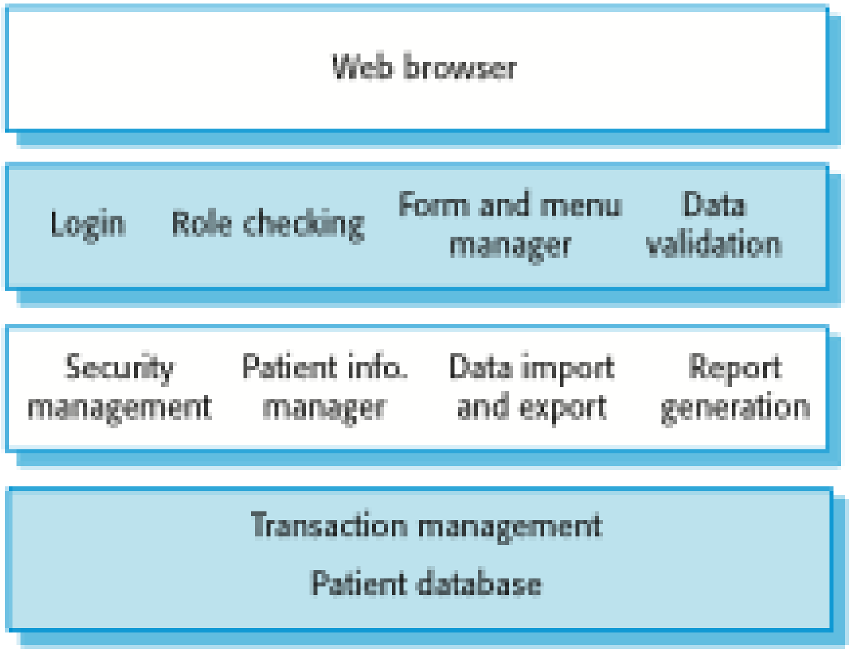
Layered Information system architecture

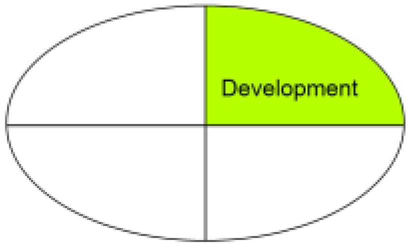


General Model:

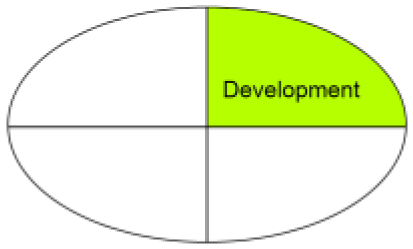
Concrete Example


User interface
User communications
Authentication and authorization
Information retrieval and modification
Transaction management
Database





Composite Design Pattern

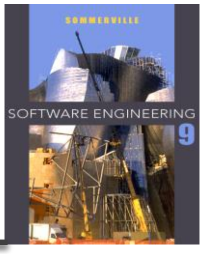


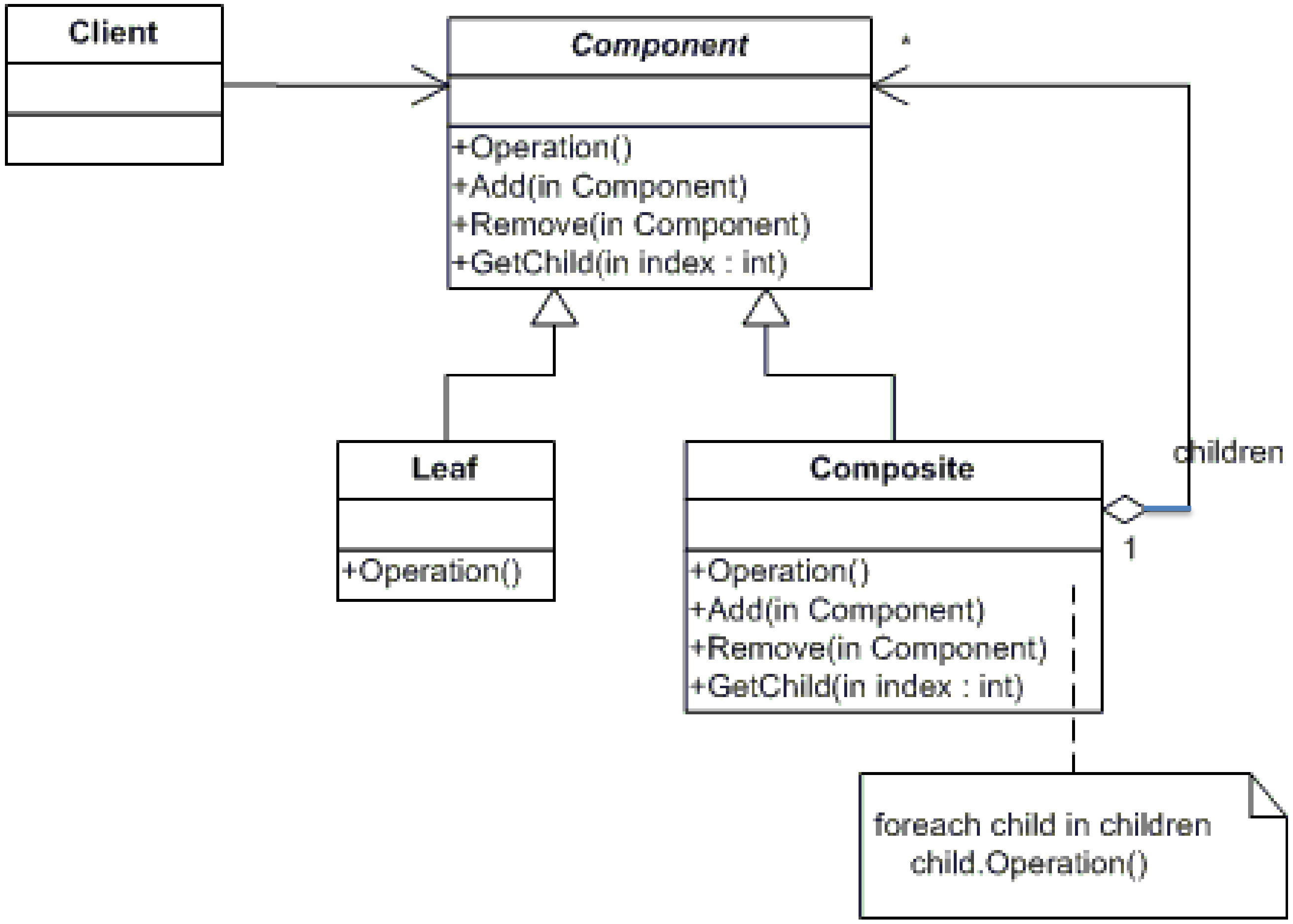
Composite Design Pattern

Description:

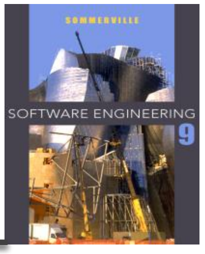
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

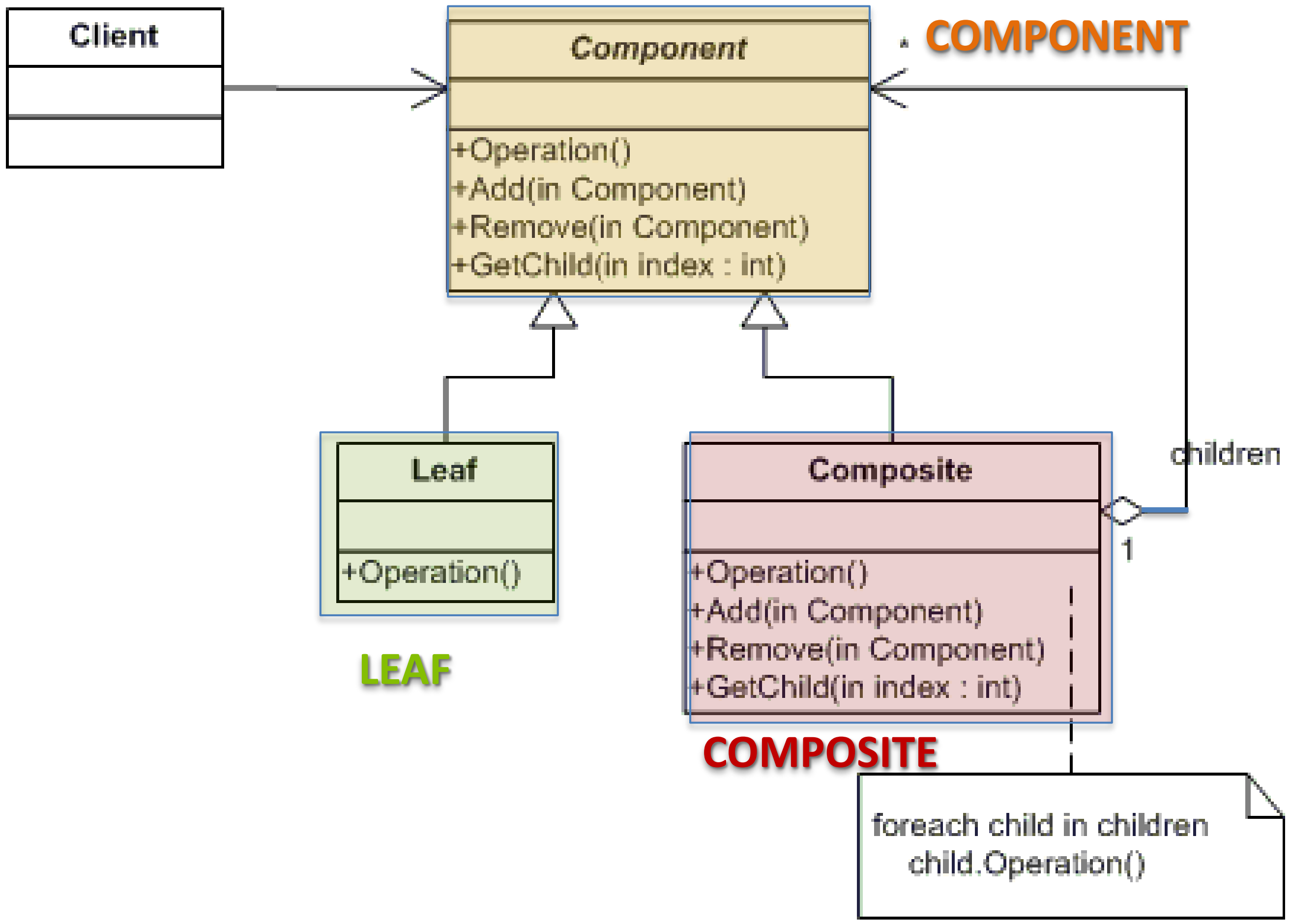
Composite Pattern



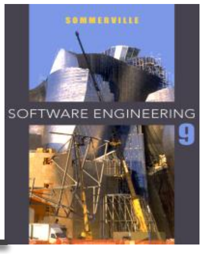


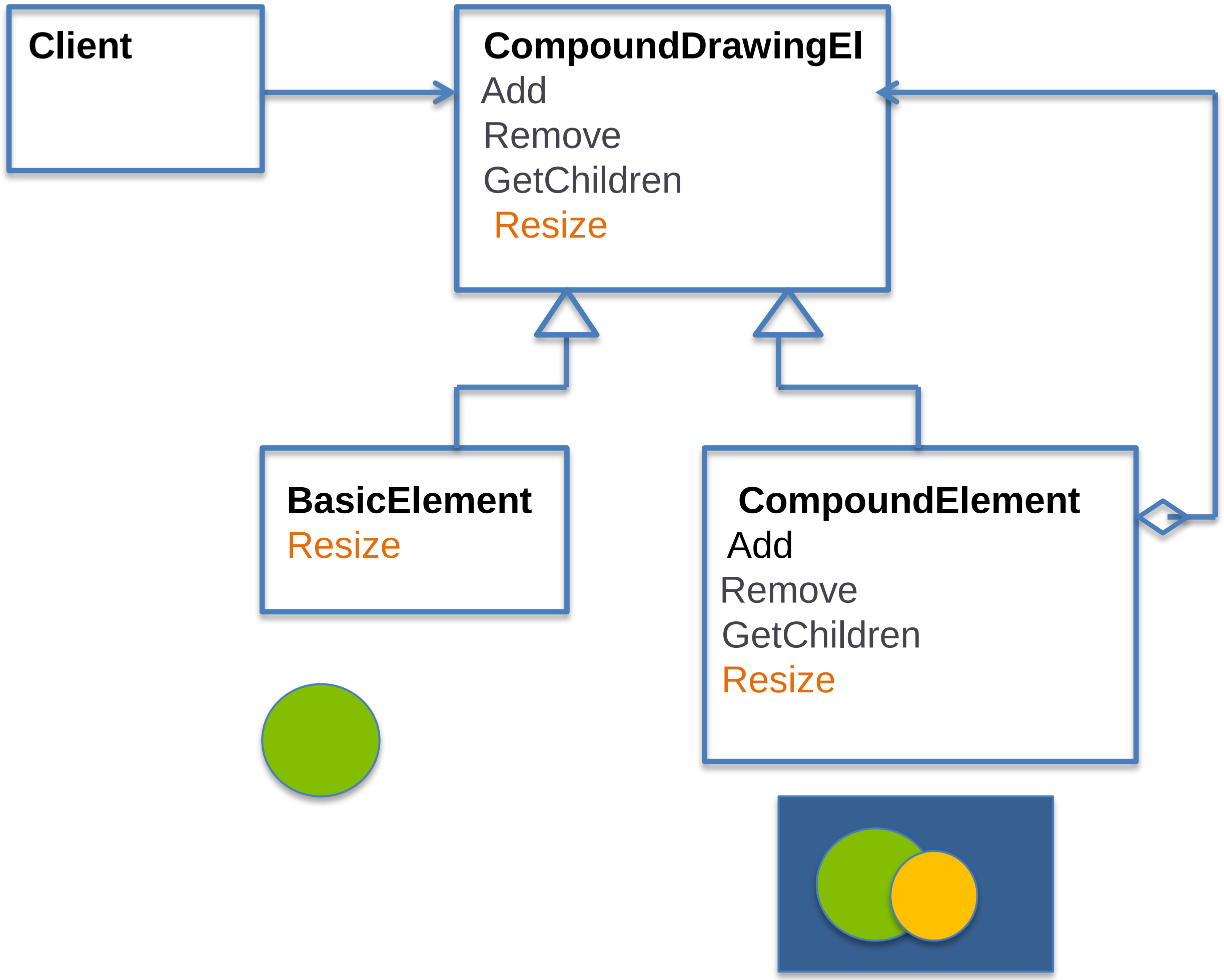
Composite Pattern



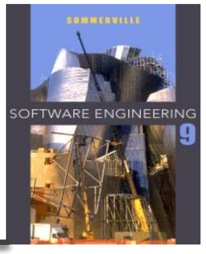


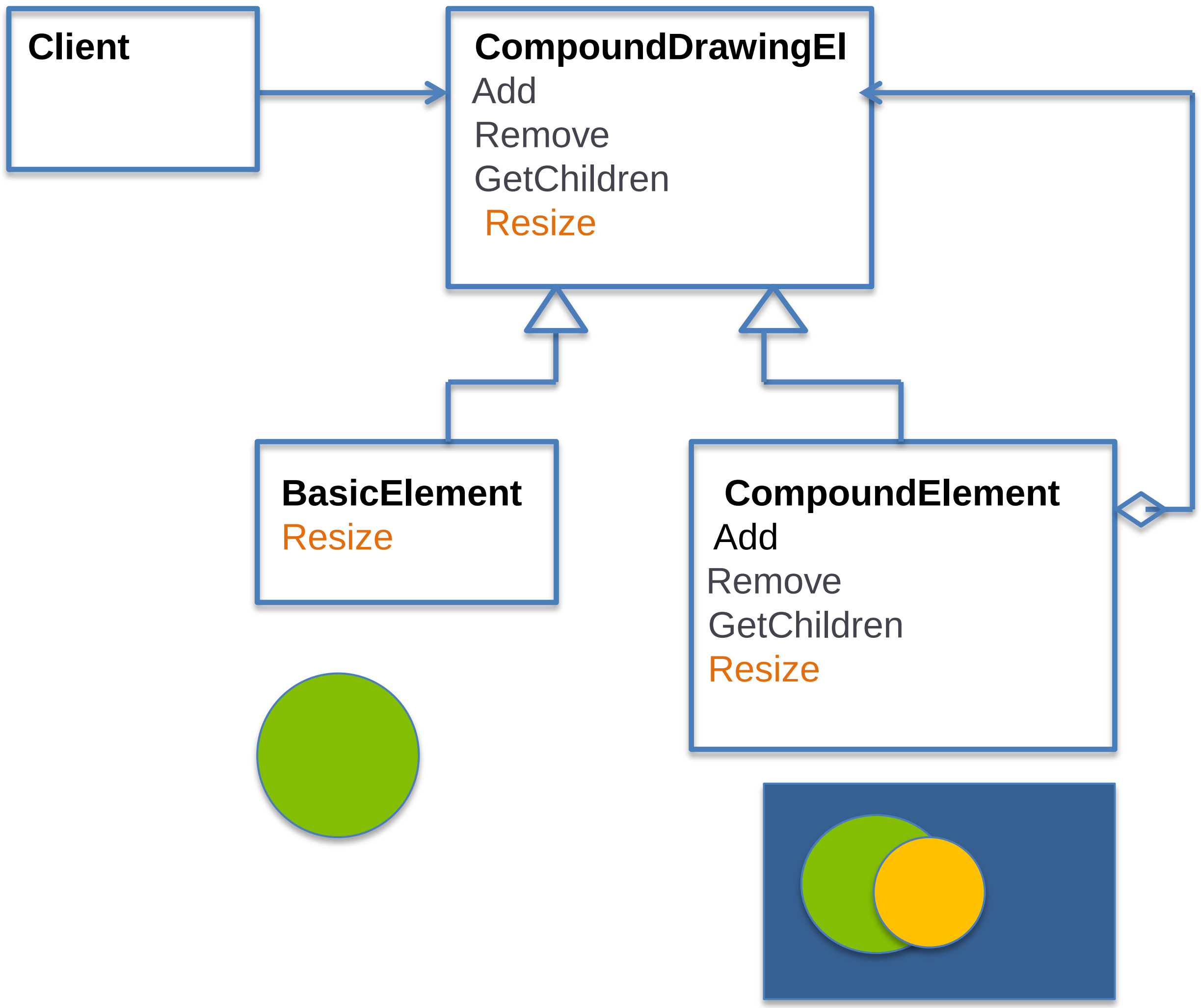
Example1: Drawing Element



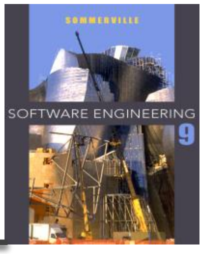


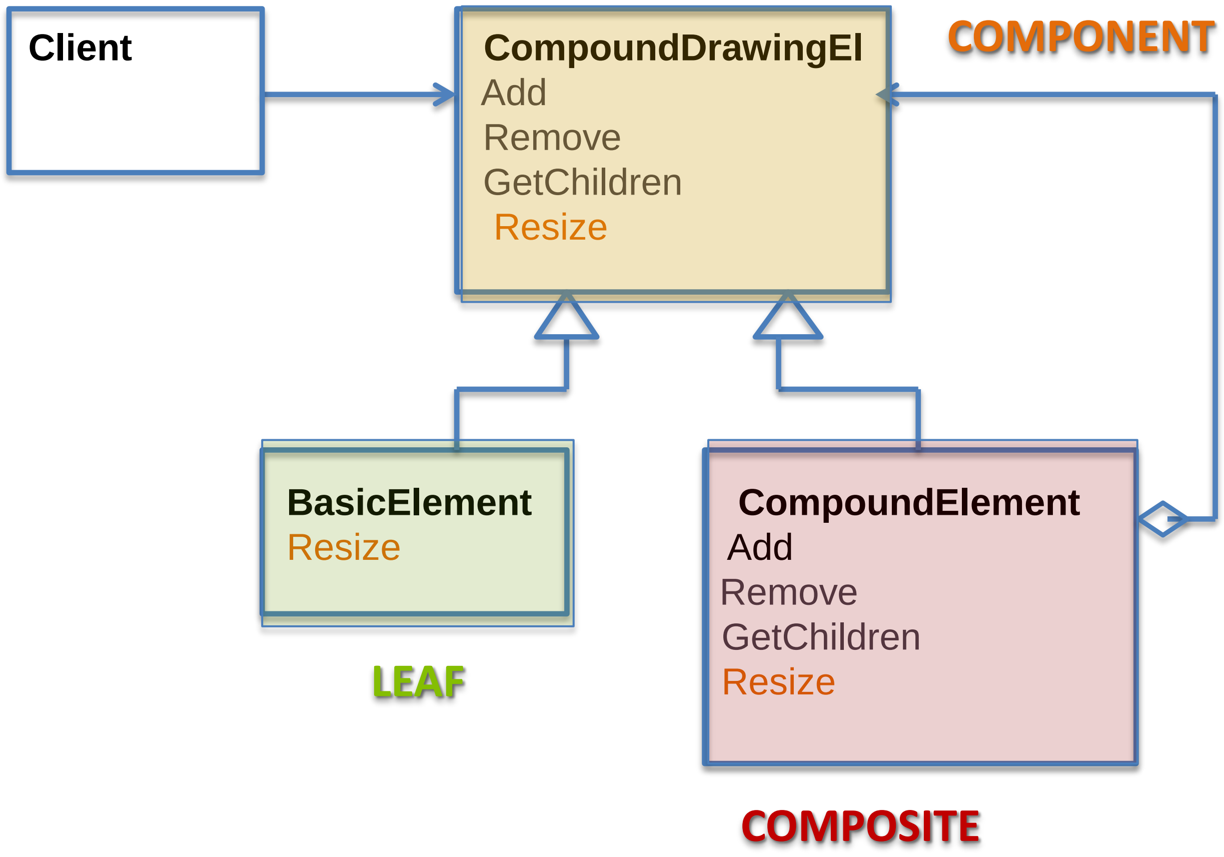
Example1: Drawing Element



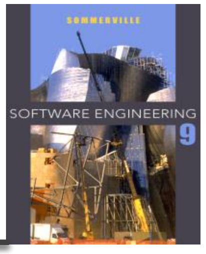


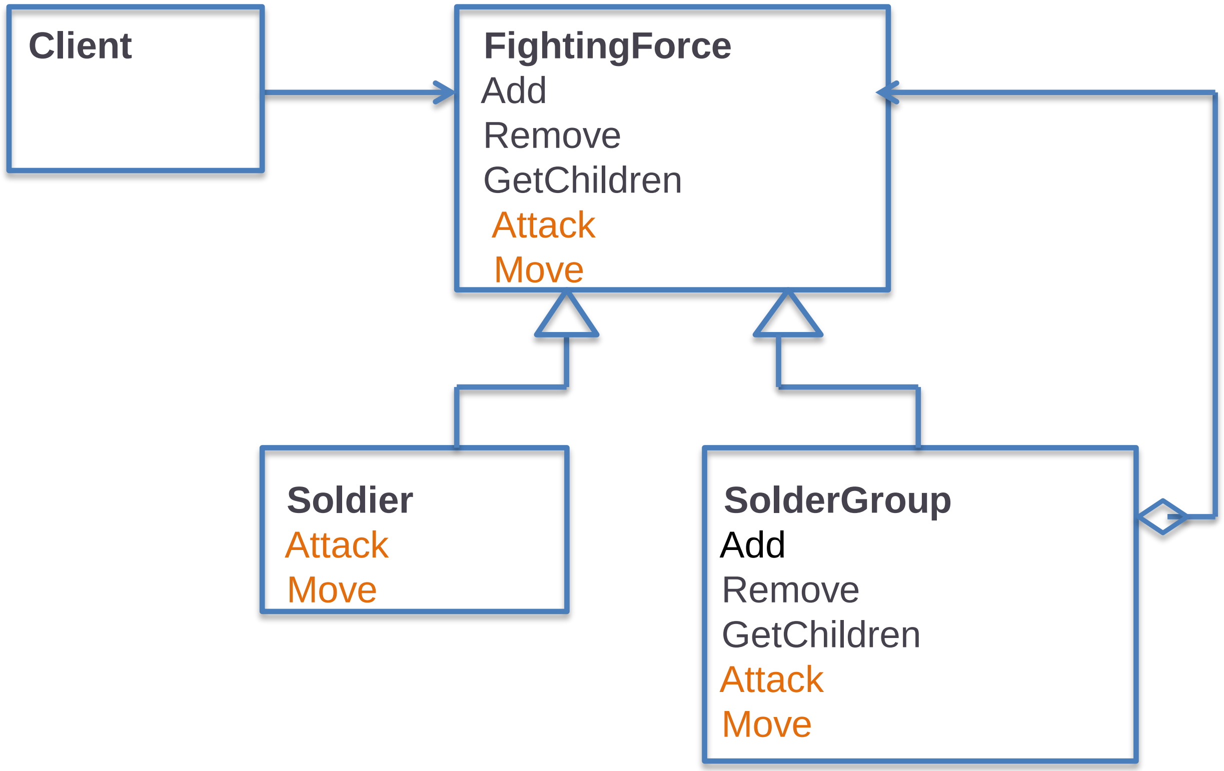
Example1: Drawing Element



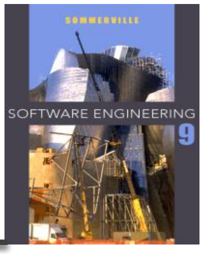


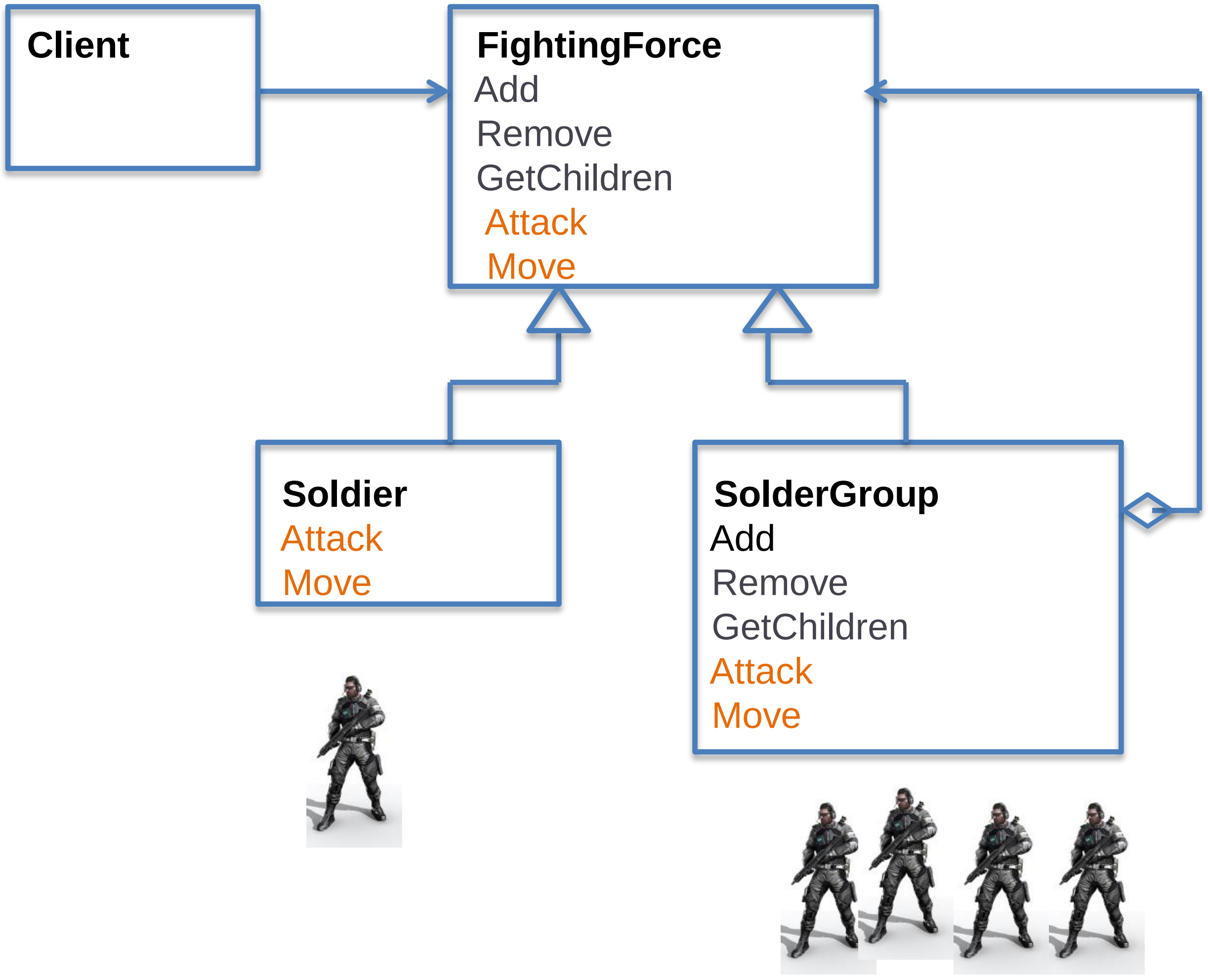
Example2: Fighting Force



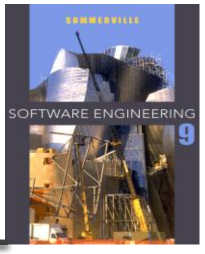


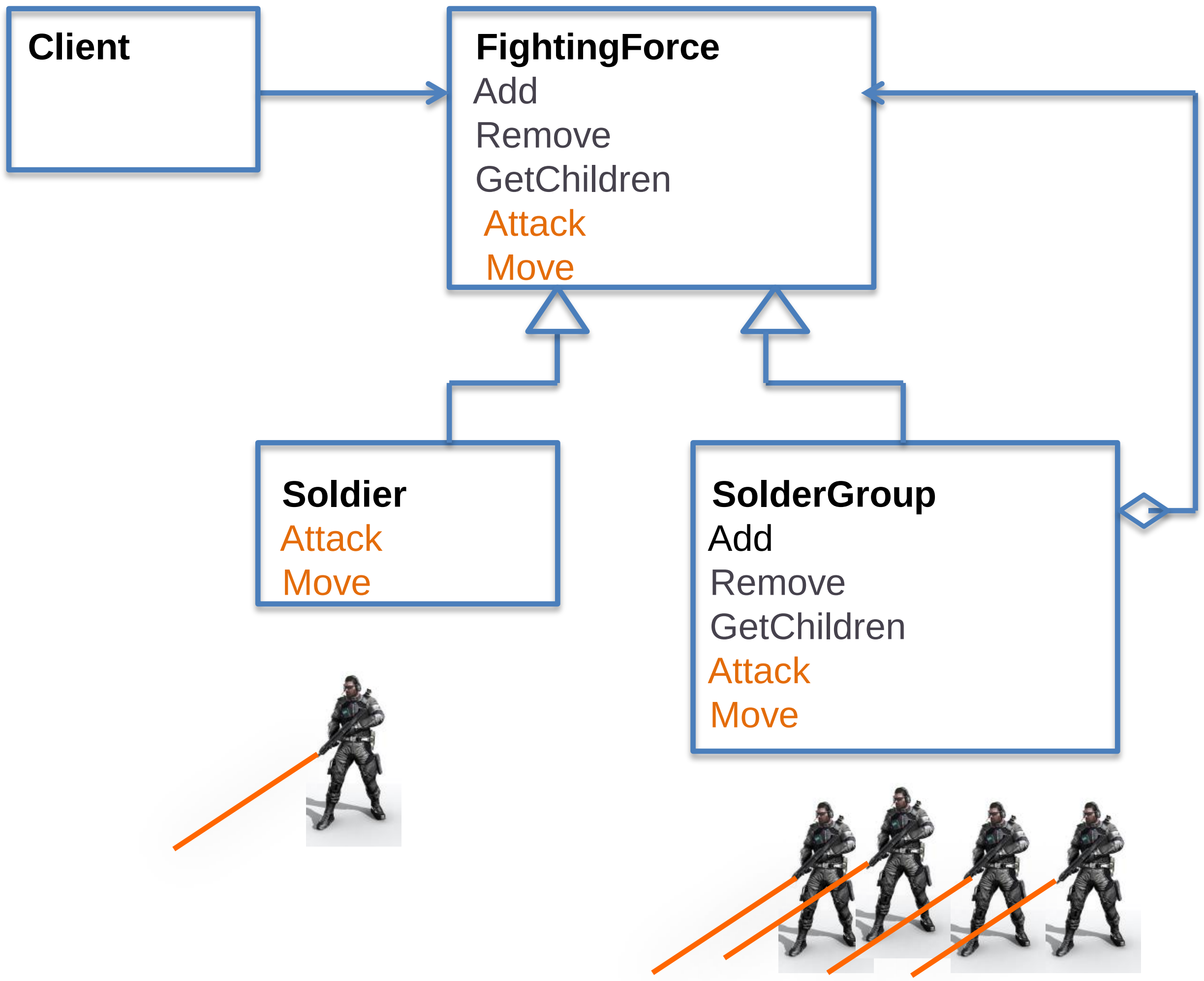
Example2: Fighting Force



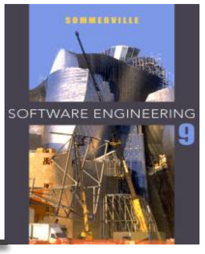


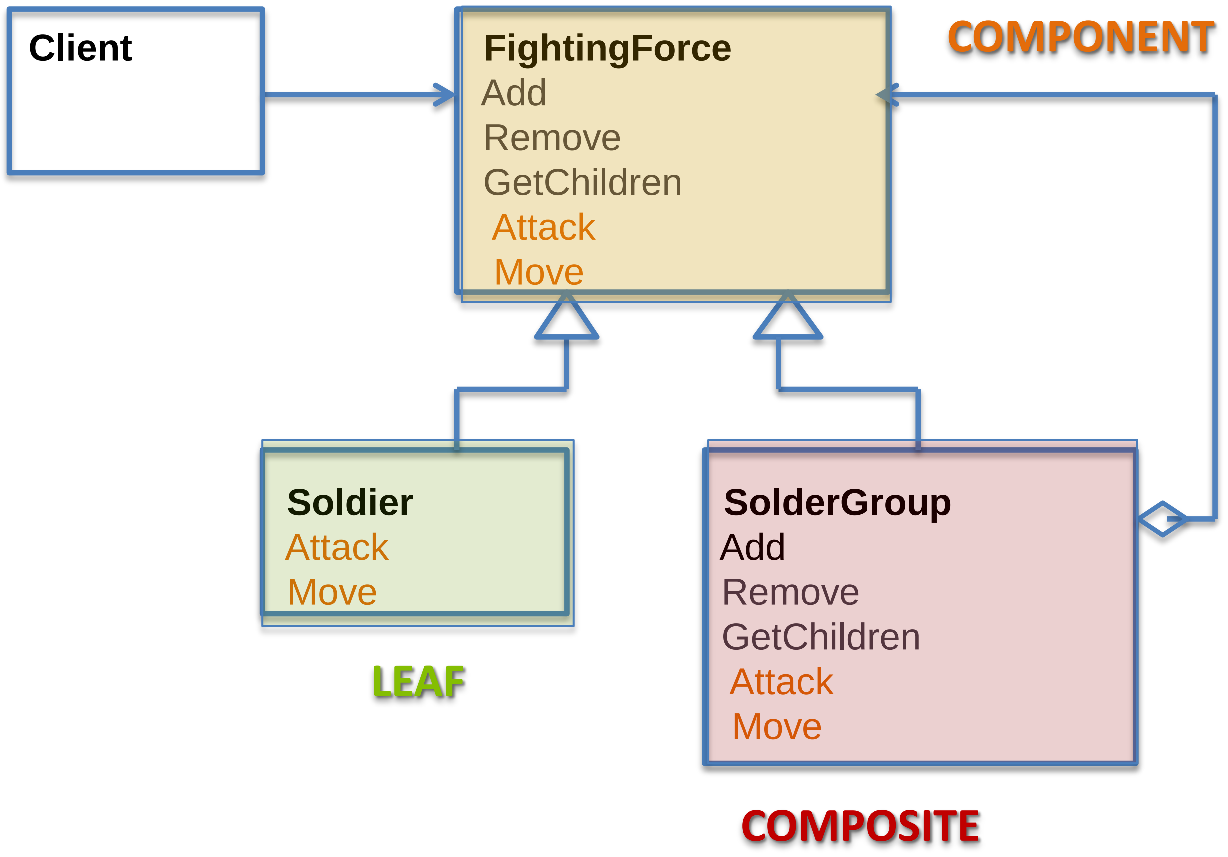
Example2: Fighting Force





Example2: Fighting Force




Client
FightingForce
Add
Remove
GetChildren
Attack
Move
COMPONENT
Soldier
Attack
Move
LEAF
SolderGroup
Add
Remove
GetChildren
Attack
Move
COMPOSITE