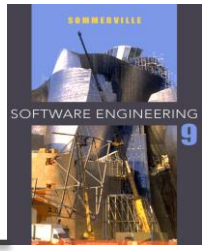
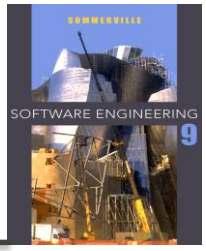


Chapter 6 – Architectural Design

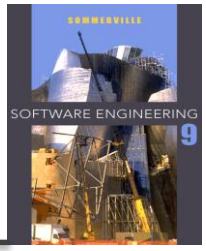
Lecture 1

Topics covered



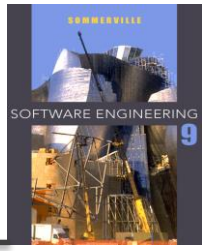
- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures

Objectives



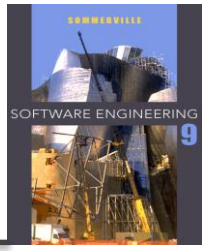
- understand **why** the architectural design of software is **important**;
- **understand the decisions that have to be made** about the system architecture during the architectural design process;
- have been introduced to **the idea of architectural patterns**, well-tried ways of organizing system architectures, which can be reused in system designs;
- **know the architectural patterns that are often used** in different types of application system, including transaction processing systems and language processing systems.

Software architecture



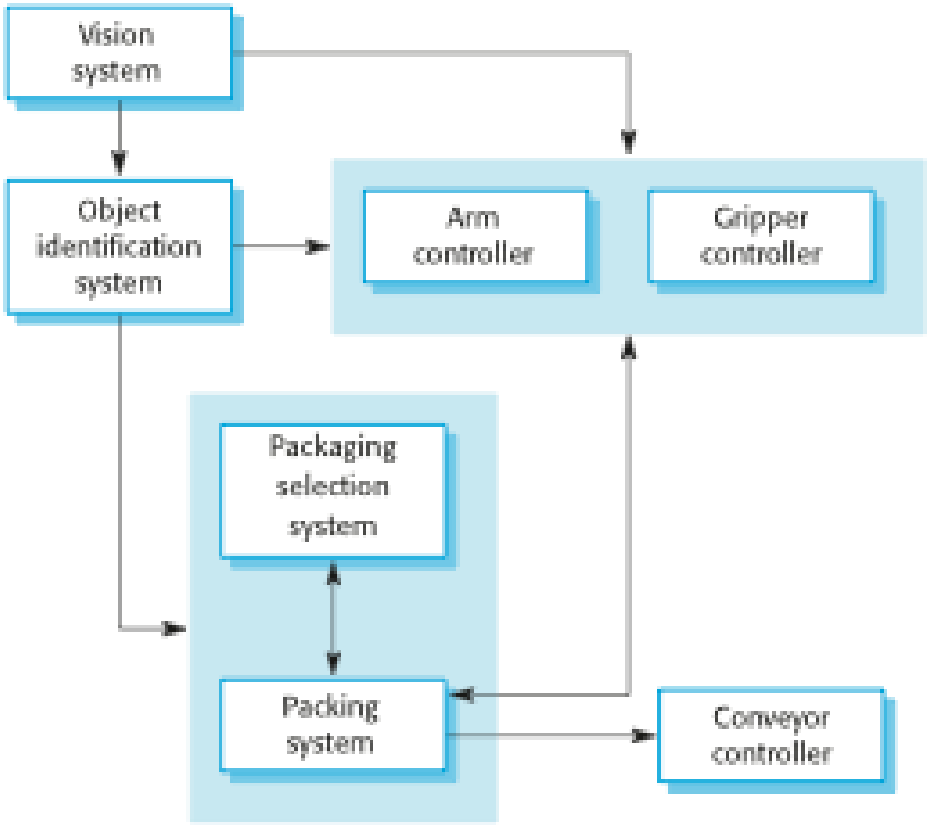
- The design process for identifying **the sub-systems** making up a system and the **framework for sub-system control and communication** is architectural design.
- The output of this design process is a description of the software architecture.

Architectural design

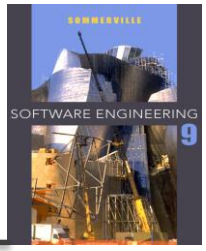


- **An early stage** of the system design process.
- Represents **the link between specification and design** processes.
- Often carried out **in parallel with some specification** activities.
- It involves **identifying major system components** and their **communications**.

The architecture of a packing robot control system

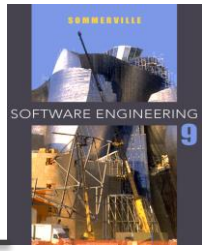


Architectural abstraction



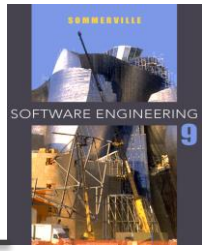
- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an **individual program is decomposed** into components.
- Architecture in the large
  - is concerned with the **architecture of complex enterprise systems** that include other systems, programs, and program components.
  - These enterprise systems are **distributed over different computers**, which may be owned and managed by different companies.

Why is Software architecture Important



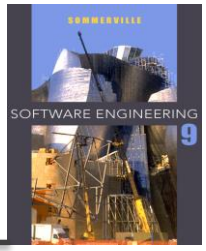
- It affects the **performance, robustness, distributability, and maintainability** of a system.
- Individual components implement **the functional system requirements.**
- The **non-functional** requirements depend on the system architecture—the way in which these components are **organized and communicate**.
- In many systems, non-functional requirements are **also influenced by individual components**, but there is no doubt that the **architecture of the system is the dominant influence.**

Advantages of explicit architecture

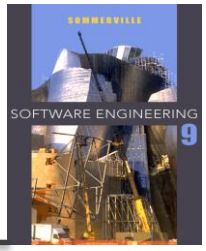


- ✧ Stakeholder communication
  - Architecture may be used as **a focus of discussion** by system stakeholders, for the negotiation of system requirements.
  - It is an essential **tool for complexity management**. **It hides details** and allows the designers to **focus on the key system abstractions.**
- ✧ System analysis
  - Means that analysis of whether the system can **meet its non-functional requirements is possible.**
  - Architectural design decisions have a **profound effect** on whether or not **the system can meet critical requirements** such as **performance, reliability, and maintainability**.
- ✧ Large-scale reuse
  - **The architecture may be reusable** across a range of systems with similar requirements and so **can support large-scale software reuse**

Architectural representations

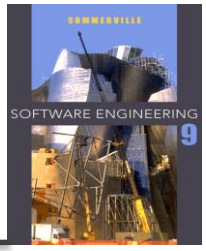


- Simple, informal **block diagrams showing entities and relationships** are the most frequently used method for documenting software architectures.
- Example the figure above.
- Each box in the diagram represents a component.
- **Boxes within boxes** indicate that the component has been decomposed to sub-components.
- **Arrows** mean that **data and or control signals** are passed from component to component in the direction of the arrows.



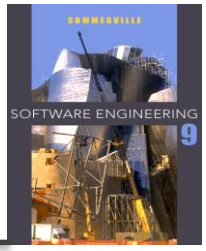
- +ve
  - present a **high-level picture** of the system structure, which people from different disciplines, who are involved in the system development process, **can readily understand.**
- -ve
  - But these have been criticized because they are **very Abstract**
  - **lack semantics**, do not show the **types of relationships** between entities nor the externally visible properties of entities in the architecture.

Use of architectural models



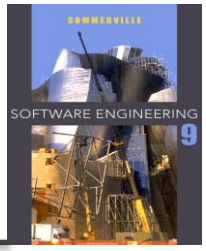
- **Depends on the use of architectural models**.
- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is **useful for communication with system stakeholders** and **project planning** because it **is not cluttered with detail**.
  - **Stakeholders can relate to it** and understand an abstract view of the system. They can then **discuss the system as a whole** without being confused by detail.
  - Block diagrams are good for this purpose
- As a way of documenting an architecture that has been designed
  - The aim here is to **produce a complete system model** that shows the different components in a system, **their interfaces and their connections**.
  - In this case it is **better to use a notation with well-defined semantics** for architectural description.

Architectural design decisions



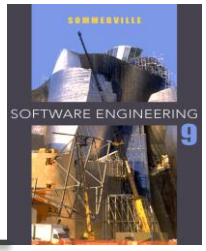
- Architectural design is a **creative process** so the process **differs depending on the type of system** being developed. the **background and experience** of the system architect, and the **specific requirements** for the system.
- However, **a number of common decisions** span all design processes
- and **these decisions affect the non-functional characteristics** of the system.
- System architects have to consider **the following fundamental questions** about the system:

Architectural design decisions



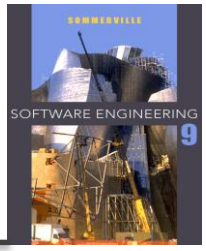
- Is there **a generic application architecture** that can be used?
  - When designing a system architecture, you have to decide **what your system and broader application classes have in common**, and decide **how much knowledge from these application architectures you can reuse**.
- How will the system be **distributed**?
  - The choice of distribution architecture is a key decision that **affects the performance and reliability of the system**
- What **architectural pattern styles** are appropriate?
  - An architectural pattern is **a description of a system organization** such as a **client–server organization** or a **layered architecture.**
  - **You should be aware of common patterns, where they can be used, and their strengths and weaknesses (section 6.3 discuss a number of frequently used patterns).**

- ✧ What approach will be used **to structure the system**?
- ✧ How will the system be **decomposed into modules**?
  - To decompose structural system units, you decide on the strategy for decomposing components into sub-components.
  - The approaches that you can use allow different types of architectures to be implemented.
- ✧ What **control strategy** should be used?
  - you make decisions about **how the execution of components is controlled**.
  - You develop a general model of **the control relationships** between the various parts of the system



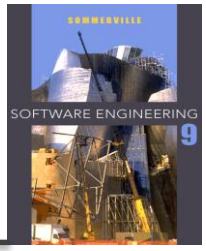
- **How** will the architectural design **be evaluated**?
  - Evaluating an architectural design is difficult because **the true test of an architecture** is how well the system meets its functional and non-functional requirements **when it is in use**.
  - However, you can do some evaluation **by comparing your design against reference architectures or generic architectural patterns**.
  - Or/And use Bosch's (2000) **description of the non-functional characteristics of architectural patterns**.
- **How** should the architecture be **documented**?
  - Section 6.2

Architecture reuse

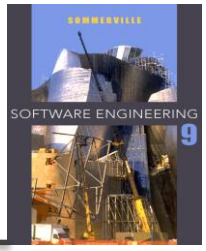


- Systems in the **same domain often have similar architectures** that reflect domain concepts.
- **Application product lines** are **built around a core architecture with variants** that satisfy particular customer requirements.
- The architecture of a system may be designed around **one of more architectural patterns or 'styles'**.
  - These **capture the essence of an architecture** and can be instantiated in different ways.
  - Discussed **later in this lecture**.

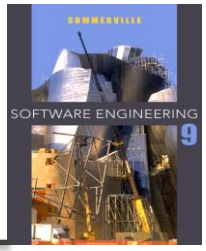
Architecture and system characteristics



- **The particular architectural style** and structure that you choose **should depend** on the **non-functional** system requirements:
- Performance
  - **Localize critical operations and minimize communications**..
  - This may **mean using a few relatively large components rather than small, fine-grain components**, which **reduces the number of component communications.**
- Security
  - **Use a layered architecture** with **critical assets in the inner layers**, with a **high level of security validation applied to these layers**



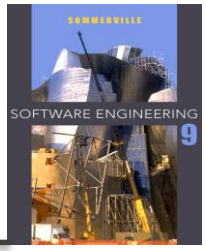
- Safety
  - **Localise safety-critical features** in a small number of sub-systems.
  - **This reduces the costs and problems of safety validation**
  - and makes it possible to provide related protection systems that can safely **shut down the system in the event of failure**.
- Availability
  - Include **redundant components and mechanisms** for fault tolerance, **so that it is possible to replace and update components without stopping the system. (see chapter 13)**



✧ Maintainability

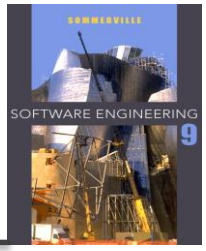
- If maintainability is a critical requirement, the system architecture should be designed using **fine-grain, self-contained components** that may readily be changed.
- **Producers of data should be separated from consumers**
- and **shared data structures should be avoided**.

A Compromise Might Be Needed

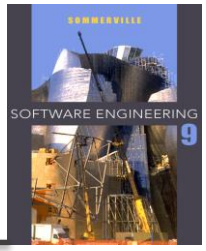


- Obviously there is potential conflict between some of these architectures.
- For example, **using large components improves performance** and **using small, fine-grain components improves maintainability**.
- **If both performance and maintainability** are important system requirements, then some compromise must be found.
- This can sometimes be achieved by **using different architectural patterns or styles for different** parts of the system.

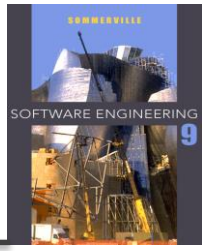
Architectural views (section 6.2)



- Architectural models of a software system can be used to
  - focus discussion about the software requirements or design.
  - document a design
- Two relevant issues
  - **What views or perspectives are useful** when designing and documenting a system's architecture?
  - **What notations** should be used for describing architectural models?

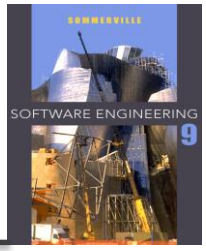


- **Each architectural** model only **shows one view** or perspective of the system.
- It might show
  - how a system is **decomposed into modules**,
  - how the **run-time processes inter**act
  - or the different ways in which system **components are distributed** across a network.
- For both design and documentation, **you usually need to present multiple views** of the software architecture.



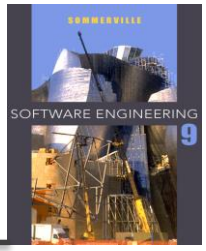
- Krutchen (1995), in **his well-known 4+1 view model** of software architecture,
- **suggests that there should be four fundamental architectural views**, which are **related using use cases or scenarios.**
- The views that he suggests are

4 + 1 view model of software architecture

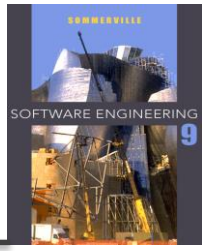


- A logical view, which **shows the key abstractions** in the system as objects or object classes.
  - **It should be possible to relate the system requirements to entities in this logical view.**
- A process view, which **shows how, at run-time, the system is composed of interacting processes**.
- A development view, which **shows how the software is decomposed for development.**
- A physical view, which **shows the system hardware and how software components are distributed across** the processors in the system.
  - This view is **useful for systems engineers** planning a system deployment.

Use UML?

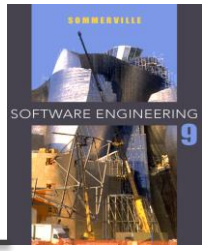


- There are **differing views** about whether or not software architects should use the UML for architectural description (Clements, et al., 2002).
- A survey in 2006 (Lange et al., 2006) showed that, when the UML was used, it was mostly **applied in a loose and informal way.**
- The authors of **that paper argued** that this was **a bad thing**.
- **The UML was designed for describing object-oriented** systems and, at the architectural design stage, you often **want to describe** systems at a **higher level of abstraction**.
- **Object classes are too close to the implementation** to be useful for architectural description.

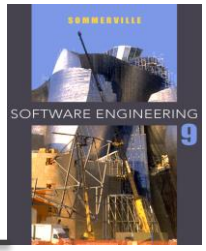


- I (the author) don't find the UML to be useful during the design process itself and prefer
- **informal notations that are quicker to write and which can be easily drawn on a white-board**.
- The UML is of most value when you are **documenting an architecture in detail** or using model-driven development (see Chapter 5).

Architectural patterns

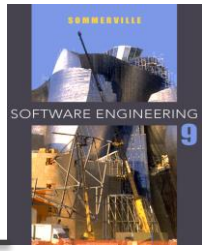


- Patterns are a **means of representing, sharing and reusing knowledge** about software systems.
- An architectural pattern is **a stylized description of good design practice,** which has been **tried and tested** in different environments.
- Patterns should **include information** about **when they are and when the are not useful.**
- Patterns may be represented using **tabular and graphical descriptions**.



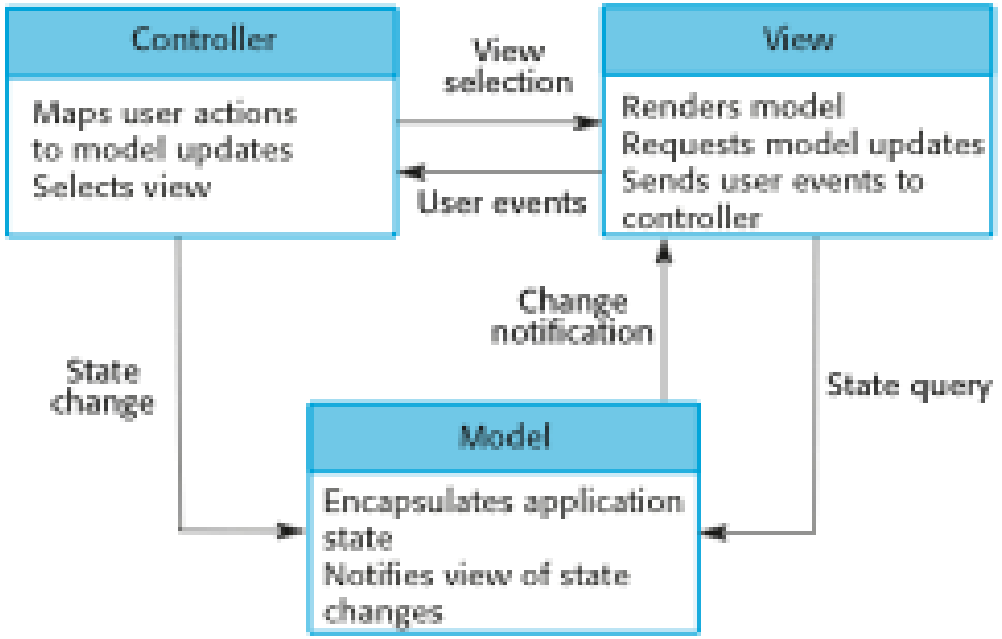
- **For example**, the following Figures describe the well-known **Model-View-Controller pattern**.
- This pattern is **the basis of interaction management in many web-based systems**.
- The stylized **pattern description includes**
  - **the pattern name**,
  - **a brief description** (with an associated graphical model),
  - and **an example of the type of system** where the pattern is used (again, perhaps with a graphical model).
- You should also include information about **when the pattern should be used** and its **advantages and disadvantages**.

The Model-View-Controller (MVC) pattern



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.

The organization of the Model-View-Controller



Web application architecture using the MVC pattern

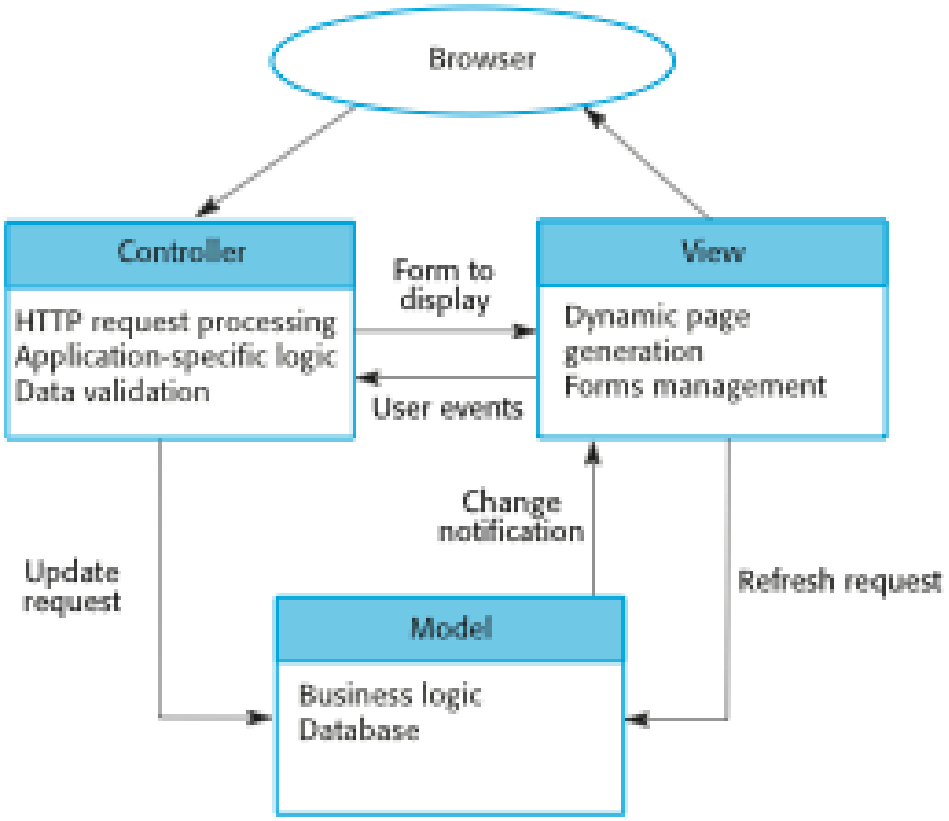
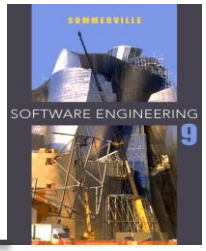
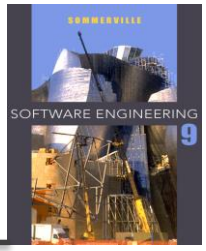


Figure 6.4 shows a possible run-time architecture when this pattern is used for interaction management in a web-based system.



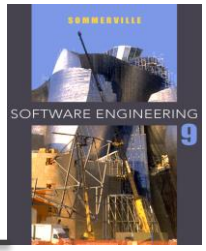
- It is **impossible to describe all of the generic patterns** that can be used in software development.
- We will discuss **some selected examples of patterns** that are **widely used** and which capture good architectural design principles.
- You can find **further examples** of generic architectural patterns on the **book's web pages**.

The Layered architecture pattern



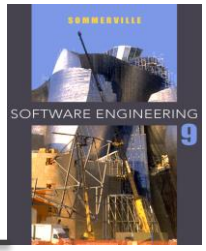
- This layered approach **supports the incremental development** of systems.
- As a layer is developed, **some of the services provided by that layer may be made available to users**.
- The architecture is also **changeable and portable**.
  - **So long as its interface is unchanged, a layer can be replaced by another, equivalent layer.**
- Furthermore, **when layer interfaces change** or new facilities are added to a layer, **only the adjacent layer is affected**.
- As layered systems **localize machine dependencies in inner layers**,
  - this makes it **easier to provide multi-platform implementations** of an application system.
  - **Only the inner, machine-dependent layers need be re-implemented** to take account of the facilities of a **different operating system or database**.

The Layered architecture pattern



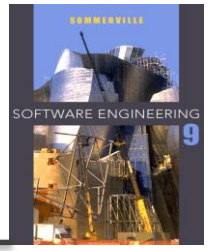
Name	Layered architecture
Description	**Organizes the system into layers** with related functionality associated with each layer. A **layer provides services to the layer above** it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	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.

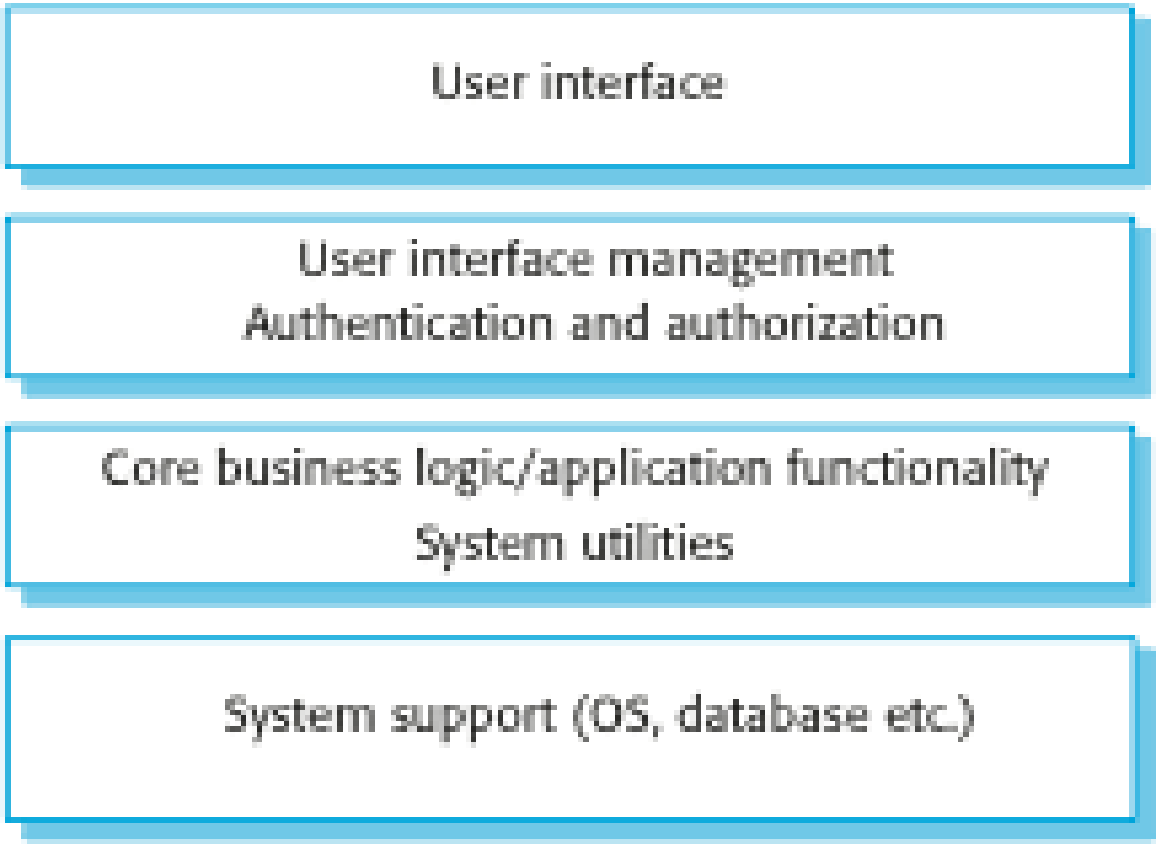
Layered architecture



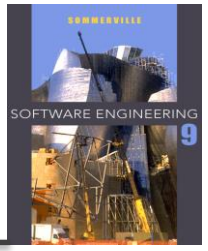
- Used to **model the interfacing** of sub-systems.
- Organises the system into **a set of layers** (or abstract machines) **each of which provide a set of services**.
- **Supports the incremental development** of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, **often artificial to structure systems** in this way.

A generic layered architecture





The above Figure



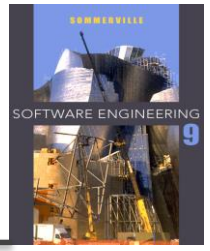
✧ is an example of a layered architecture with four layers.

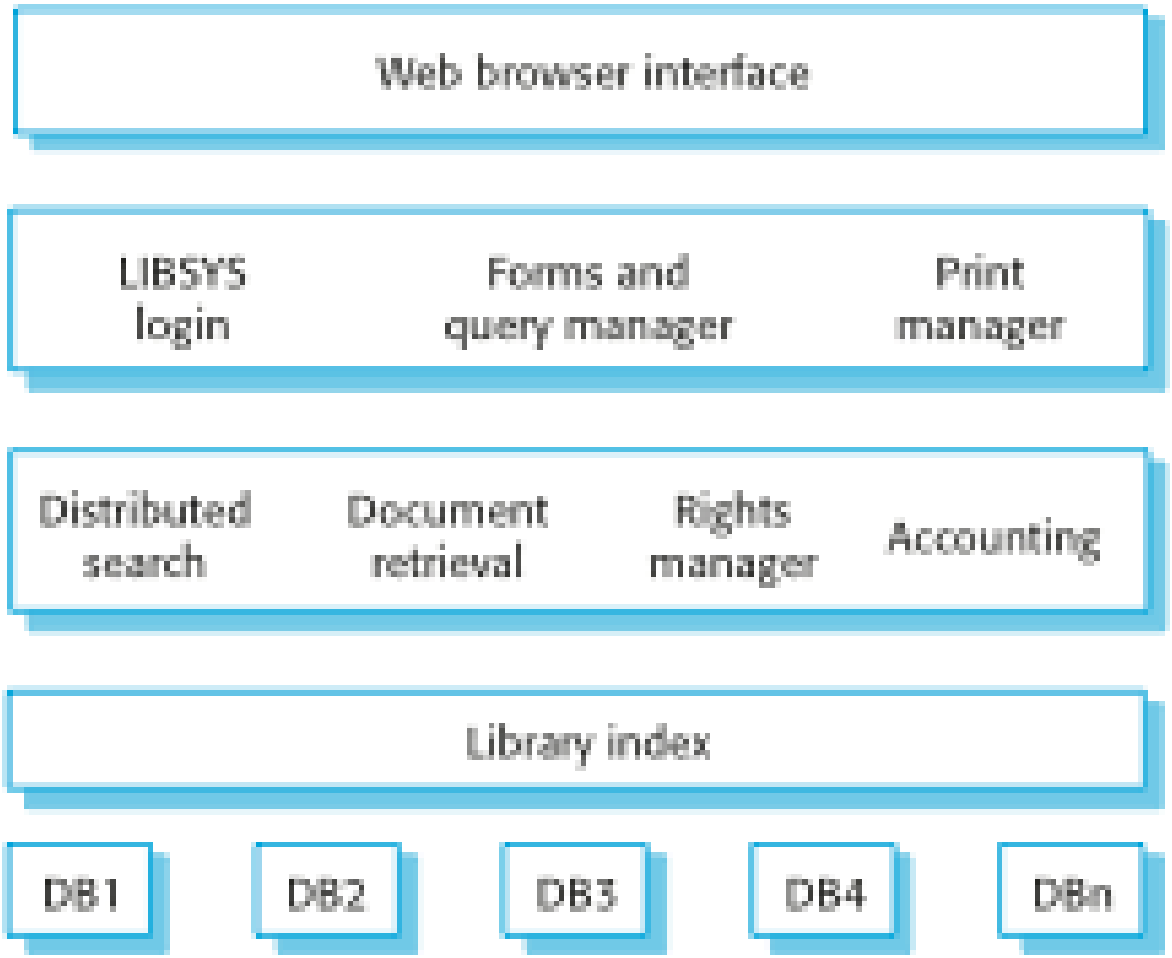
1. **The lowest layer includes system support software**—typically database and operating system support.
2. The next layer is the **application layer** that includes the components concerned with the application functionality and utility components that are used by other application components.
3. The third layer is concerned with **user interface management and providing user authentication and authorization**,
4. the top layer providing **user interface facilities**.

✧ Of course, **the number of layers is arbitrary**.

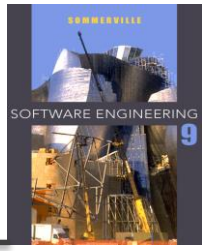
✧ Any of the layers in the Figure could be split into two or more layers.

The architecture of the LIBSYS system (Another Example of Layered Architecture pattern when applied to a library system)



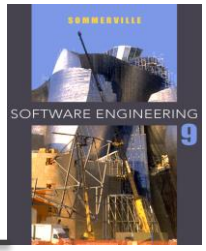


Other Patterns



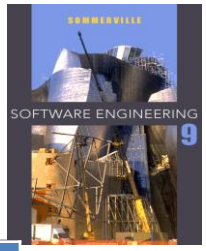
- **Repository pattern** (Figure 6.8), describes how a set of interacting components can share data.
- The **Client–server pattern** is a very commonly used run-time organization for **distributed systems**.
- The final example of an architectural pattern is the **pipe and filter patte**rn.
  - This is a model of the **run-time organization of a system** where functional transformations process their inputs and produce outputs

Repository architecture



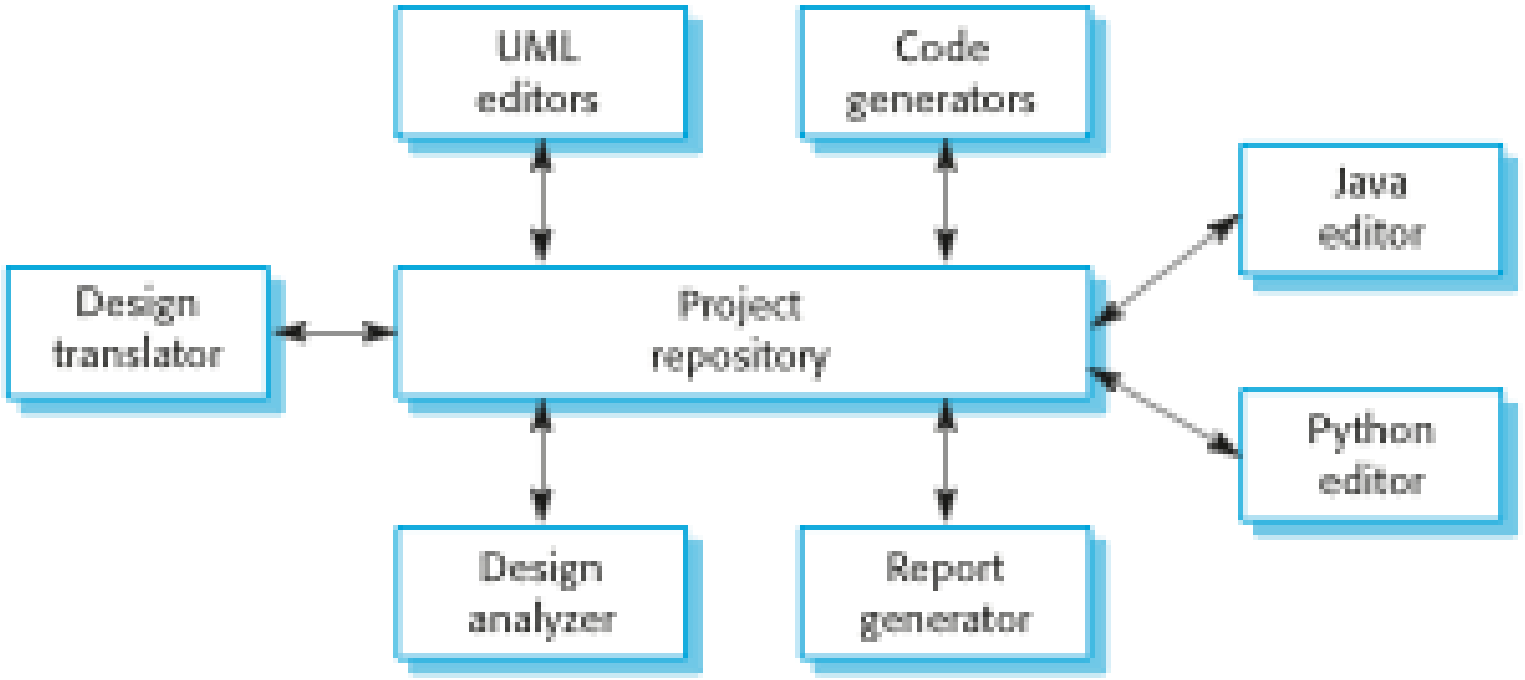
- **Sub-systems must exchange data**. This may be done in two ways:
  - **Shared data is held in a central database or repository** and may be accessed by all sub-systems;
  - **Or each sub-system maintains its own database and passes data** explicitly to other sub-systems.
- When **large amounts of data** are to be shared, the **repository model of sharing is most commonly** used as this is an **efficient data sharing** mechanism.

The Repository pattern

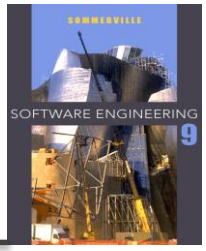


Name	Repository
Description	All data in a system is managed in **a central repository** that is accessible to all system components. **Components do not interact directly,** only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design 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 has 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.**

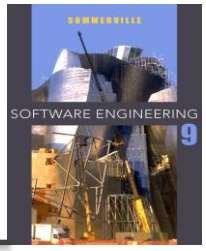
A repository architecture for an IDE



Client-server architecture

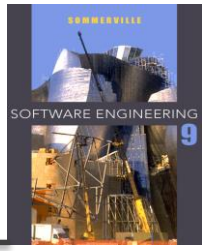


- **Distributed system model** which shows **how data and processing is distributed across a range of components**.
  - **Can be implemented on a single computer**.
- **Set of stand-alone servers** which p**rovide specific services such as printing, data management, et**c.
- **Set of clients which call on these services**.
- **Network w**hich allows clients to access servers.



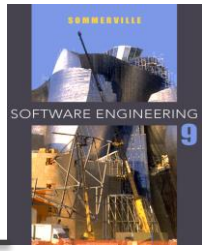
- An important **benefit is separation and independence**.
- Services and **servers can be changed without affecting other parts of the system**.
- **Clients may have to know the names** of the available servers and the services that they provide.
- However, **servers do not need to know the identity of clients or how many clients are accessing** their services.
- **Effective use can be made of networked systems** with many distributed processors.
- It is **easy to add a new server and integrate** it with the rest of the system.

The Client–server pattern



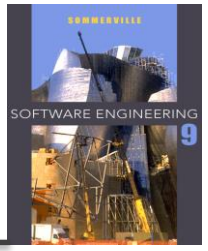
Name	Client-server
Description	In a client–server architecture, the **functionality** of the system is **organized into services**, with each service delivered from a **separate server**. **Clients are users of these services** and access servers to make use of them.
Example	Figure 6.11 is an example 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.**

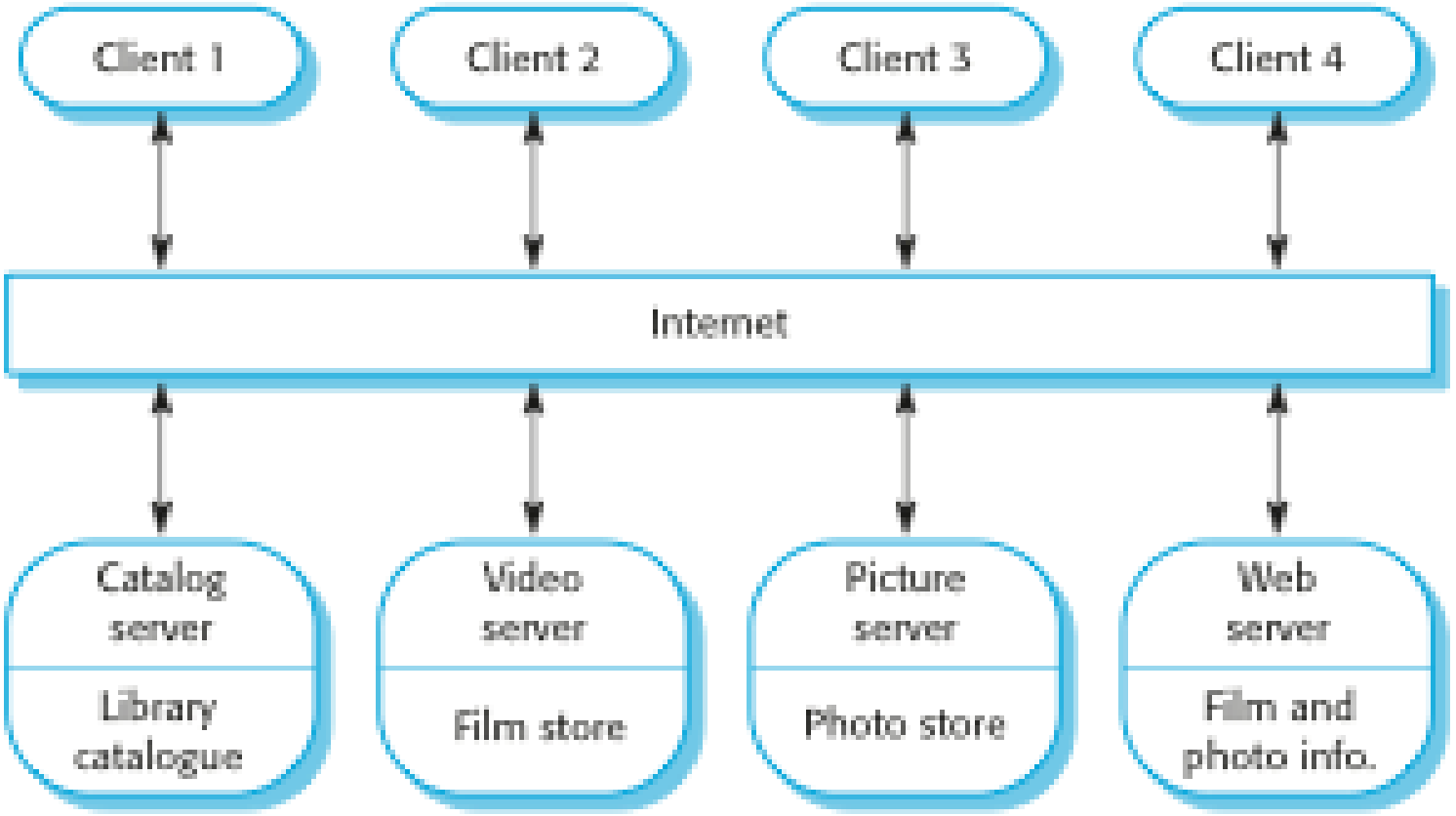
An example of a system that is based on the client–server model.



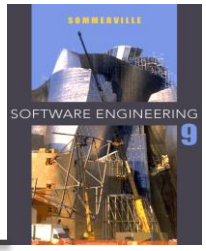
- This is a multi-user, web-based system for providing a film and photograph library.
- In this system, several servers manage and display the different types of media.
- **Video frames need to be transmitted quickly** and in synchrony but at **relatively low resolution**.
- They may be **compressed in a store**, so the video server can handle video compression and decompression in different formats.
- **Still pictures however**, must be **maintained at a high resolution**, so it is appropriate to maintain them on a separate server.

A client–server architecture for a film library



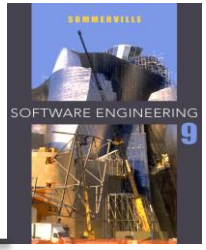


Pipe and filter architecture

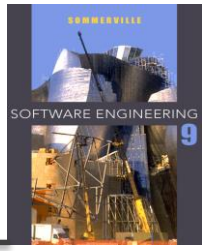


- Functional transformations **process their inputs to produce outputs**.
- May be referred to as a **pipe and filter model** (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is **extensively used in data processing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

The pipe and filter pattern

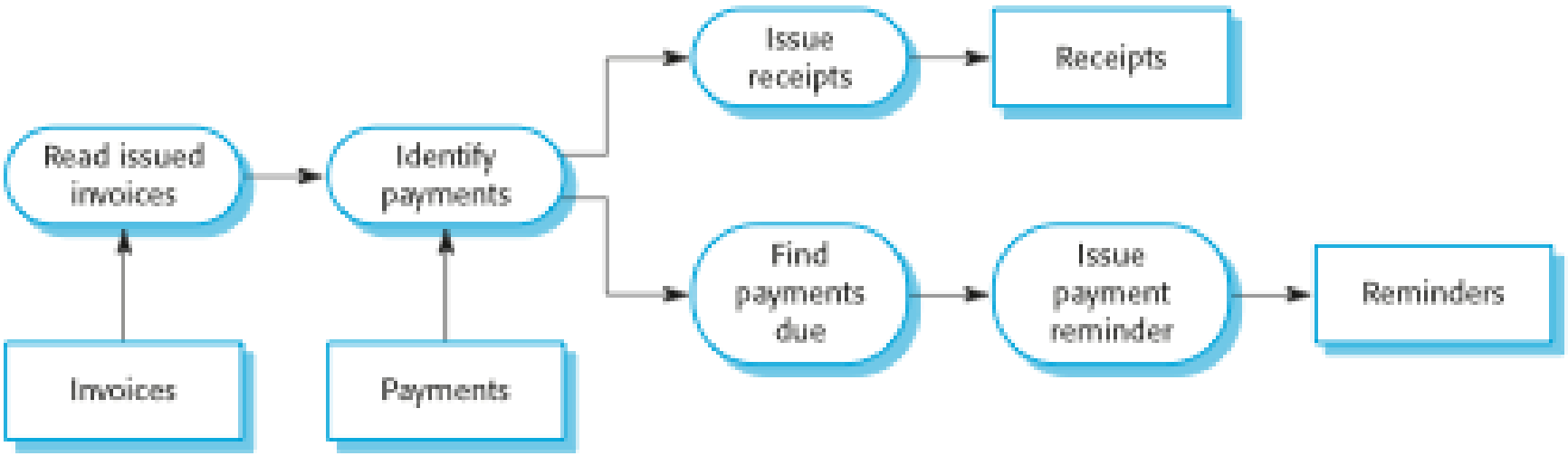


Name	Pipe and filter
Description	The processing of the data in a system is organized so that **each processing component** (filter) is discrete and **carries out one type of data transformation**. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example 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 unparse its output** to the agreed form. **This increases system overhead** and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

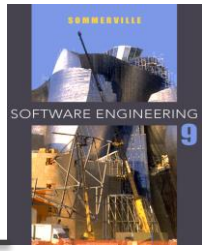


- An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.13. An organization has issued invoices to customers.
- Once a week, **payments that have been made are reconciled with the invoices**.
- For those invoices that have been paid, **a receipt is issued**.
- For those invoices that **have not been paid** within the allowed payment time, **a reminder is issued**.

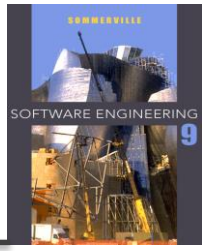
An example of the pipe and filter architecture



Key points



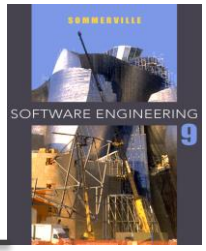
- A software architecture is a description of how a software system is organized.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.



Chapter 6 – Architectural Design

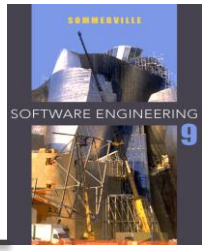
Lecture 2

Application architectures (section 6.4)



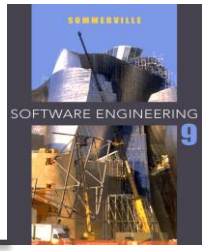
- Application systems are designed to meet an organizational need.
- As **businesses have much in common**, their application systems also **tend to have a common architecture** that reflects the application requirements.
- A **generic application architecture** is an architecture for a type of software system that **may be configured and adapted to create a system that meets specific requirements**.

Use of application architectures



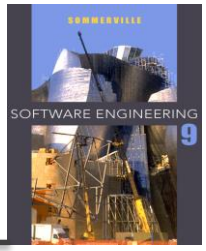
- As **a starting point** for architectural design.
- As a **design checklist**.
- As a way of **organizing the work** of the development team.
- As a means of **assessing components for reuse**.
- As a **vocabulary for talking** about application types.

Examples of application types



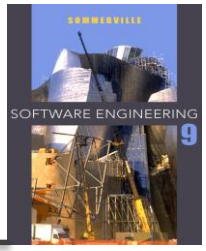
- ✧ Data processing applications
  - Data driven applications that **process data in batches** without explicit user intervention during the processing.
- ✧ Transaction processing applications
  - Data-centred applications **that process user requests and update information** in a system database.
- ✧ Event processing systems
  - Applications where system **actions depend on interpreting events** from the system's environment.
- ✧ Language processing systems
  - Applications where the users**' intentions are specified in a formal language** that is processed and interpreted by the system.

Application type examples



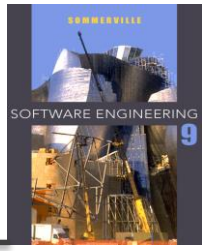
- **Focus here is on transaction processing and language processing systems**.
- Transaction processing systems
  - E-commerce systems;
  - Reservation systems.
- Language processing systems
  - Compilers;
  - Command interpreters.

Transaction processing systems



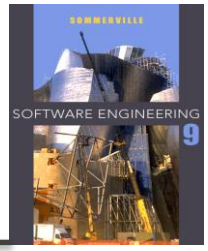
- Technically, a database transaction is sequence of operations that is **treated as a single unit** (**an atomic unit**).
- **All of the operations** in a transaction **have to be completed before the database changes are made permanent.**
- This ensures that **failure of operations** within the transaction **does not lead to inconsistencies** in the database.

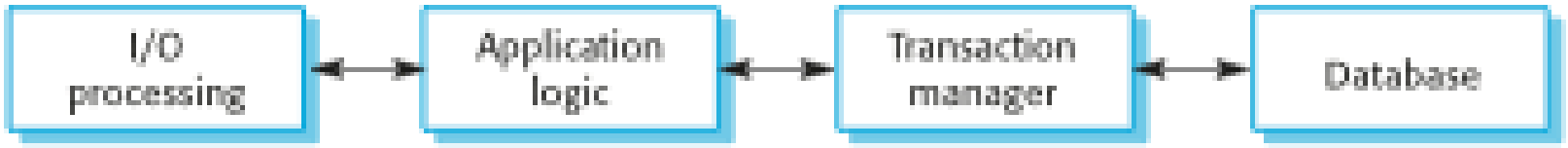
Transaction processing systems



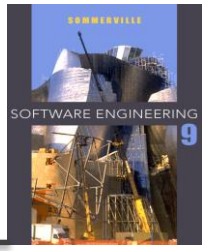
- Process user **requests for information** from a database **or requests to update** the database.
- From a user perspective a transaction is:
  - Any **coherent sequence** of operations that **satisfies a goal**;
  - For example - **find the times** of flights from London to Paris.
- Users make **asynchronous requests** for service which are then processed by **a transaction manager**.

The conceptual architectural structure of transaction processing



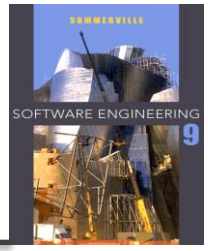


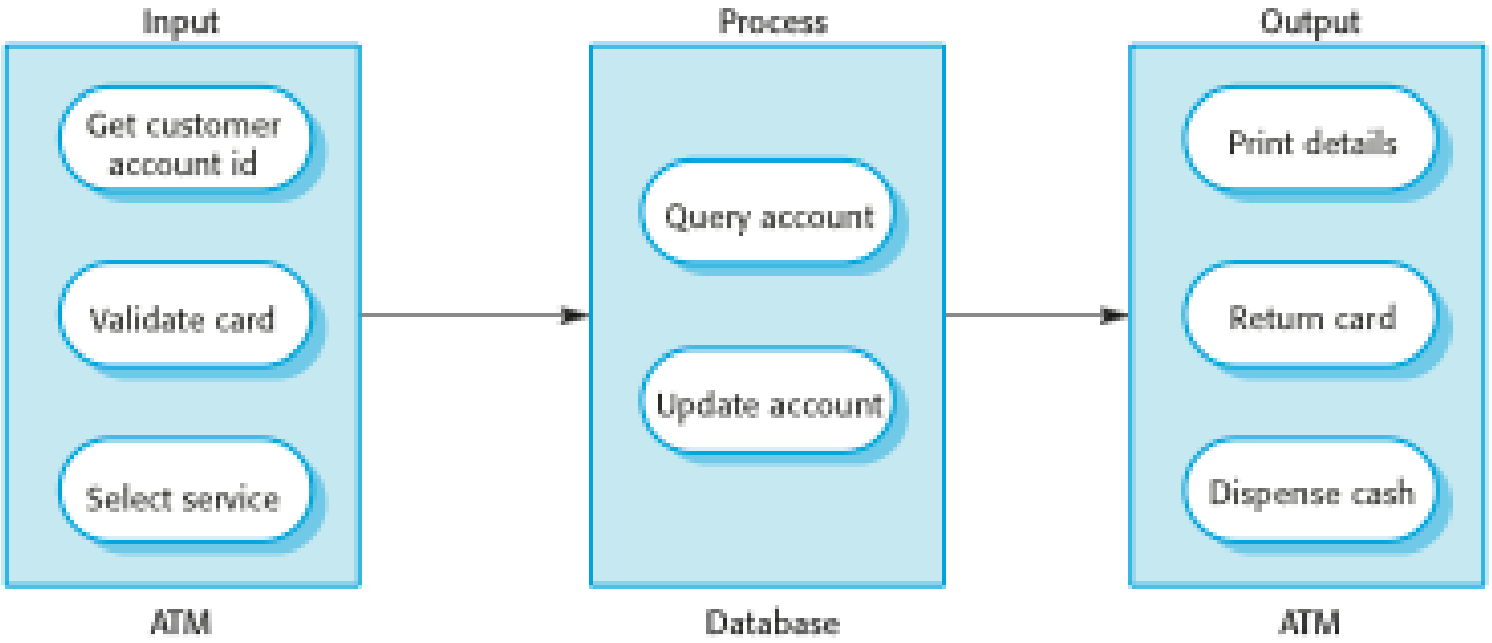
‘pipe and filter’ Architecture



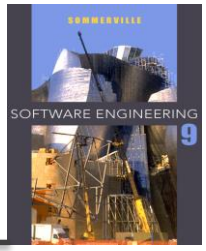
- Transaction processing systems may be organized as a ‘pipe and filter’ architec-ture with system components responsible for input, processing, and output.
- For example, **consider an ATM**.
  - The system is composed of **two cooperating soft-ware components—the ATM software** and **the account processing software** in the bank’s database server.
  - The **input and output components** are implemented as **software in the ATM** and **the processing component is part of the bank’s** database server.

The software architecture of an ATM system: A ‘pipe and filter’ architecture



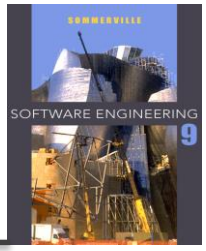


Information systems



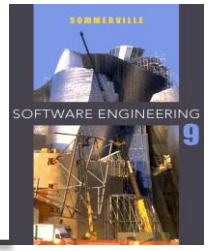
- All systems that involve interaction with **a shared database** can be considered to be **transaction-based information systems**.
- An information system **allows controlled access to a large base of information**, such as
  - a library catalog,
  - a flight timetable,
  - or the records of patients in a hospital.
- Increasingly , information systems **are web-based systems** that are **accessed through a web browser**.

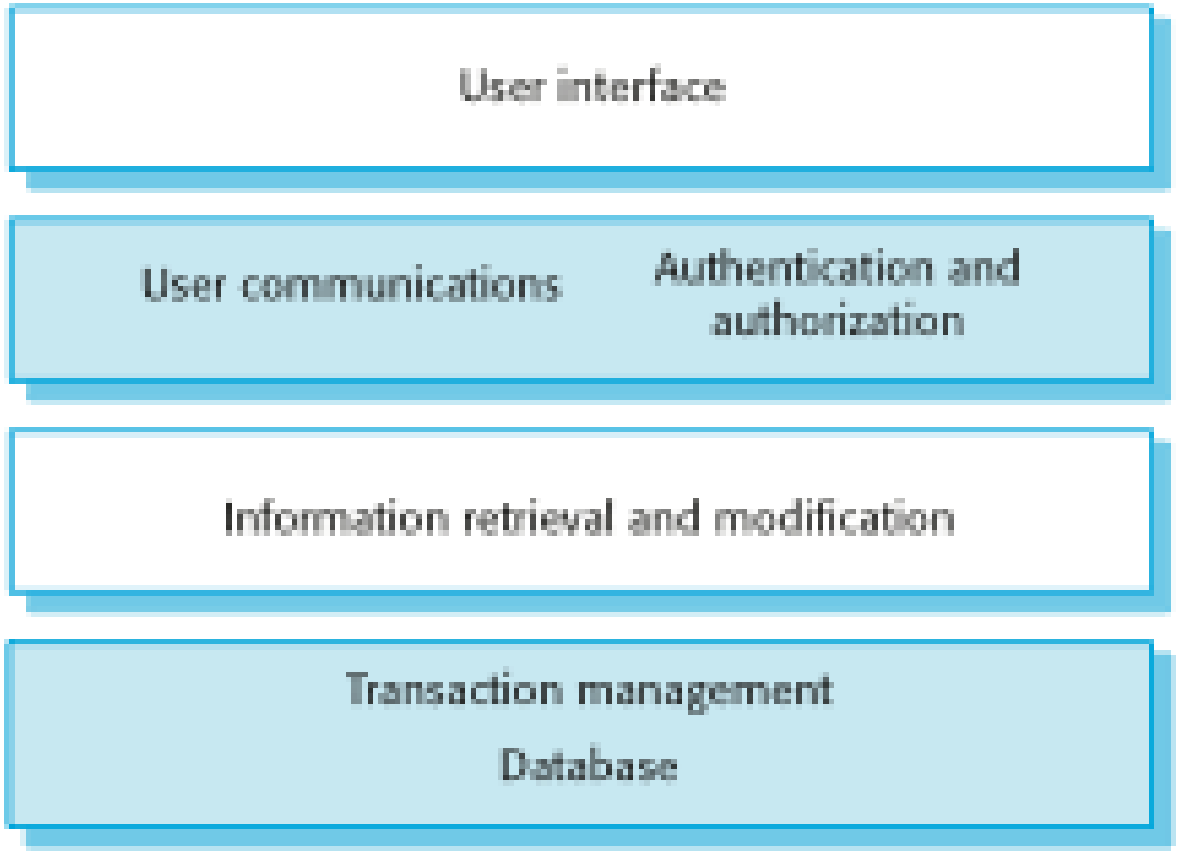
Information systems architecture



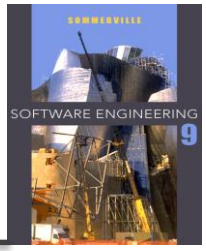
- Information systems **have a generic architecture** that can be **organized as a layered architecture**.
- These are **transaction-based systems** as interaction with these systems generally **involves database transactions.**
- Layers include:
  - The user interface
  - User communications
  - Information retrieval
  - System database

Layered information system architecture





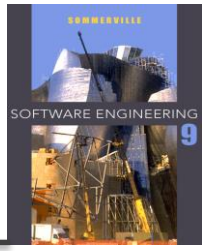
Layered information system architecture

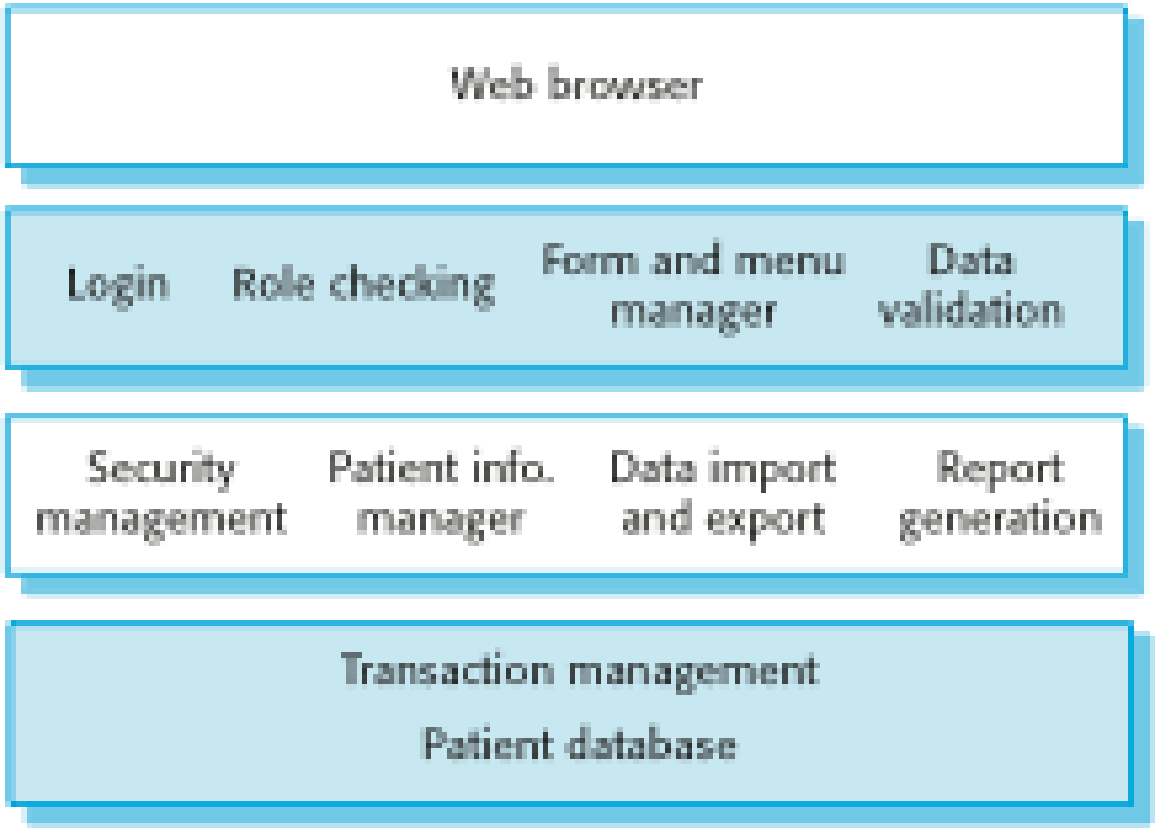


1. **The top layer** is responsible for **implementing the user interface**. In this case, the UI has been implemented using a web browser.
2. **The second layer provides the user interface functionality that is delivered through the web browser**.

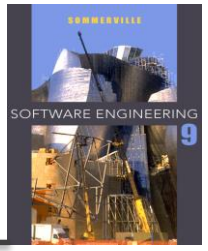
   It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
3. **The third layer implements the functionality of the system** and provides components that implement **system security, patient information creation and updating, import and export of patient data** from other databases, and report generators that create management reports.

The architecture of the MHC-PMS



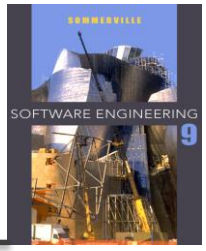


Web-based information systems



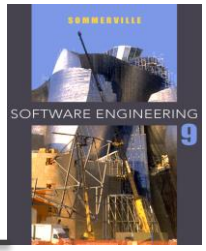
- Information and resource management systems are now usually web-based systems where the **user interfaces are implemented using a web browser**.
- For example, **e-commerce** systems are
  - **Internet-based** resource management systems
  - that **accept electronic orders** for goods or services and then
  - **arrange delivery** of these goods or services to the customer.
- In an e-commerce system, the **application-specific layer** includes **additional functionality supporting a 'shopping cart'** in which users can place a number of items in separate transactions, then **pay for them all together in a single transaction.**

Server implementation



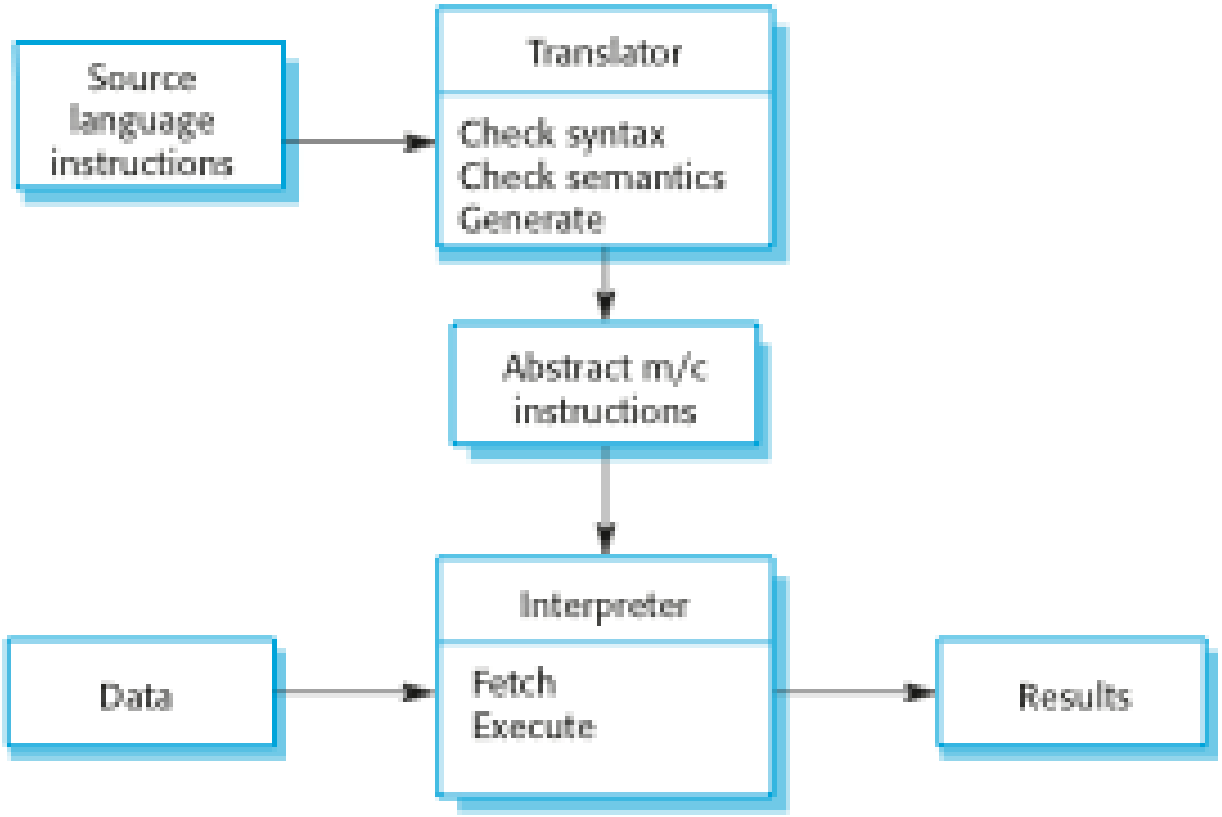
- These systems are often implemented **as multi-tier client server/architectures** (discussed in Chapter 18)
  - **The web server** is responsible for **all user communications**, with the user interface implemented using a web browser;
  - **The application server** is responsible for **implementing application-specific logic** as well as information storage and retrieval requests;
  - **The database server** moves information to and from the database and **handles transaction management**.
- Using **multiple servers** allows **high throughput** and makes it **possible to handle hundreds of transactions** per minute.
- **As demand increases**, servers can be added at each level to **cope with the extra processing involved**.

Language processing systems

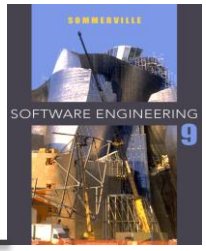


- **Accept a natural or artificial language** as input and
- **generate some other representation** of that language.
- May **include an interpreter** to act on the instructions in the language that is being processed.
- Used in situations where the easiest way to solve a problem is to **describe an algorithm** or **describe the system data**
  - **Meta-case tools** process tool descriptions, **method rules**, etc and **generate tools**.

The architecture of a language processing system

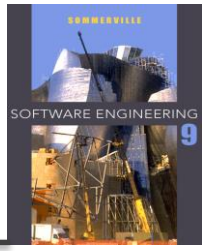


Compiler components

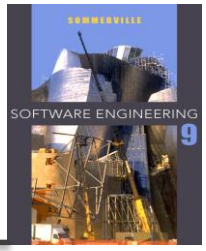


- A lexical analyzer, which takes input language tokens and converts them to an internal form.
- A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- A syntax analyzer, which checks the syntax of the language being translated.
- A syntax tree, which is an internal structure representing the program being compiled.

Compiler components

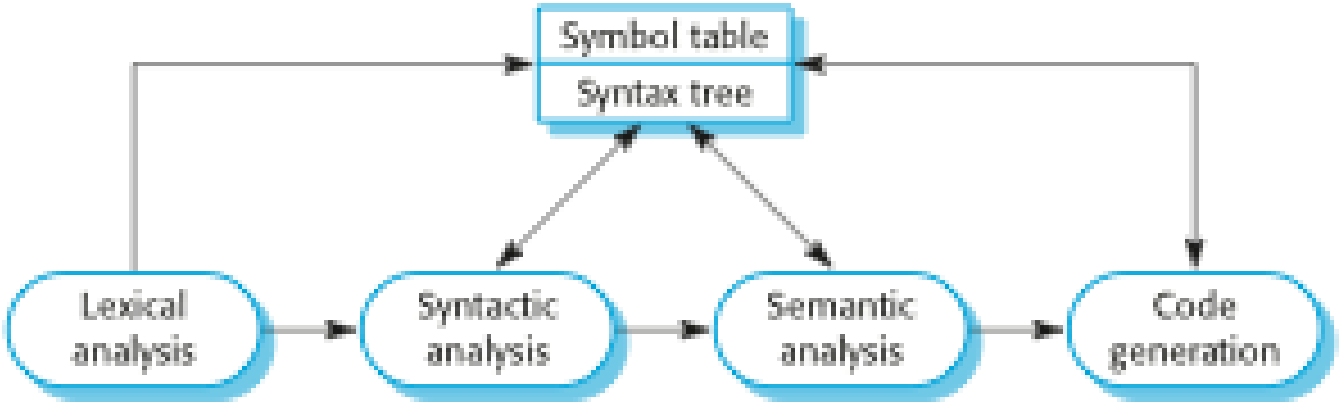


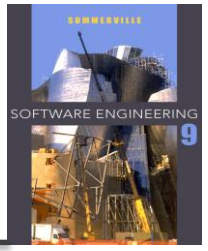
- A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- A code generator that ‘walks’ the syntax tree and generates abstract machine code.



- Compilers can be implemented using **a composite of a repository and a pipe and filter model**.
- In a compiler architecture, the symbol table is a repository for shared data.
- The phases of lexical, syntactic, and semantic analysis are organized sequentially, as shown the Figure and communicate through the shared symbol table.

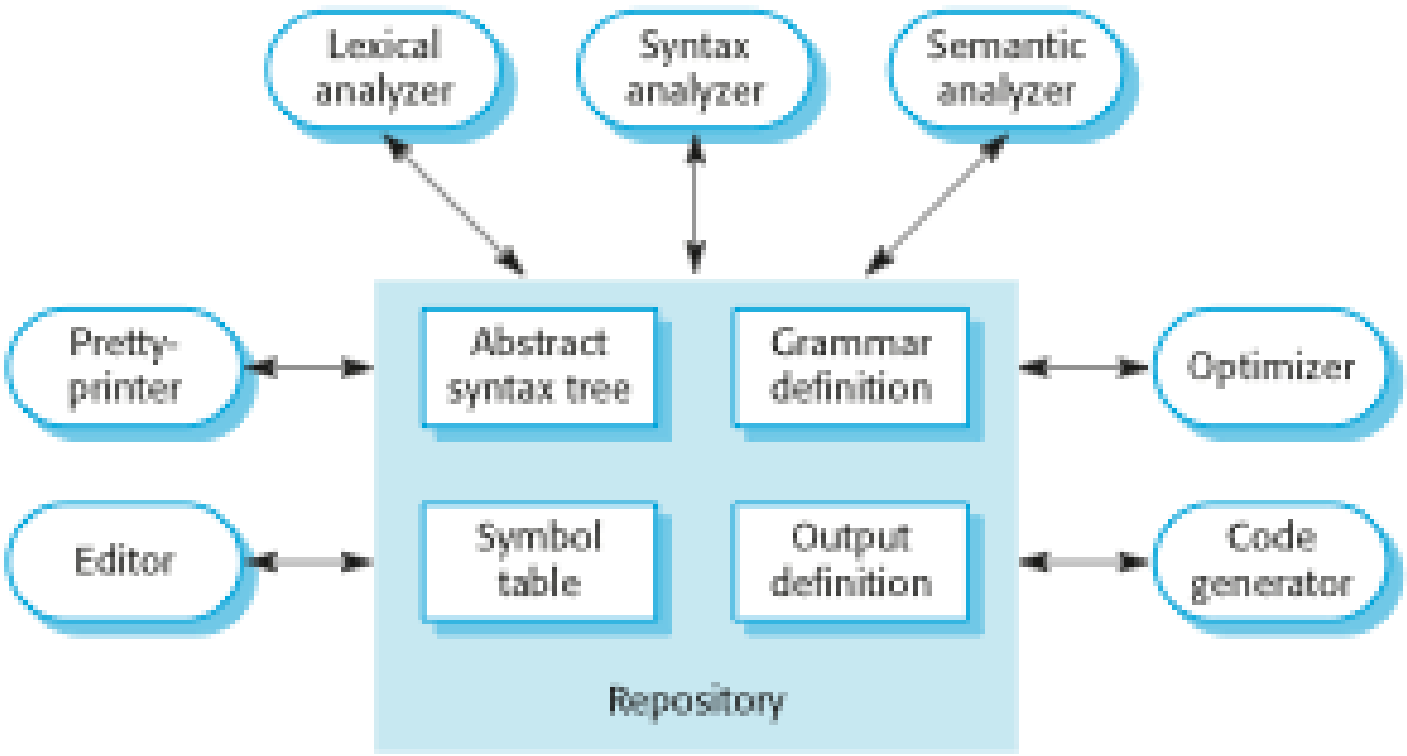
A pipe and filter compiler architecture



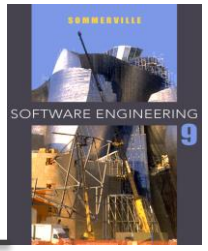


- **This pipe and filter** model of language compilation is **effective in batch environments** where programs are compiled and executed **without user interaction**;
- It is **less effective when a compiler is integrated with other language processing tools** such as a structured editing system, an interactive debugger.
- In this situation, **changes from one component need to be reflected immediately** in other components.
- It is better, therefore, to organize the system around a repository, as shown in Figure 6.20.

A repository architecture for a language processing system



Key points



- Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.