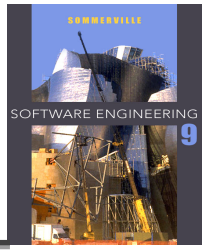
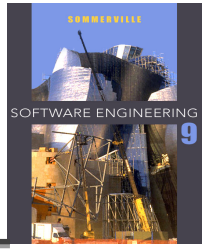


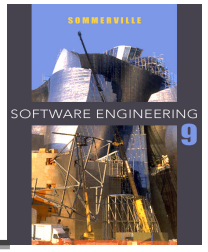
Software Architecture

Based on
Sommerville:
Software Engineering 9,
Chapter 6 – Architectural Design

The software process



CO520: Programming in the Large



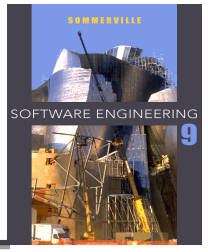
Designing Classes

- responsibility-driven design
- coupling (loose between classes)
- cohesion (high within a class)
- refactoring

Focus was on a couple of classes.

Now: a really big program or whole enterprise system

Software architecture

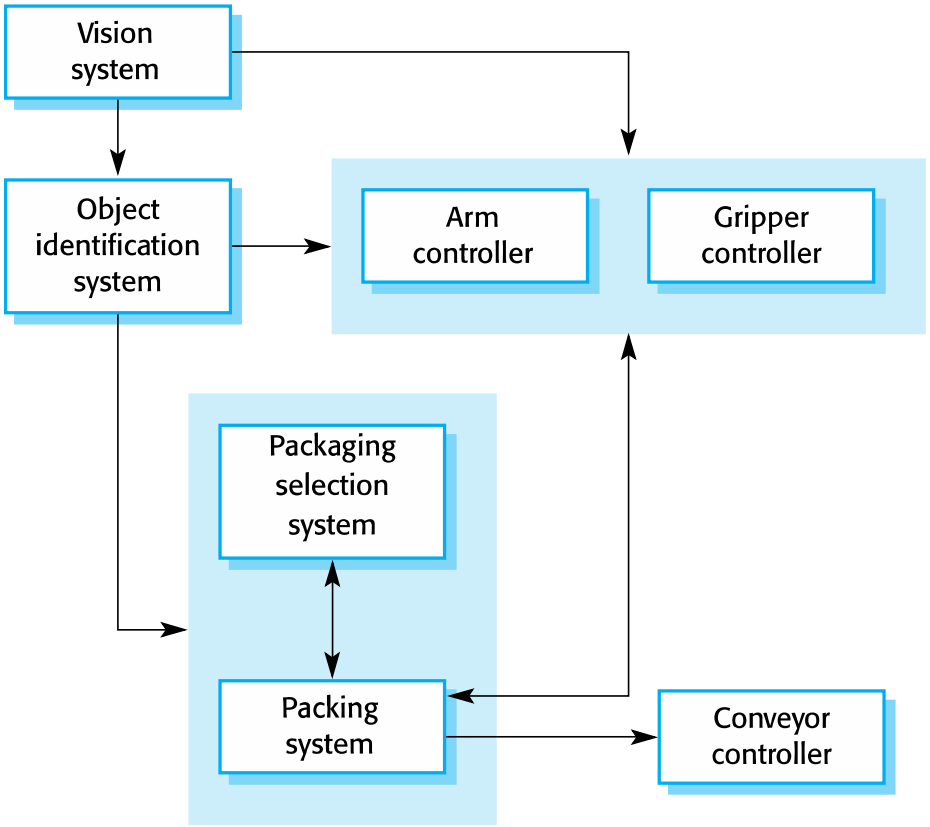


Software Architecture is the high-level structure of a software system, the discipline of creating such a high level structure and the documentation of this structure.

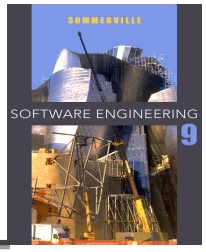
Wikipedia

- identifies the sub-systems making up a system
- a framework for sub-system control and communication

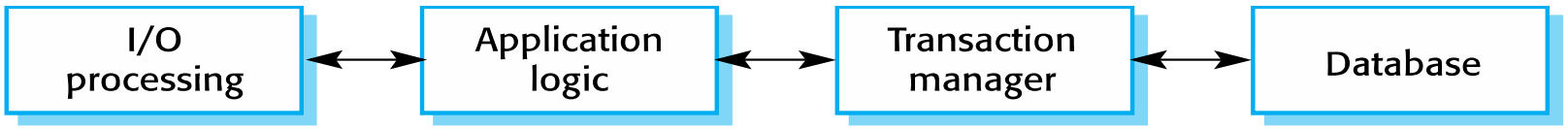
Example: The architecture of a packing robot control system



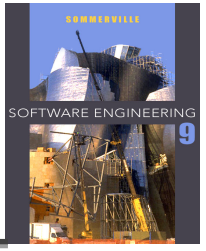
Architecture models: Box and line diagrams



- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.

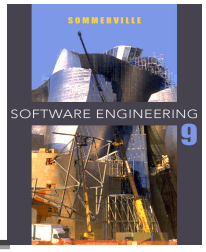


Use of architectural models



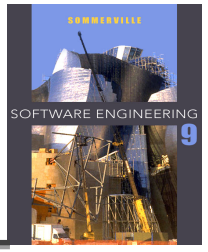
- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system **stakeholders** and **project planning** because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- 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.

Advantages of explicit architecture



- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

Architectural design decisions

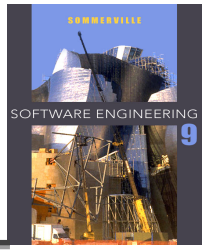


- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Note:

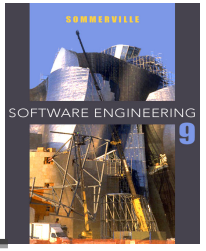
- a **functional requirement** describes input, behaviour and output; what a system is supposed to do

Architecture and non-functional requirements



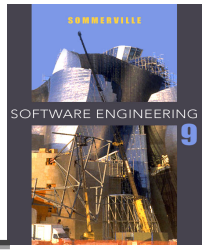
- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of sub-systems.
- Availability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

Architecture reuse



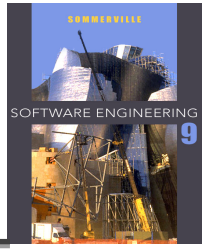
- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- The architecture of a system may be designed around one of more architectural patterns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

Architectural patterns



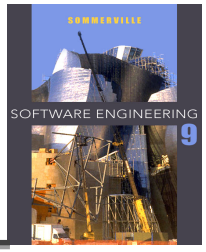
- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns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

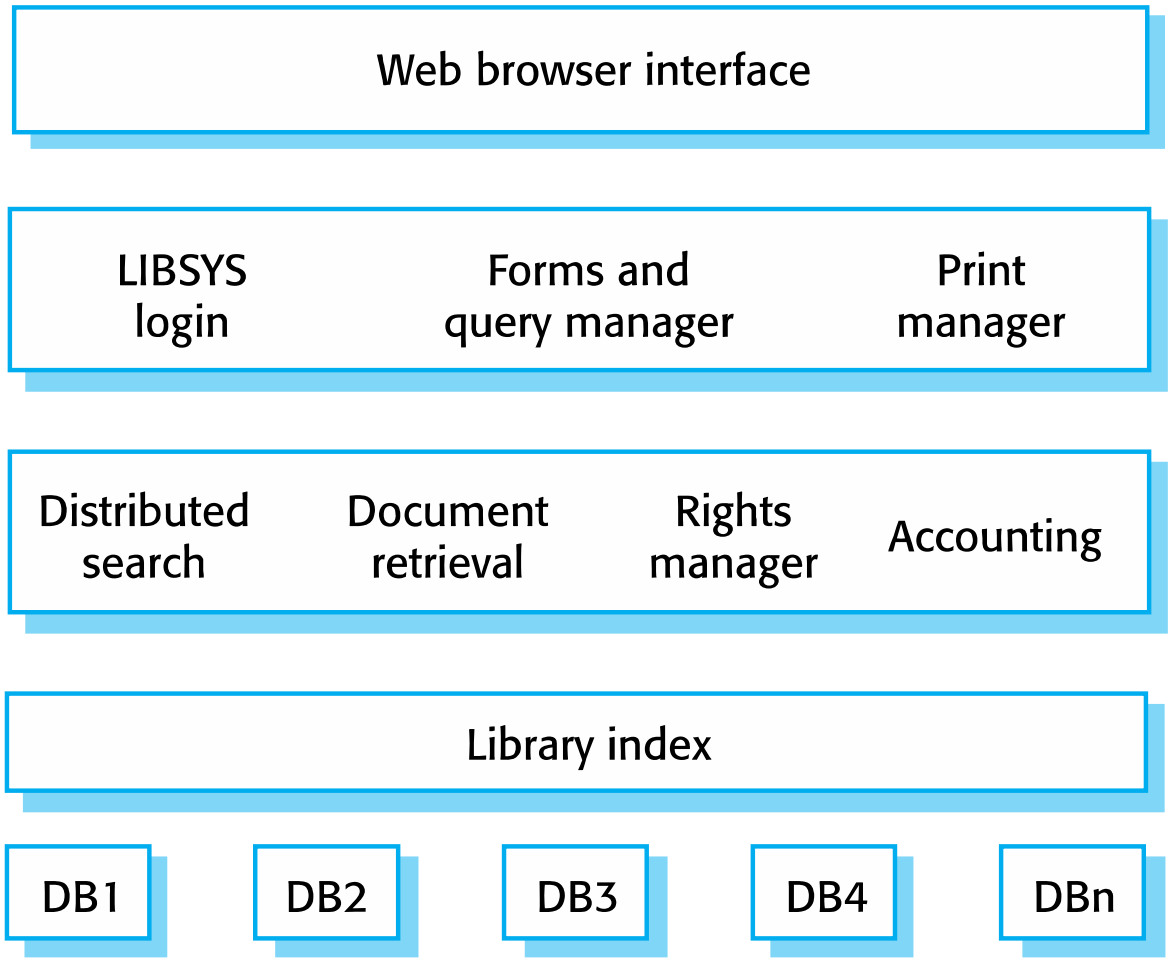
Some widely used patterns



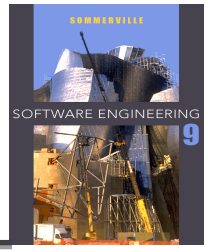
- layered architecture
- repository architecture
- client-server architecture
- pipe and filter architecture

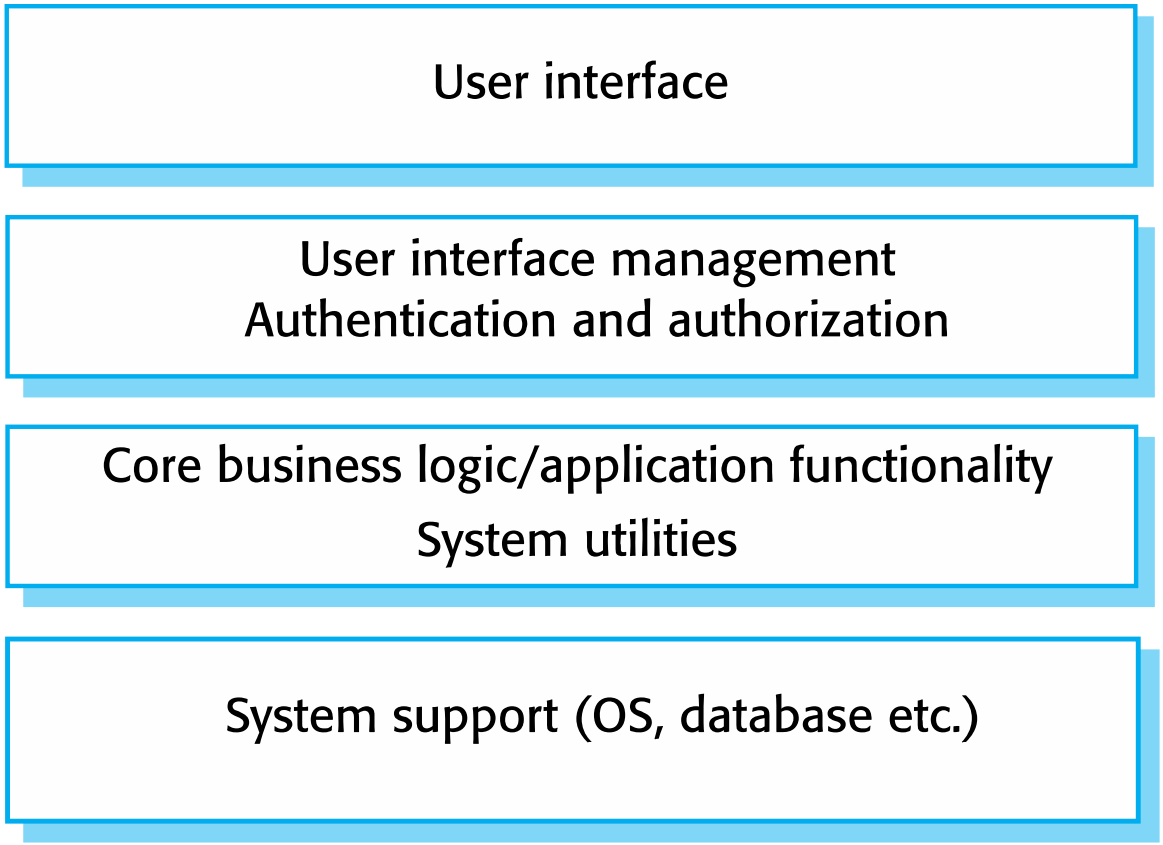
The architecture of the LIBSYS system



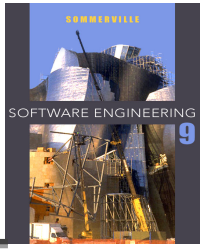


A generic layered architecture



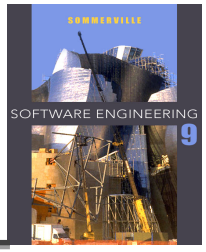


Layered architecture



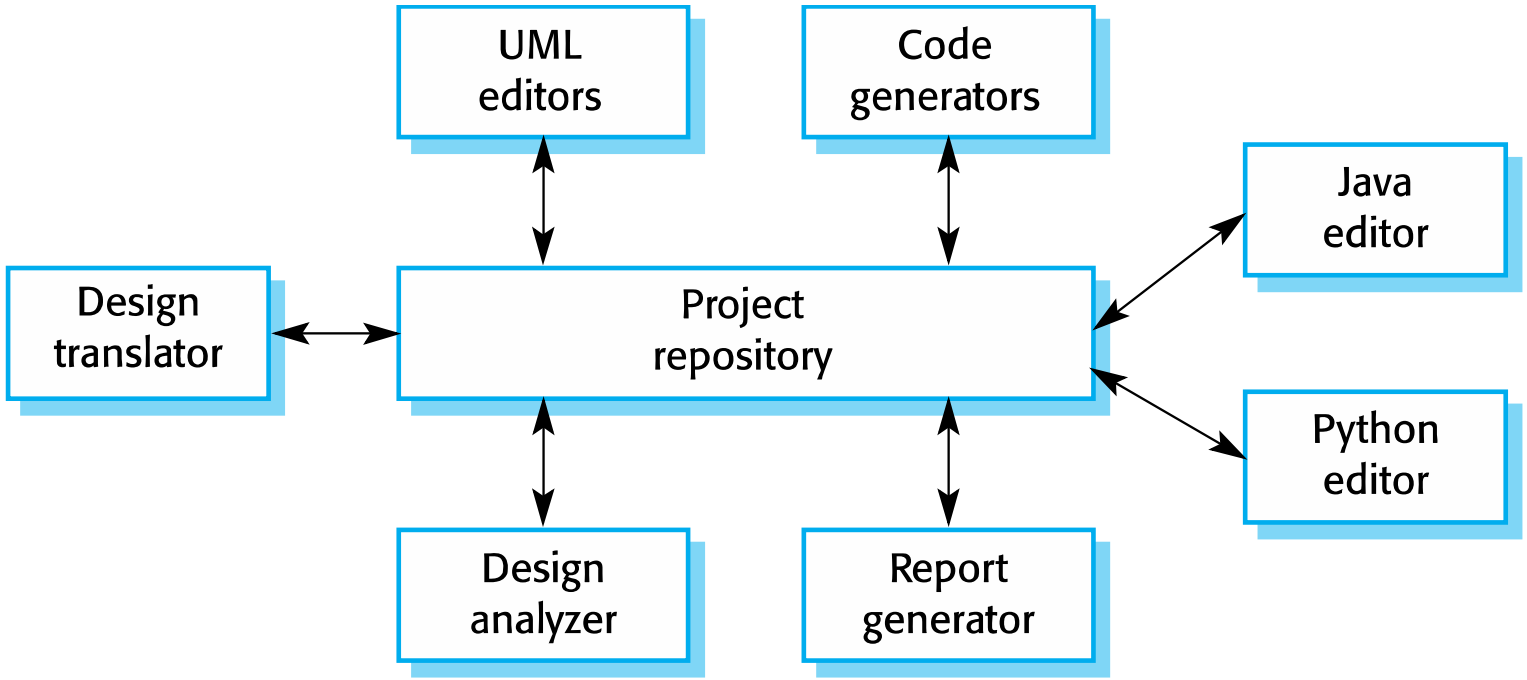
- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
-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

The Layered architecture pattern

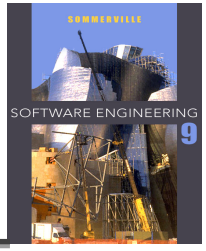


Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in a preceding slide.
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.

A repository architecture for an IDE

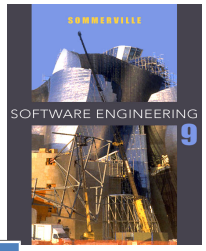


Repository architecture



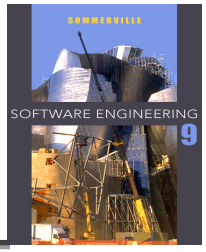
- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared,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

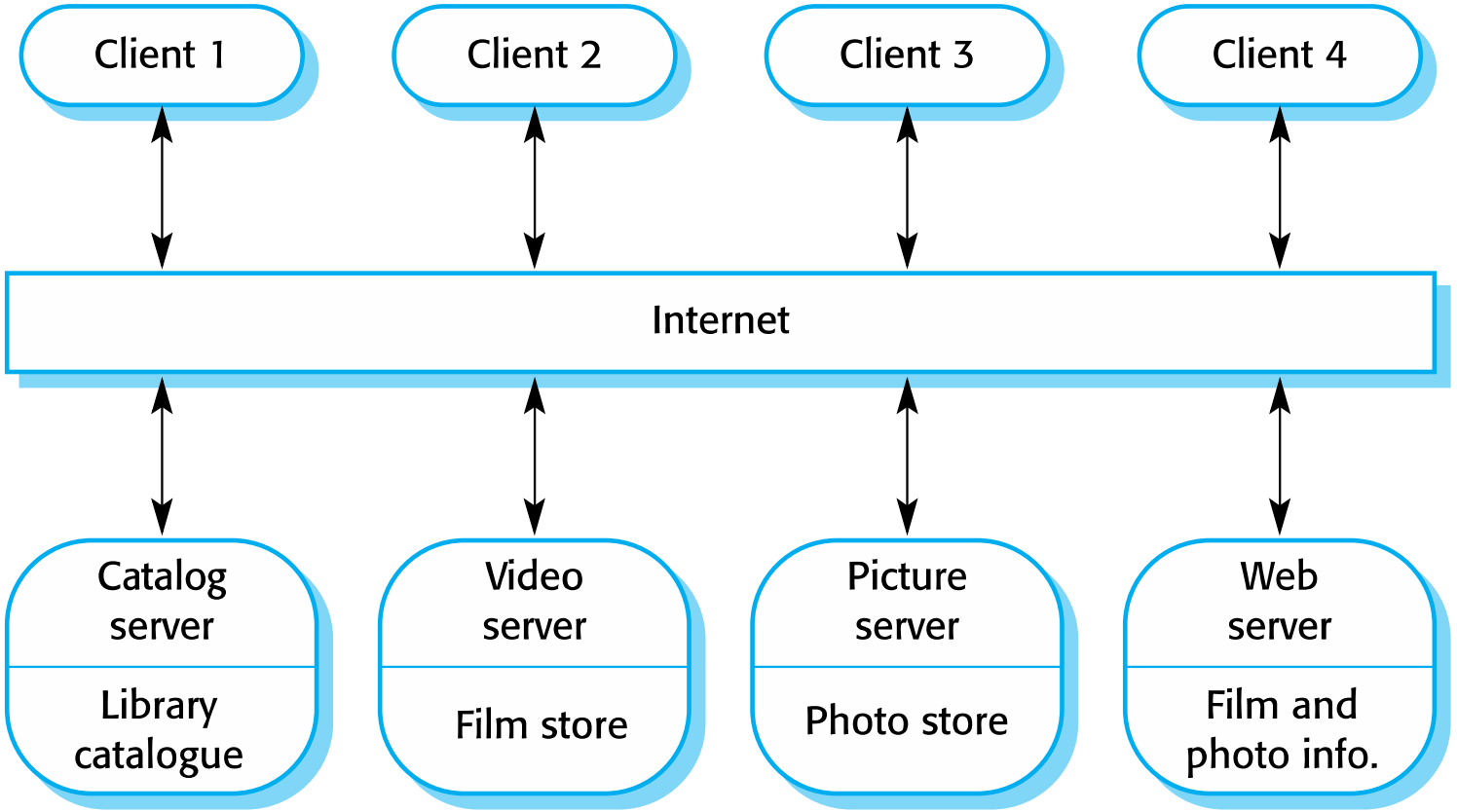
The Repository pattern



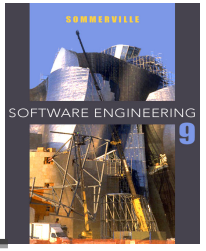
Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Preceding figure is an example of an IDE where the components use a repository of system design information. 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 client–server architecture for a film library



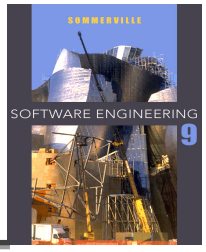


Client-server architecture



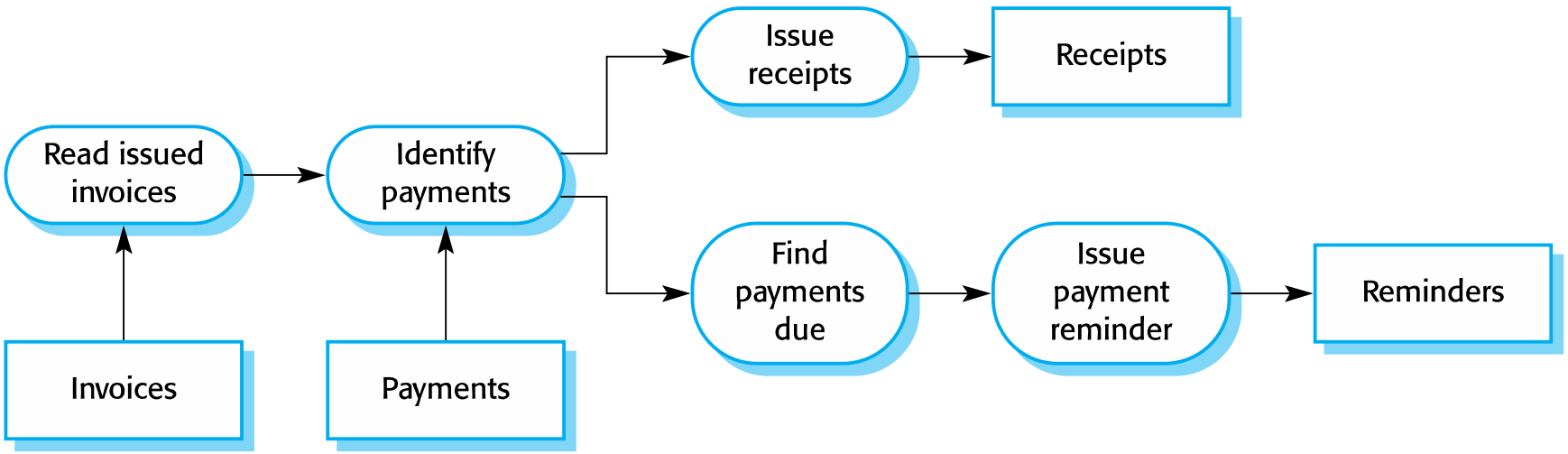
- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.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

The Client–server pattern

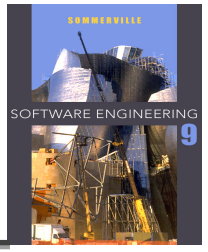


Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Preceding figure is an example of a film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

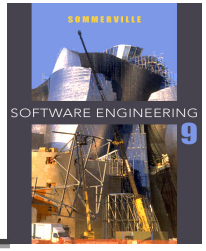


Pipe and filter architecture



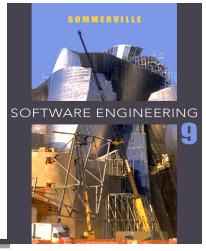
- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.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

The pipe and filter pattern



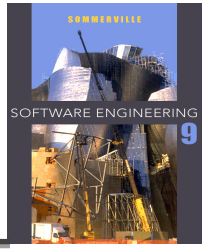
Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Preceding figure is an example 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.

Application architectures



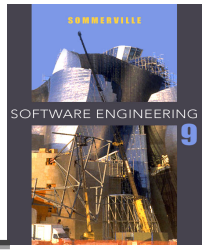
- Application systems are designed to meet an organisational need.
- As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- A generic application architecture is an architecture for a type of software system that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

Use of application architectures



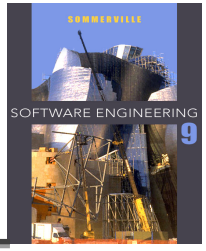
- As a starting point for architectural design.
- As a design checklist.
- As a way of organising the work of the development team.
- As a means of assessing components for reuse.
- As a vocabulary for talking about application types.

Examples of application types



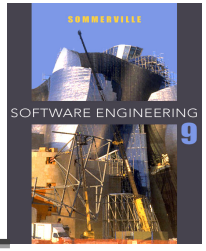
- Data processing applications
  - Data driven applications that process data in batches without explicit user intervention during the processing.
- Transaction processing applications
  - Data-centred applications that process user requests and update information in a system database.
- Event processing 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.

Examples for application types



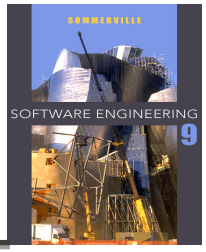
- Transaction processing systems
  - E-commerce systems;
  - Reservation systems.
- Language processing systems
  - Compilers;
  - Command interpreters.

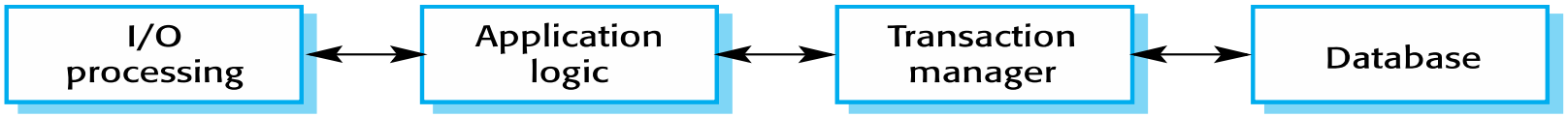
Transaction processing systems



- 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 structure of transaction processing applications

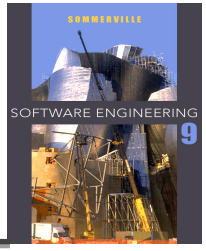




Architectures can be combined

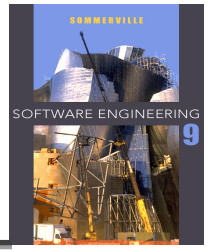
For an ATM may additionally use client-server architecture, splitting application logic between ATM and server(s).

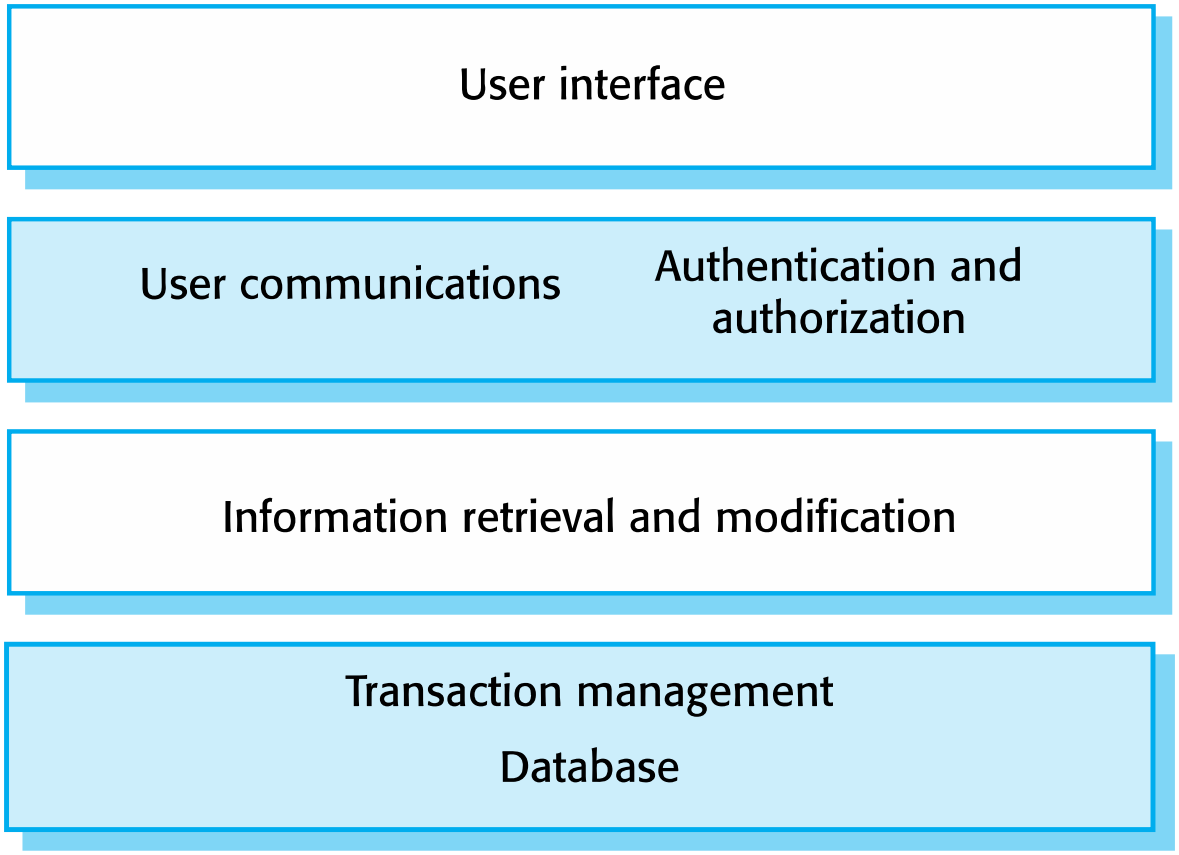
Information systems architecture



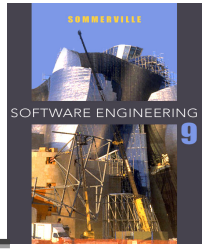
- Information systems have a generic architecture that can be organised as a layered architecture.
- These are transaction-based systems as interaction with these systems generally involves database transactions.
- Layers include:
  - The user interface
  - User communications
  - Information retrieval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

Layered information system architecture



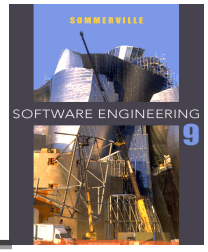


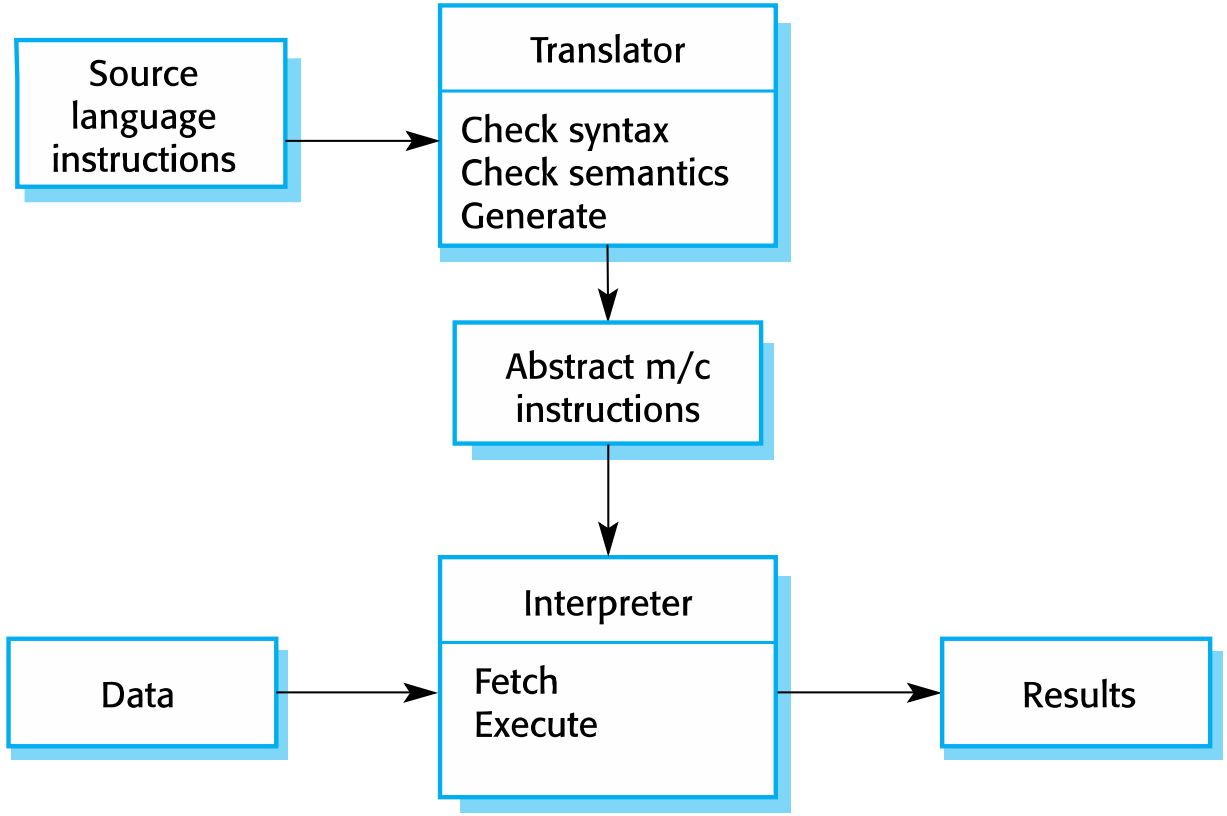
Language processing systems



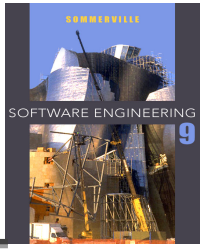
- Accept a natural or artificial language as input and generate some other representation of that language.
- May include an interpreter to act on the instructions in the language that is being processed.
- Used in situations where the easiest way to solve a problem is to describe an algorithm or describe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

The architecture of a language processing system



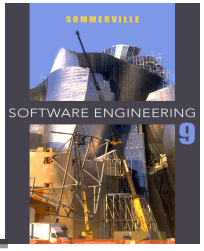


Compiler components (1)



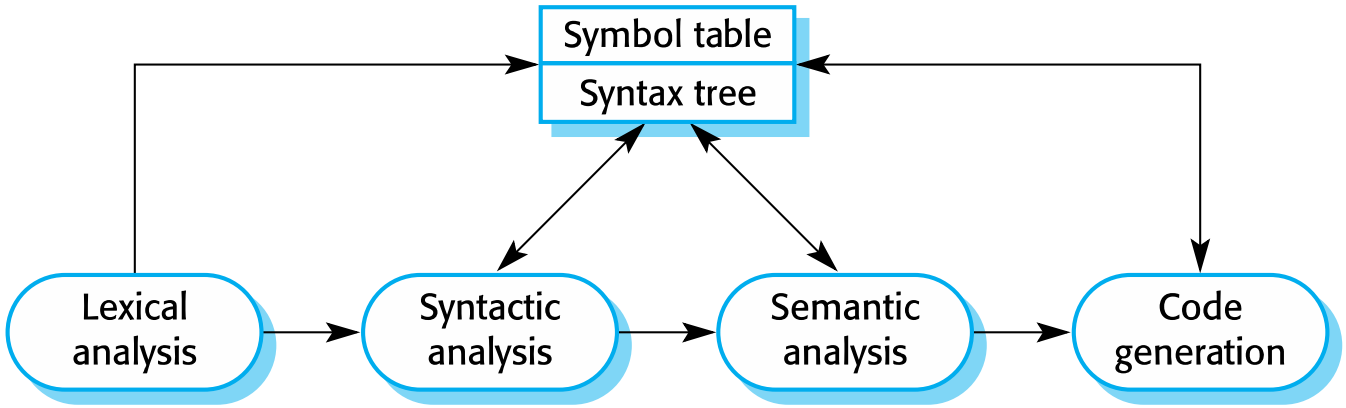
- A lexical analyser, which takes input language tokens and converts them to an internal form.
- A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
- A syntax analyzer, which checks the syntax of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

Compiler components (2)

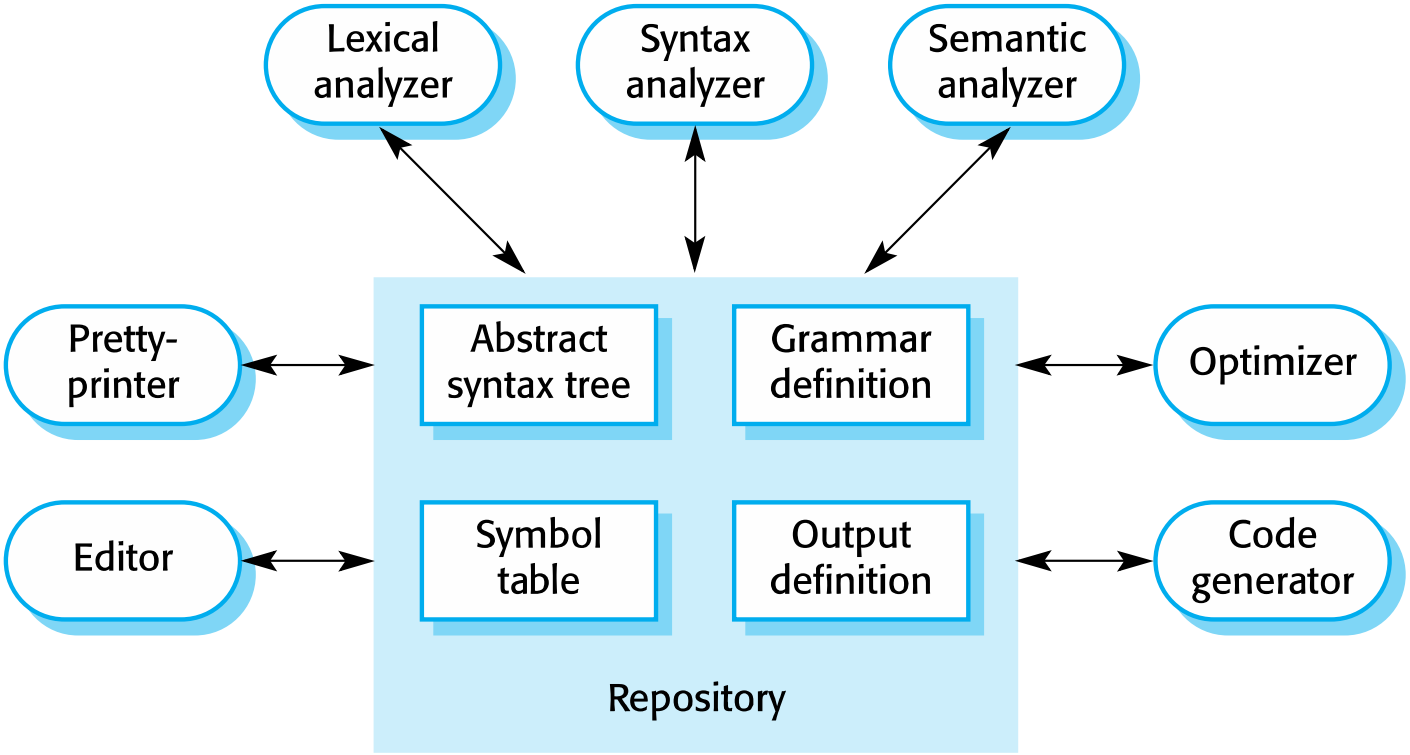


- A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
- A code generator that ‘walks’ the syntax tree and generates abstract machine code.

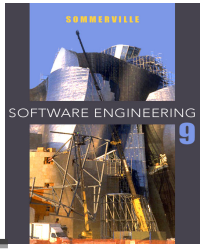
A pipe and filter compiler architecture



A repository architecture for a language processing system



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.
- 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.