SOFTWARE ENGINEERING

Software Design

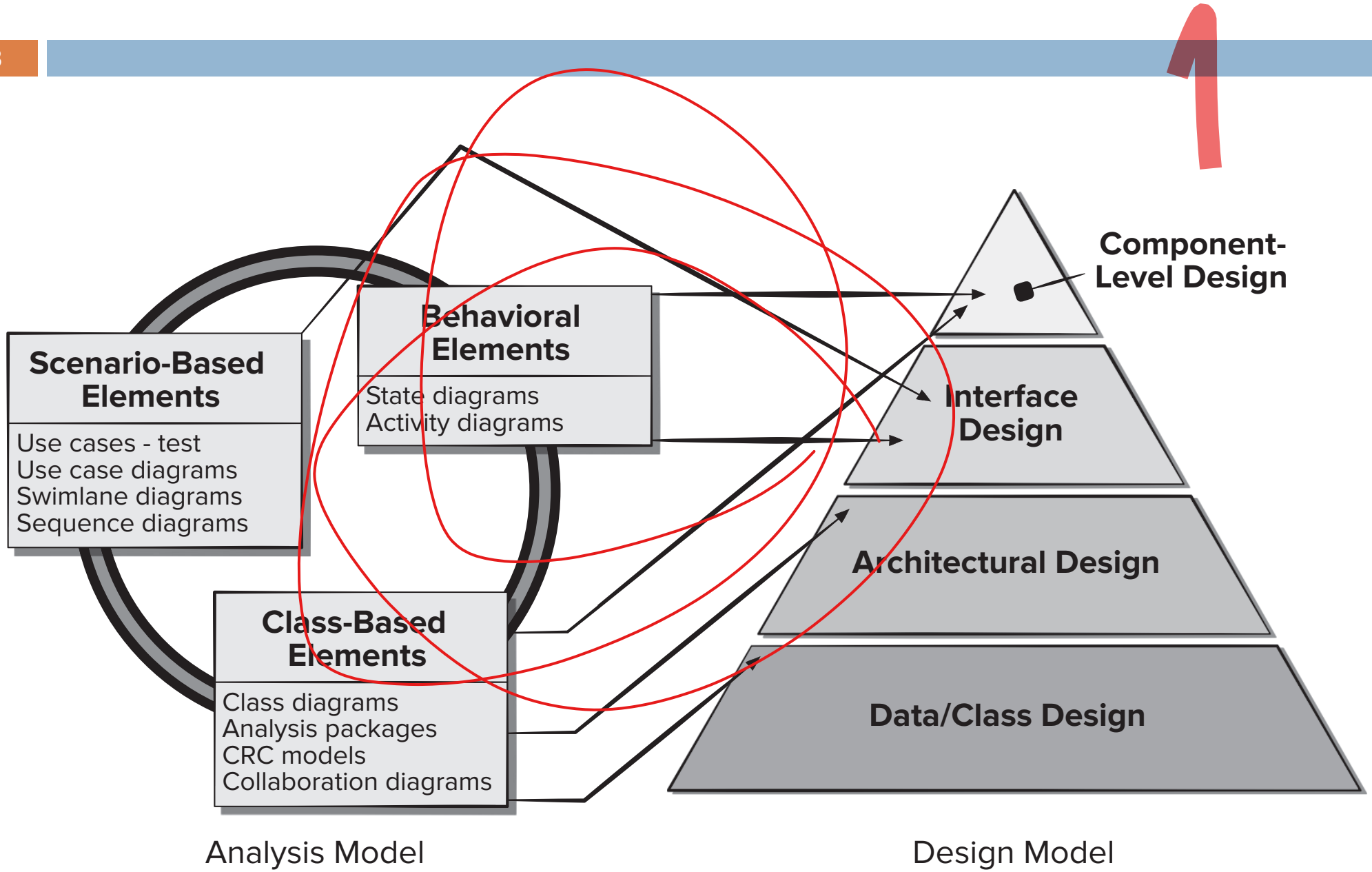
Course ID 06016410, 06016321

Nont Kanungsukkasem, B.Eng., M.Sc., Ph.D.
nont@it.kmitl.ac.th

Software Design

- It is the place where creativity rules—where requirements and technical considerations come together in the formulation of a product or system.
- Design creates a representation or model of the software and provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.
- Software design is the last software engineering action within the **modeling** activity and sets the stage for **construction** (code generation and testing).
- The importance of software design can be stated with a single word — **quality**.

Requirements model into Design model



Characteristics of Good Design

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

Generic Task Set for Design

1. Examine the information model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an **architectural style (pattern)** that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
   - Be certain that each subsystem is functionally cohesive.
   - Design subsystem interfaces.
   - Allocate analysis classes or functions to each subsystem.

Generic Task Set for Design

4. Create a set of design classes or components:
   - Translate an analysis class description into a design class.
   - Check each design class against design criteria; consider inheritance issues.
   - Define methods and messages associated with each design class.
   - Evaluate and select **design patterns** for a design class or a subsystem.
   - Review design classes and revise as required.
5. Design any interface required with external systems or devices.

Generic Task Set for Design

6. Design the user interface:
   - Review results of task analysis.
   - Specify action sequence based on user scenarios.
   - Create a behavioral model of the interface.
   - Define interface objects and control mechanisms.
   - Review the interface design, and revise as required.
7. Conduct component-level design. Specify all algorithms at a relatively low level of abstraction.
   - Refine the interface of each component.
   - Define component-level data structures.
   - Review each component, and correct all errors uncovered.

Generic Task Set for Design

8. Develop a deployment model.

- Please note:
  - These tasks are often performed iteratively and in parallel.
  - They are rarely completed sequentially and in isolation from one another unless you are following the waterfall process model.

Design Concepts

- Abstraction
- Architecture
- Patterns
- Separation of Concerns
- Modularity
- Information Hiding
- Functional Independence
- Stepwise Refinement
- Refactoring
- Design Classes

Abstraction

- It is the act of representing the essential features without including the background details or explanations.
- Many level of abstraction can be posed:
  - Highest level of abstraction
  - Lower level of abstraction
  - Lowest level of abstraction
- As different levels of abstraction are developed:
  - Procedural abstraction
  - Data abstraction

Architecture

- The overall structure of the software and the ways in which that structure provides conceptual integrity for a system
- In its simplest form, architecture is the structure or organization of program components (modules), the ways in which these components interact, and the structure of data that are used by the components.

Patterns

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

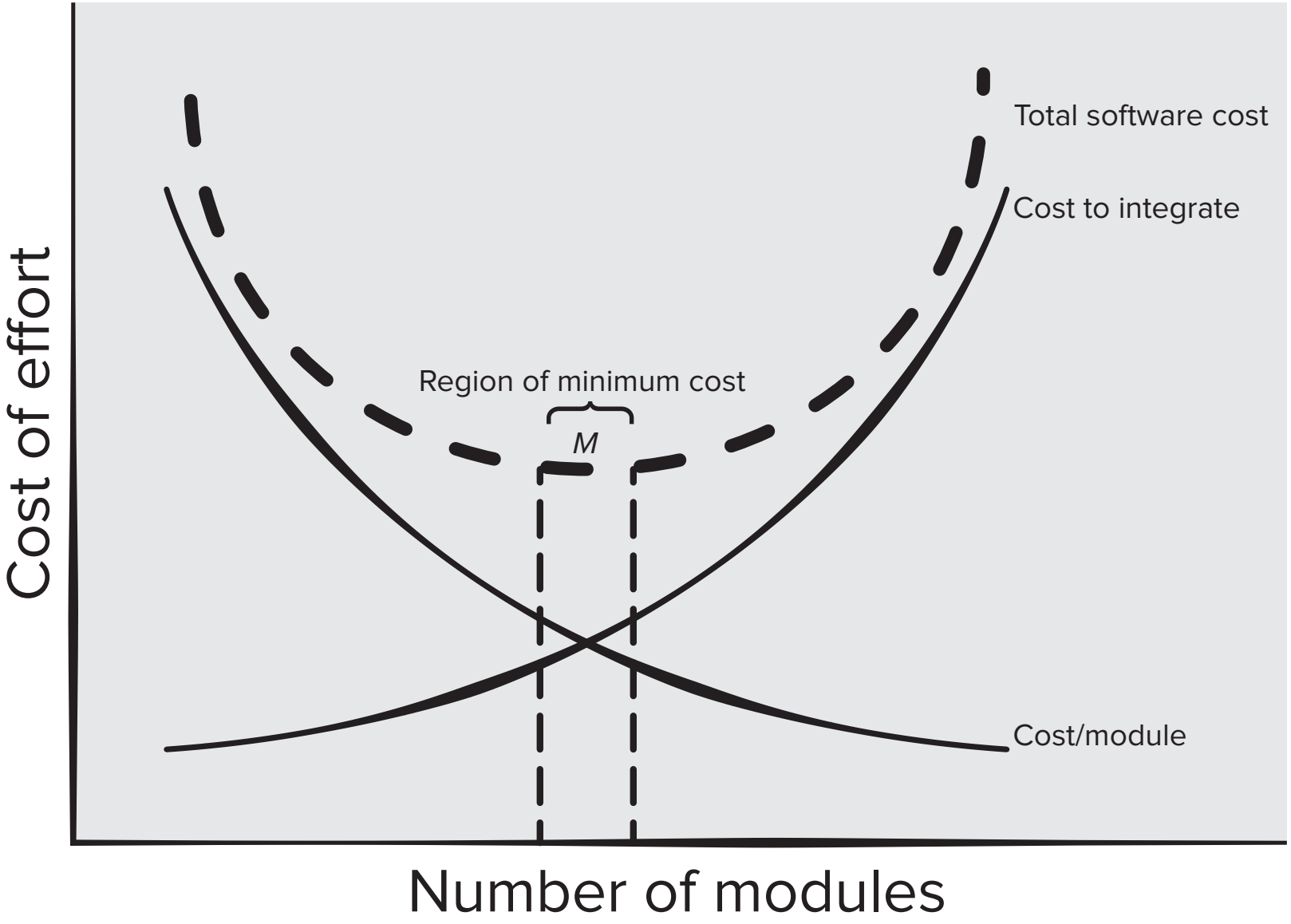
Separation of Concerns

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

Modularity

- Modularity is the most common manifestation of separation of concerns.
- Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and reduce the cost required to build the software.
- You modularize a design (and the resulting program) so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Modularity and Software Cost



Information Hiding

- Modules should be characterized by design decisions that (each) hides from all others.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

Encapsulation

```java
class Person {
    private String name;    // private attribute
    private int age;        // private attribute

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Public method to display information (indirectly using private data and methods)
    public void displayPersonDetails() {
        System.out.println("Name: " + getName());
        System.out.println("Age: " + getAge());
        greet();  // Indirectly using a private method
    }

    // Public method to get the name attribute
    public String getName() {
        return name;
    }

    // Public method to get the age attribute
    public int getAge() {
        return age;
    }

    // Private method that is only accessible within the class
    private void greet() {
        System.out.println("Hello, nice to meet you!");
    }
}
public class Main
{
   public static void main(String args[])
  {
    Person person = new Person("John Doe", 25);
    person.displayPersonDetails();
 }
}
```

```
Name: John Doe
Age: 25
Hello, nice to meet you!
```

Abstraction in Object-Oriented Programming

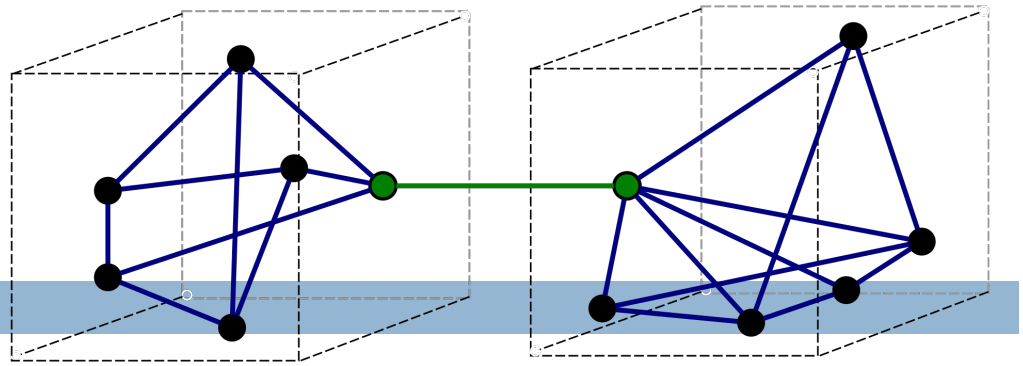

```java
abstract class Shape {
    public abstract void draw();
    public void resize() {
        System.out.println("Resizing the shape");
    }
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

```java
interface Drawable {
    void draw();  // Abstract method
}
```

Functional Independence

- Functional independence is achieved by developing modules with “single-minded” function and an “aversion” to excessive interaction with other modules.
- Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

Qualitative Criteria



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

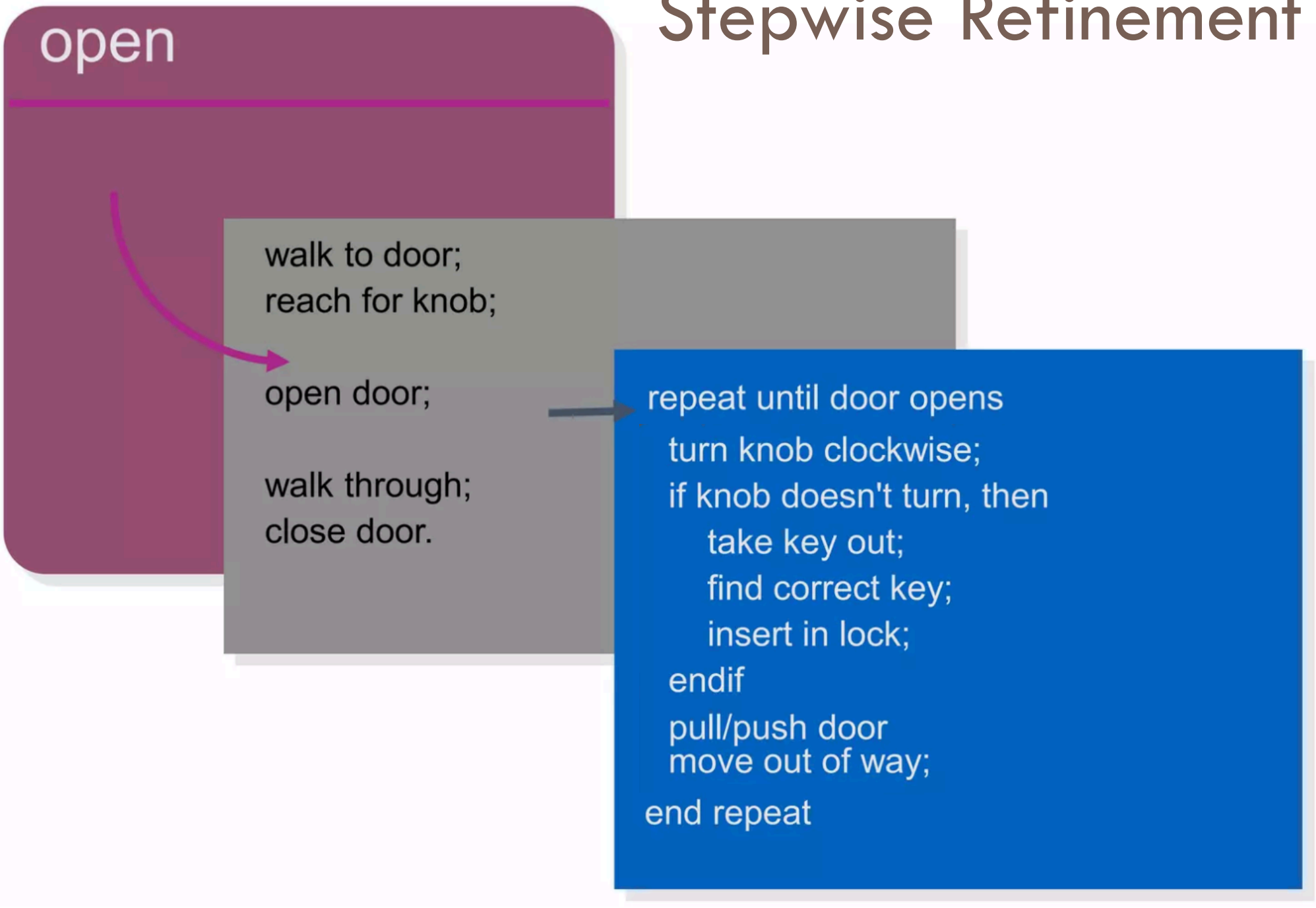
Stepwise Refinement

- Refinement is a process of *elaboration*.
- You begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- After defining an abstraction, we elaborate on the original statement, providing more detail as each successive refinement (elaboration) occurs.

Abstraction and Refinement

- Abstraction and refinement are complementary concepts.
- Abstraction enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details.
- Refinement helps you to reveal low-level details as design progresses.
- Both concepts allow you to create a complete design model as the design evolves.

Stepwise Refinement



Refactoring

- Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for
  - Redundancy
  - Unused design elements
  - Inefficient or unnecessary algorithms
  - Poorly constructed or inappropriate data structures
  - Or any other design failure that can be corrected to yield a better design

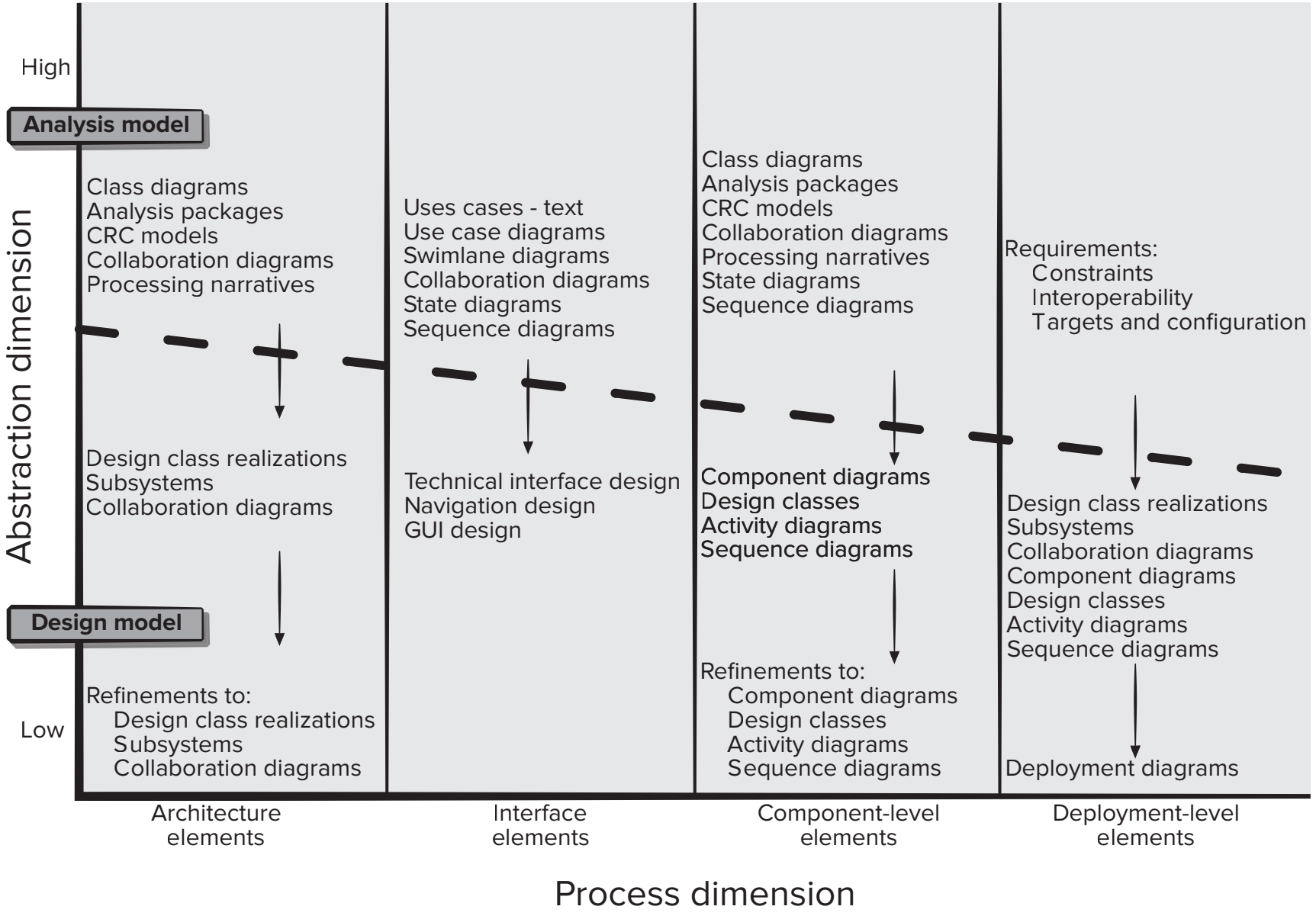
Design Classes

- Design classes present significantly more technical detail as a guide for implementation.
- Each design class should be reviewed to ensure that it is "well-formed."
  - Complete and sufficient
    - The **complete** encapsulation of all attributes and methods that can reasonably be expected.
    - The design class contains only those methods that are **sufficient** to achieve the intent of the class, no more and no less.
  - Primitiveness
    - Methods associated with a design class should be focused on accomplishing one service for the class.
    - The class should not provide another way to accomplish the same thing.
  - High cohesion
  - Low coupling

Design Models

- The design model can be viewed in two different dimensions:
  - The **process** dimension indicates the evolution of the design model as design tasks are executed as part of the software process.
  - The **abstraction** dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

Dimensions of the Design model



Design Modeling Principles

- Design should be traceable to the requirements model.
- Always consider the architecture of the system to be built.
- Design of data is as important as design of processing functions.
- Interfaces (both internal and external) must be designed with care.
- Component-level design should be functionally independent.
- Components should be loosely coupled to one another and to the external environment.
- Design representations (models) should be easily understandable.
- The design should be developed iteratively.
- Creation of a design model does not preclude an agile approach.

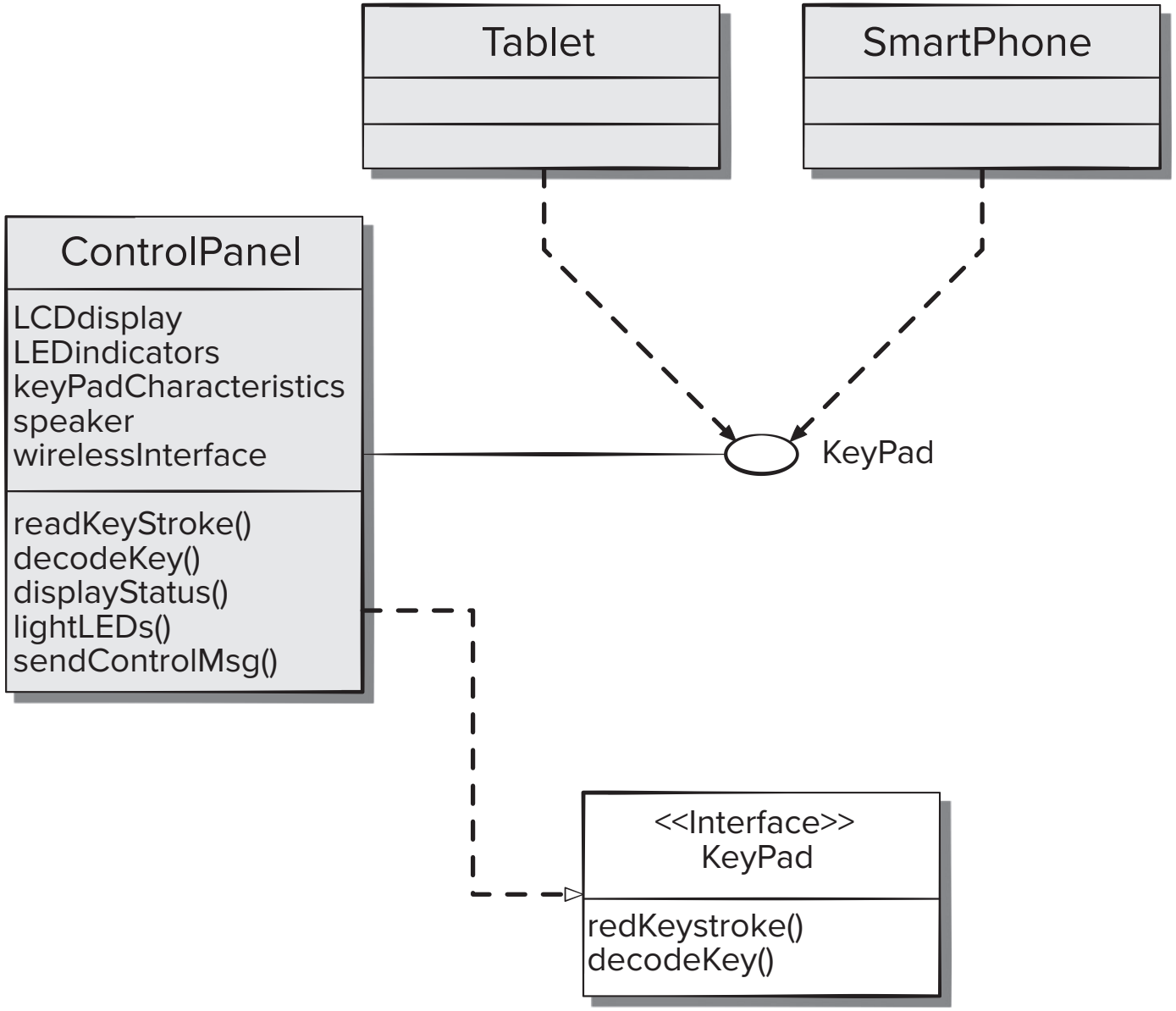
Design Model Elements

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

Interface Elements

- The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.
- Three interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.
  - UI
  - External interfaces
  - Internal interfaces

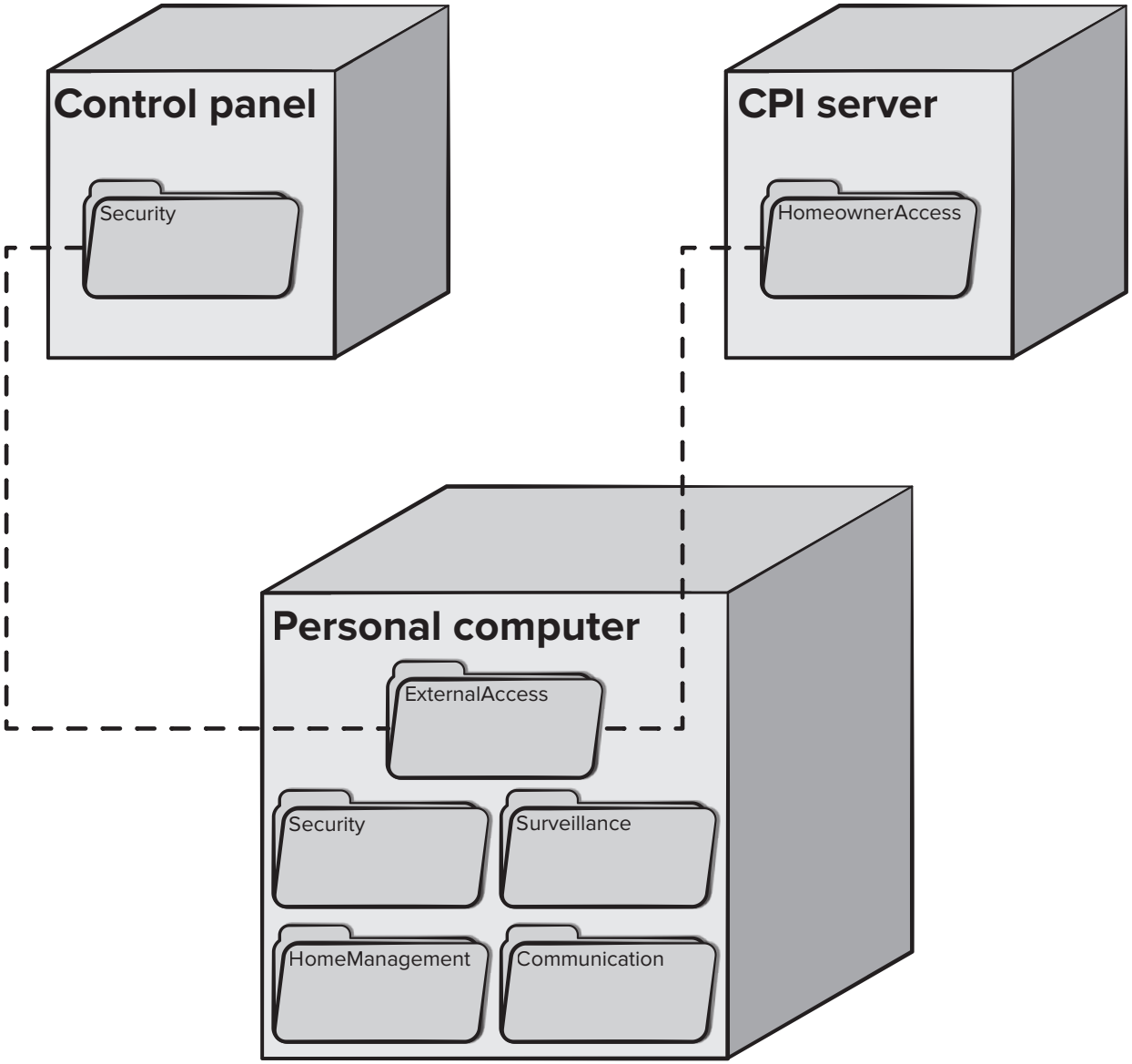
Interface representation for ControlPanel



Deployment-level Elements

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
- UML Deployment Diagram
  - *Descriptor Form* shows the computing environment but does not explicitly indicate configuration details.
  - *Instance Form* identifies each instance of the deployment (a specific, named hardware configuration)

A UML deployment diagram

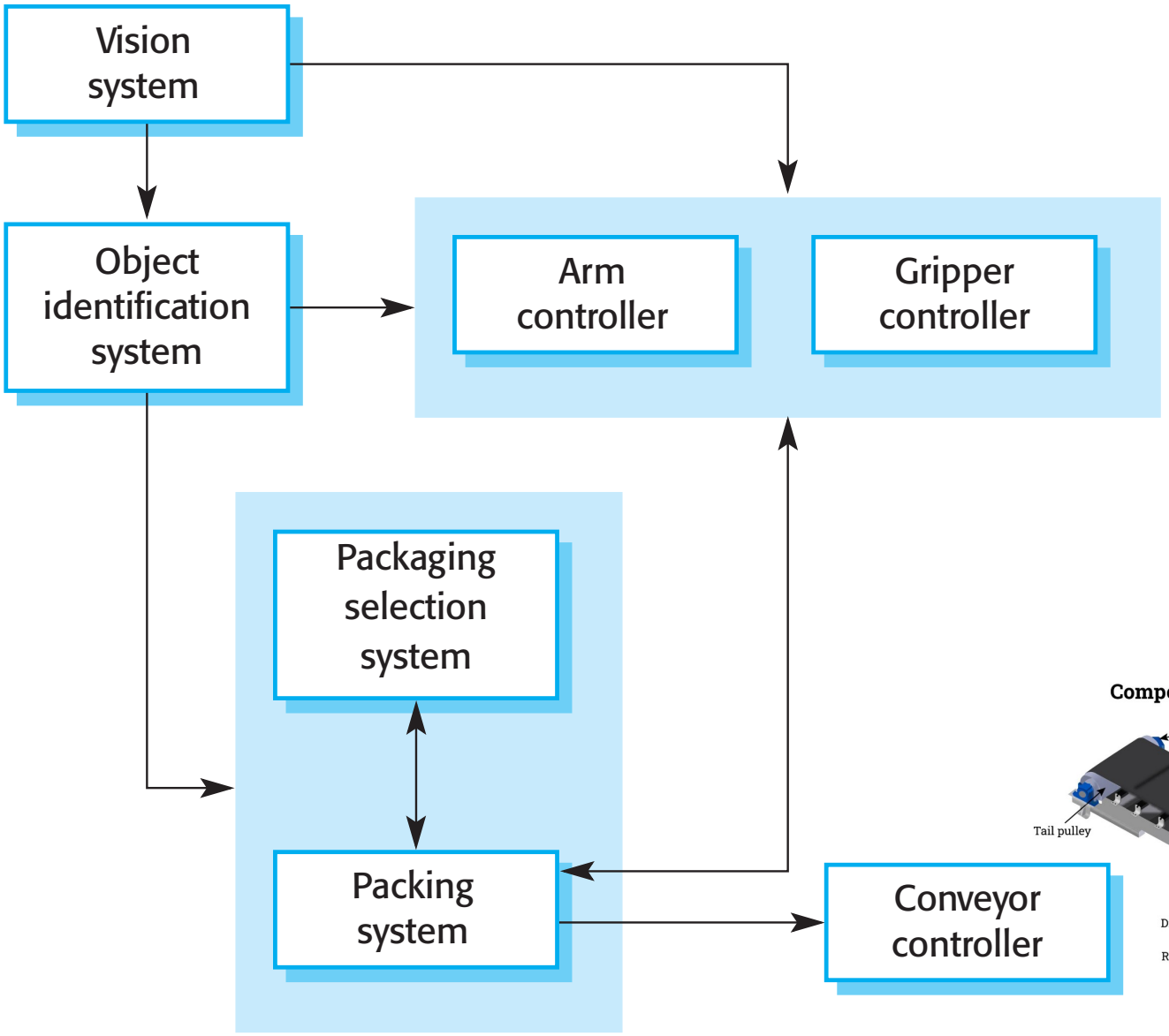


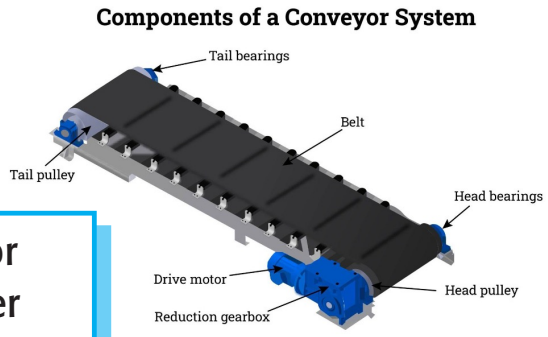
Architectural Design

Architectural Design in Agile

- In agile processes, it is generally accepted that an early stage of an agile development process should focus on designing an overall system architecture.
- Incremental development of architectures is not usually successful.
- Refactoring components in response to changes is usually relatively easy.
- However, refactoring the system architecture is expensive because you may need to modify most system components to adapt them to the architectural changes.

The architecture of a packing robot control system

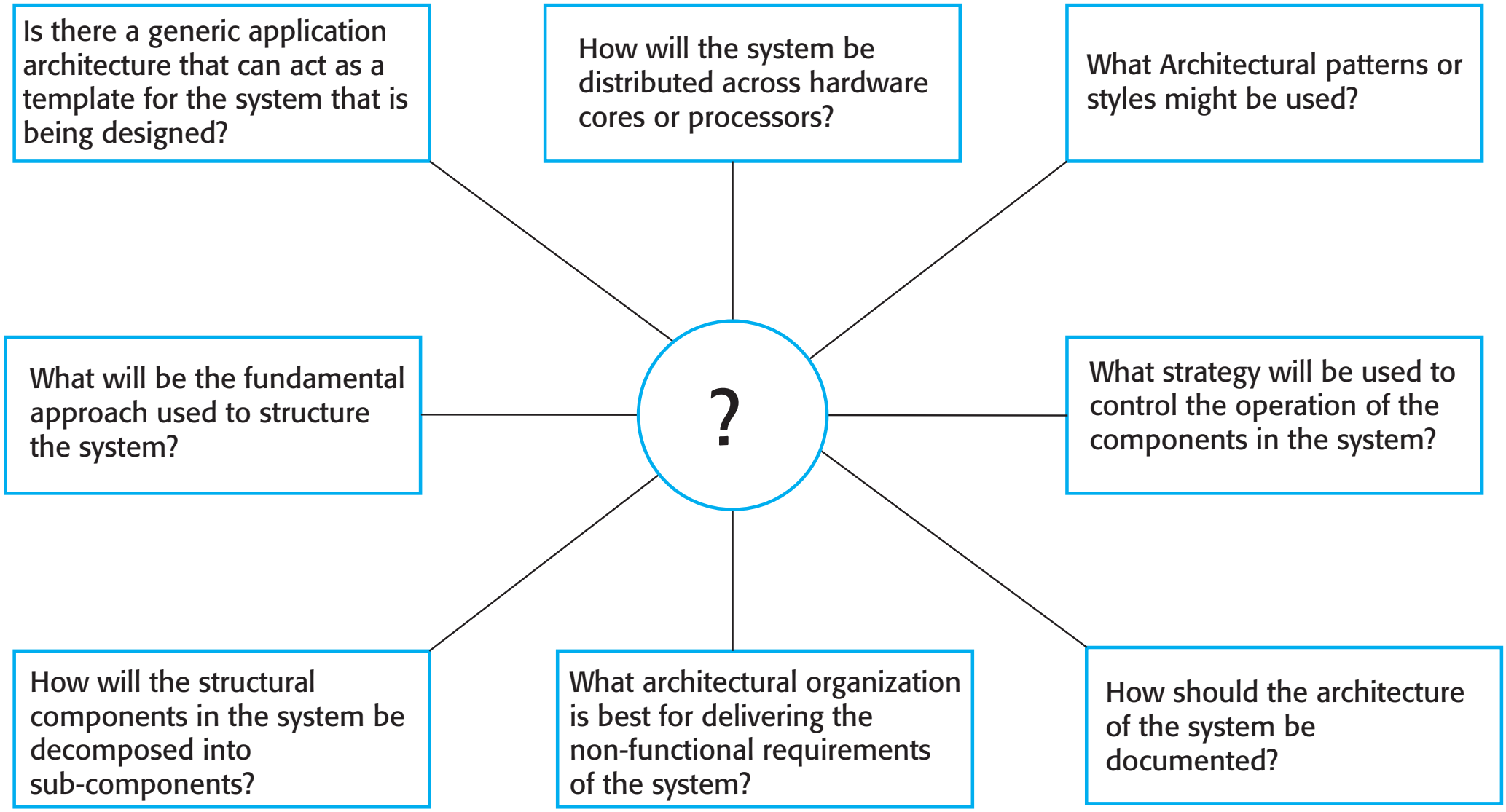




Uses of Architectural Model

- An architectural model of a program is used in 2 ways:
  - As a way of encouraging discussions about the system design
  - As a way of documenting an architecture that has been designed

Architectural design decisions



Choice of architectural style and structure based on the non-functional requirements

- Performance
  - a small number of components
  - the same computer
- Security
  - a layered structure
- Safety
  - colocation

Choice of architectural style and structure based on the non-functional requirements

- Availability
  - redundant components
- Maintainability
  - small, fine-grain, self-contained components
  - shared data structures should be avoided

- Obviously, there is potential conflict between some of these architectures.
  - Some compromise must be found.
- You can sometimes do this by using different architectural patterns or styles for separate parts of the system.

Architectural patterns/styles

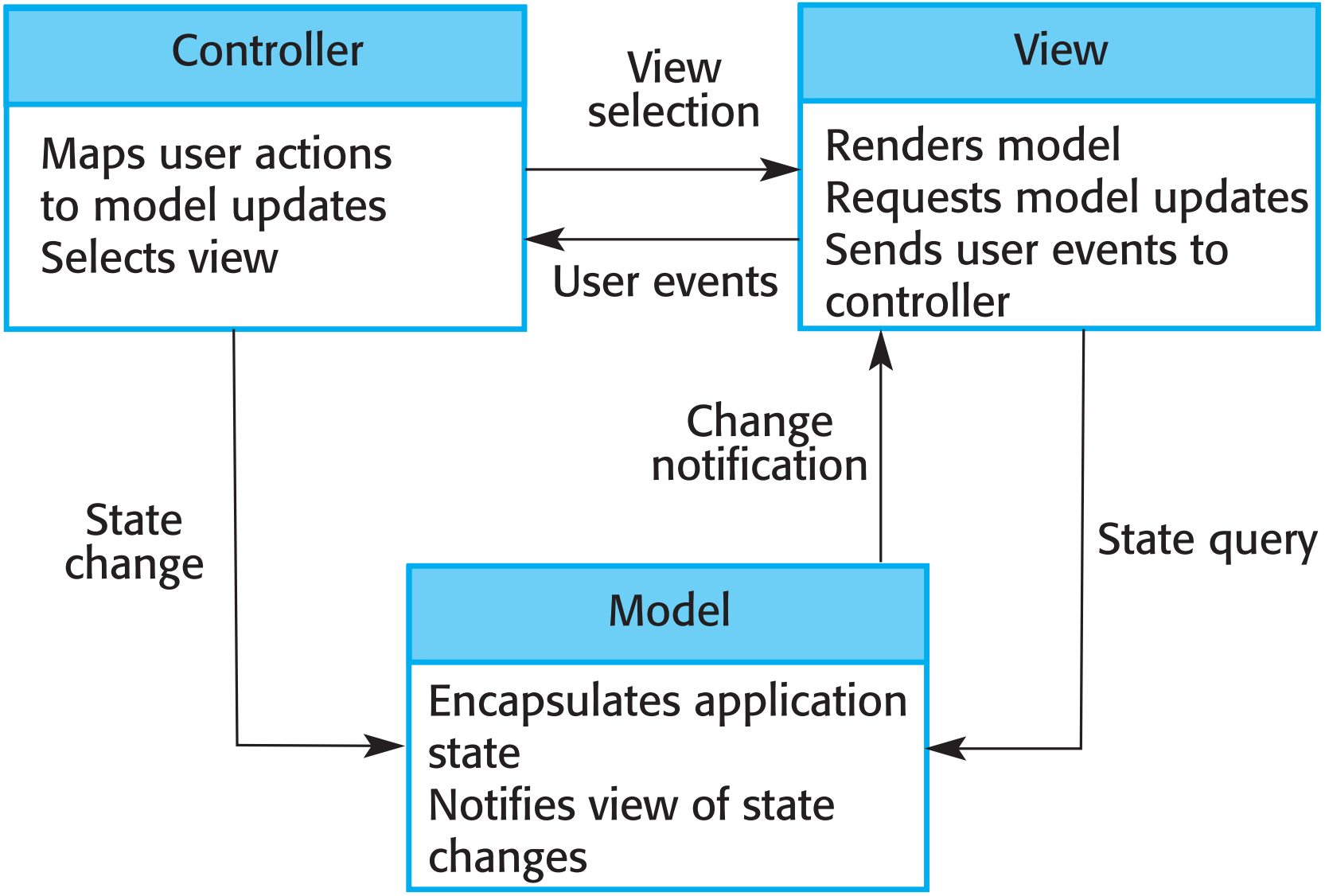
- The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems has been adopted in a number of areas of software engineering.
- An architectural pattern should
  - describe a system organization that has been successful in previous systems
  - include information on when it is and is not appropriate to use that pattern
  - Include details on the pattern's strengths and weaknesses.

Shaw, M., and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall.
Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Vo*lume 4: A Pattern Language for Distributed Computing. New York: John Wiley & Sons.
Buschmann, F., K. Henney, and D. C. Schmidt. 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. *New York: John Wiley & Sons*.
Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Archi*tecture Volume 1: A System of Patterns. New York: John Wiley & Sons.
Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.
Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. *New York: John Wiley & Sons*.

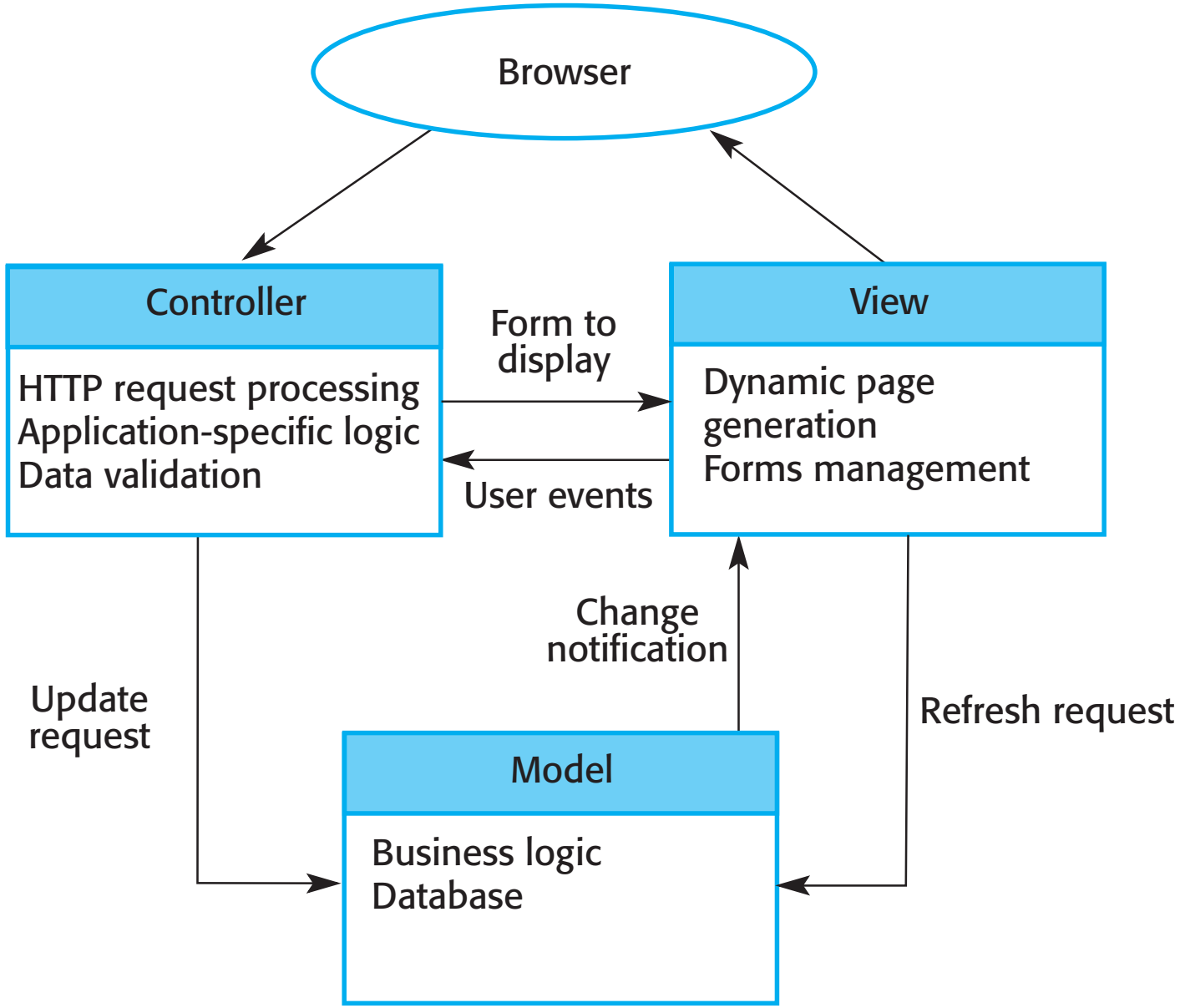
Model-View-Controller

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 6.5.
Example	Figure 6.6 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	May involve additional code and code complexity when the data model and interactions are simple.

The organization of the Model-View-Controller



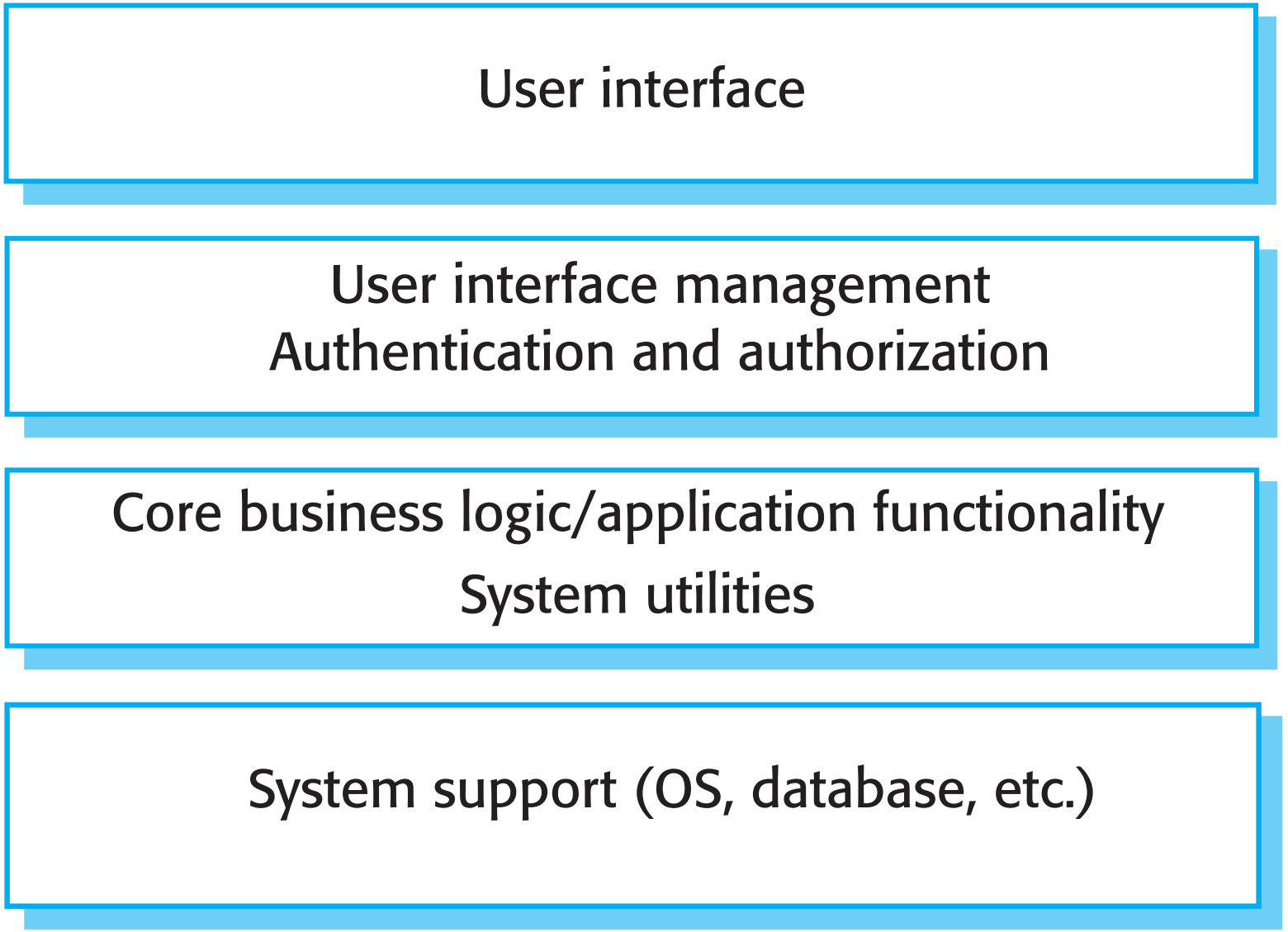
Web application architecture using the MVC pattern



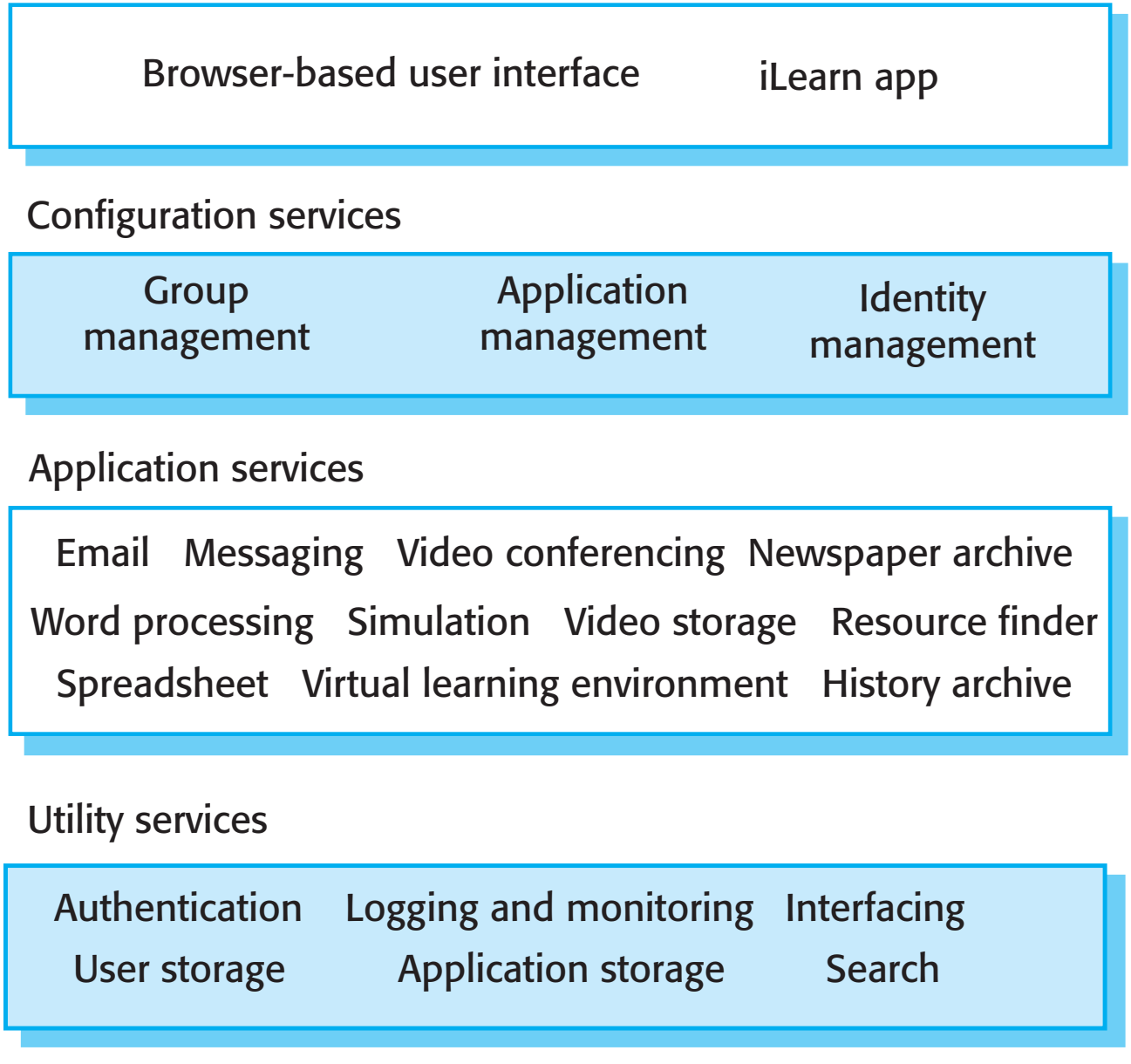
Layered Architecture

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 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
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 multilevel security.
Advantages	Allows replacement of entire layers as 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.

The Layered Architecture pattern



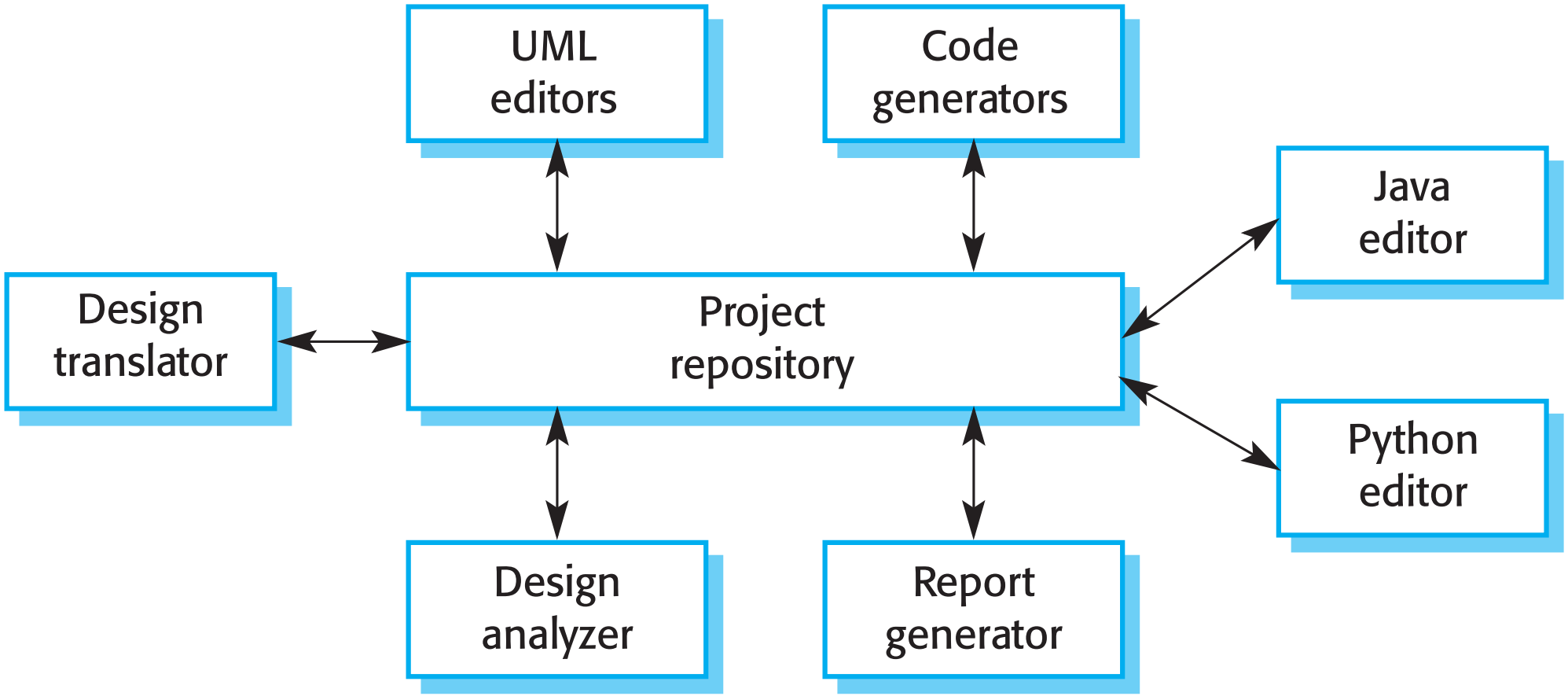
The architecture of the iLearn system using the Layered Architecture pattern



Repository architecture

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 6.11 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

- The major components of this model are:
  - A set of servers that offer services to other components
  - A set of clients that call on the services offered by servers
  - A network that allows the clients to access these services
- Client-server architectures are usually thought of as distributed systems architectures, but the logical model of independent services running on separate servers can be implemented on a single computer.

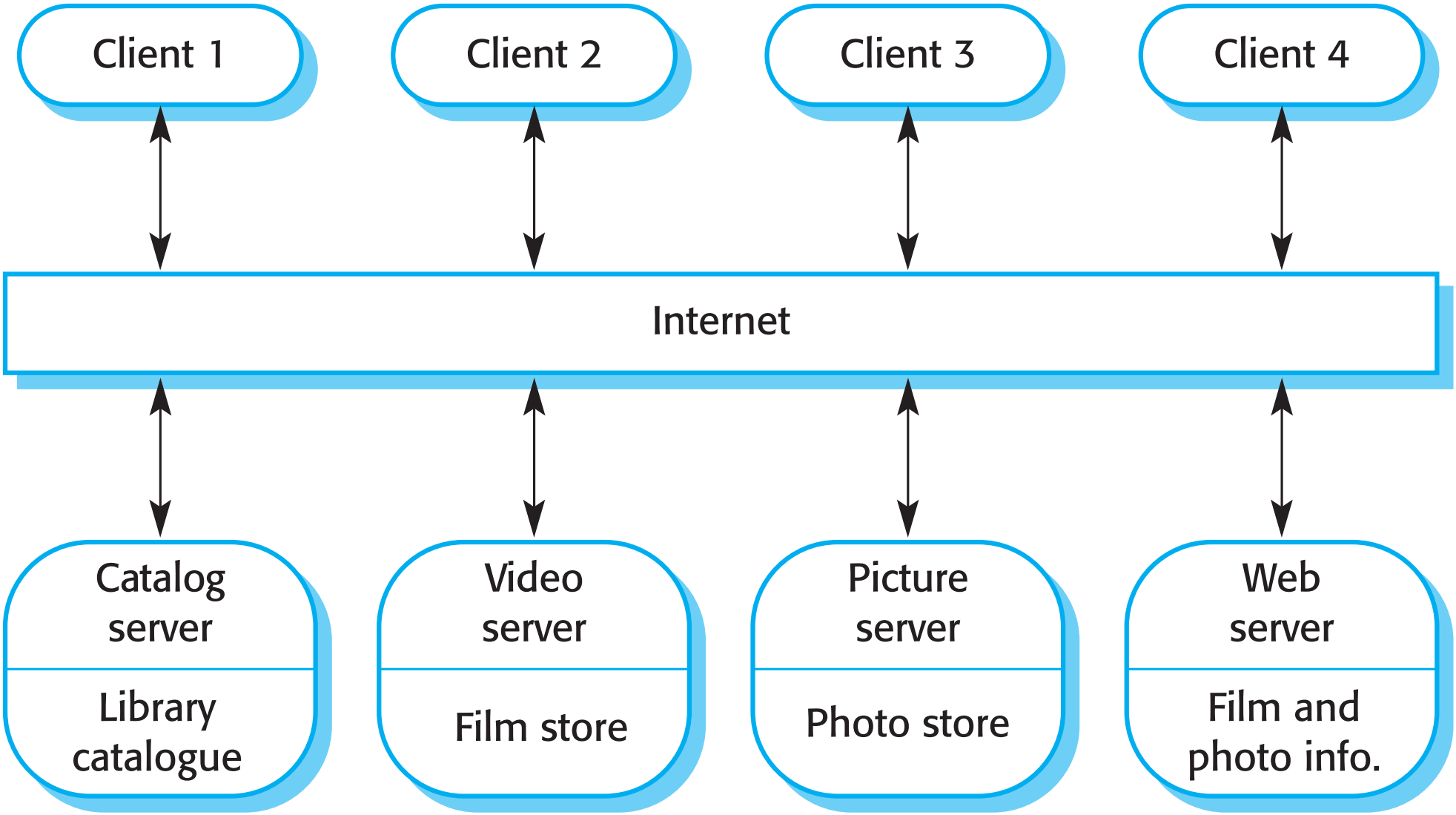
Client-Server

- Clients may have to know the names of the available servers and the services they provide.
- However, servers do not need to know the identity of clients or how many clients are accessing their services.
- Clients access the services provided by a server through remote procedure calls using a request–reply protocol (such as http), where a client makes a request to a server and waits until it receives a reply from that server.

Client-Server

Name	Client–server
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 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 and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

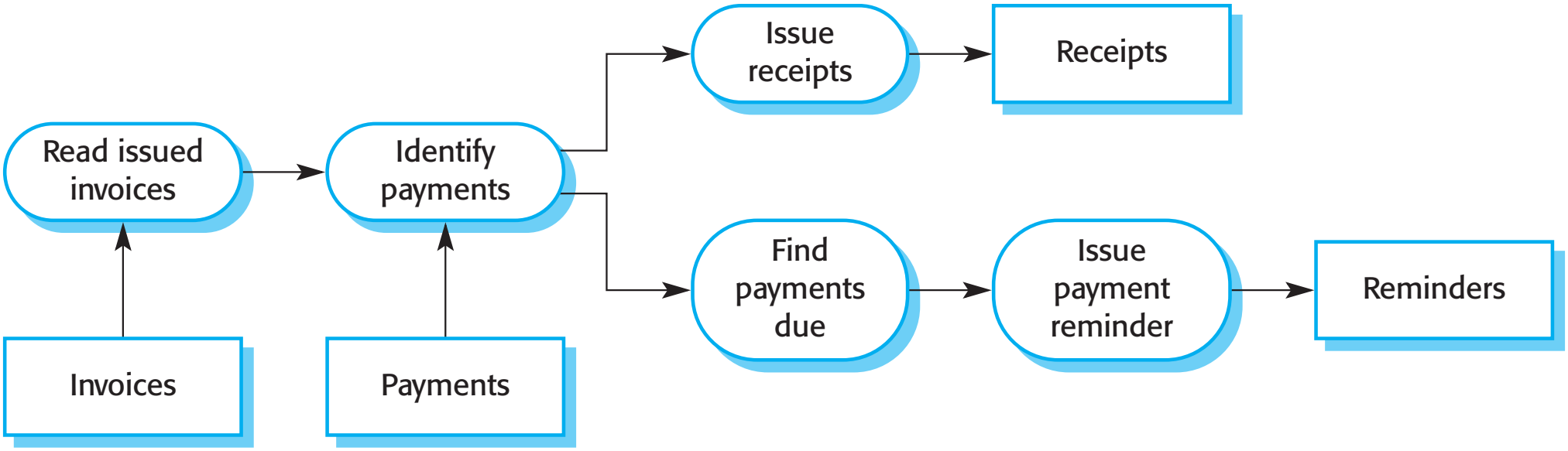
A Client-Server architecture for a film and photograph library



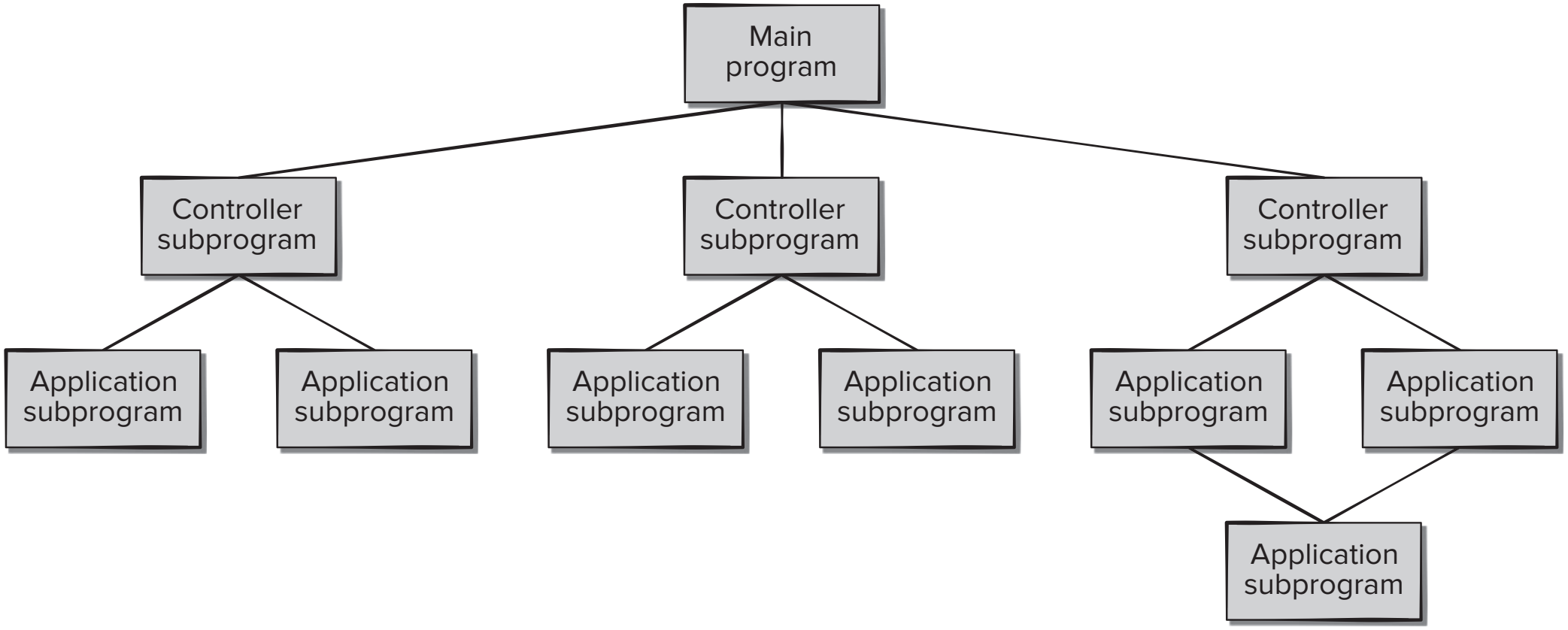
Pipe and Filter

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 6.15 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 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 architectural components that use incompatible data structures.

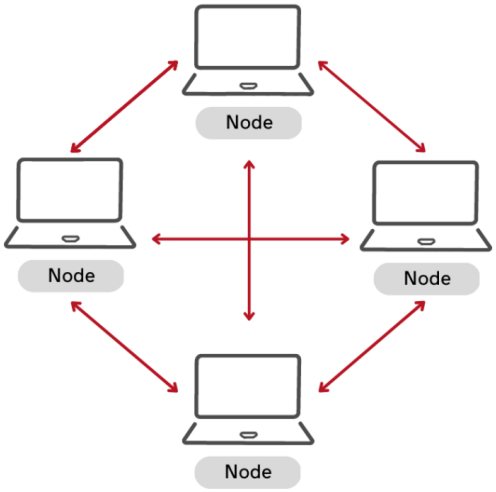
A Pipe and Filter architecture for a payment system



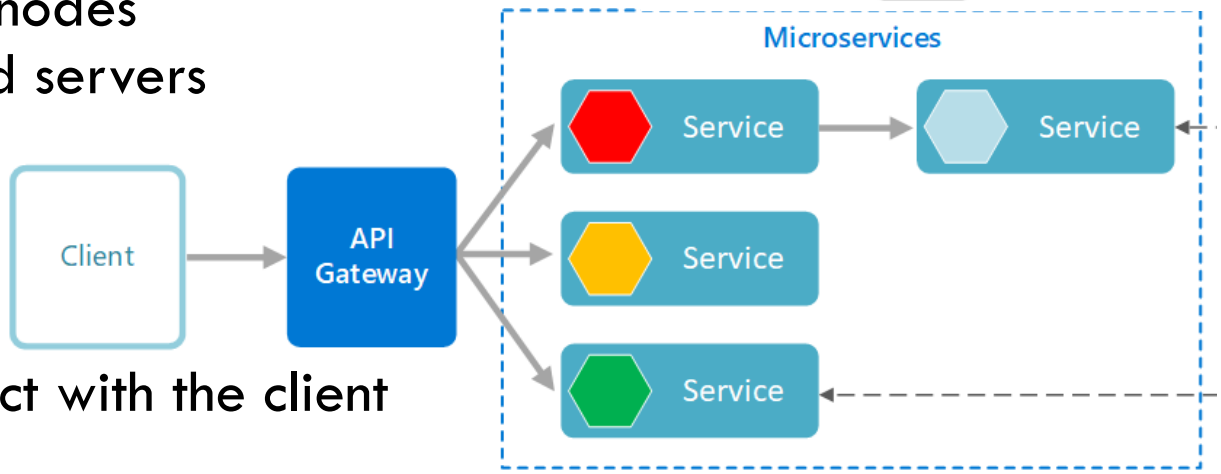
Call-and-Return Architecture



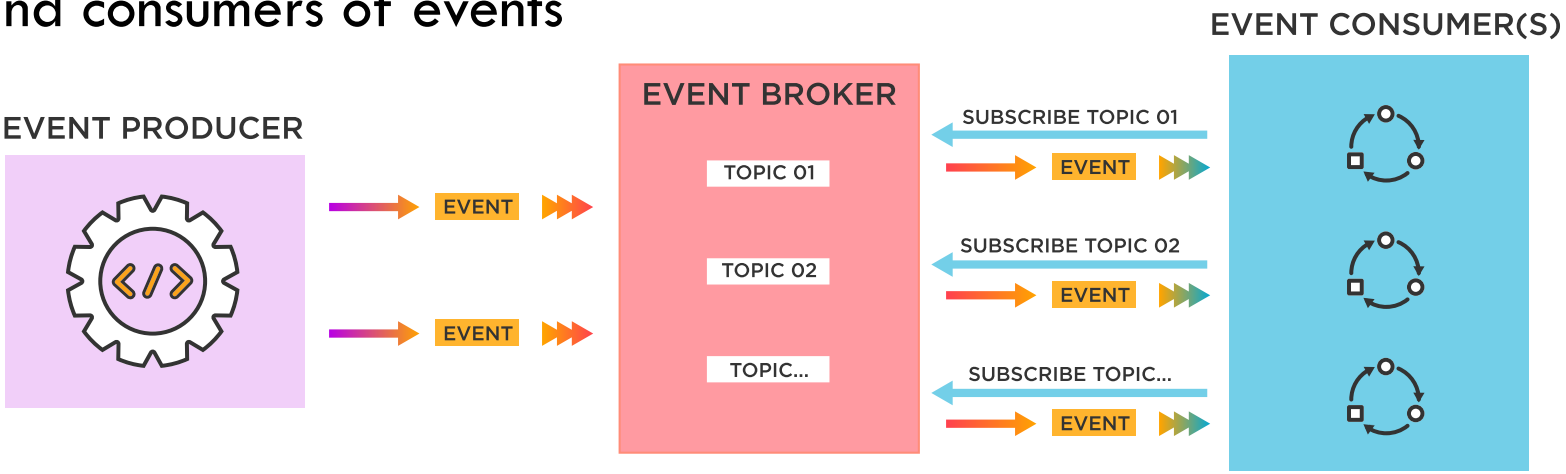
Other architectural patterns



- Peer-to-peer architectures
  - decentralized network of nodes that act as both clients and servers
- Microservices
  - loosely coupled individual services that behave as a single system and interact with the client



- Event-driven
  - producers and consumers of events



Other architectural patterns

- There are specific Architectural patterns that reflect commonly used ways of organizing control in a system. These include
  - centralized control, based on one component calling other components, and
  - event-based control, where the system reacts to external events.

http://software-engineering-book.com/web/archpatterns/

Application architectures

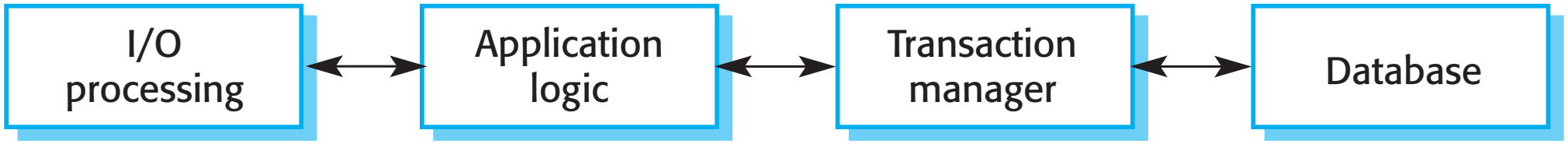
- Application systems are intended to meet a business or an organizational need.
- As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements
- Application architectures encapsulate the principal characteristics of a class of systems

Uses of application architectures

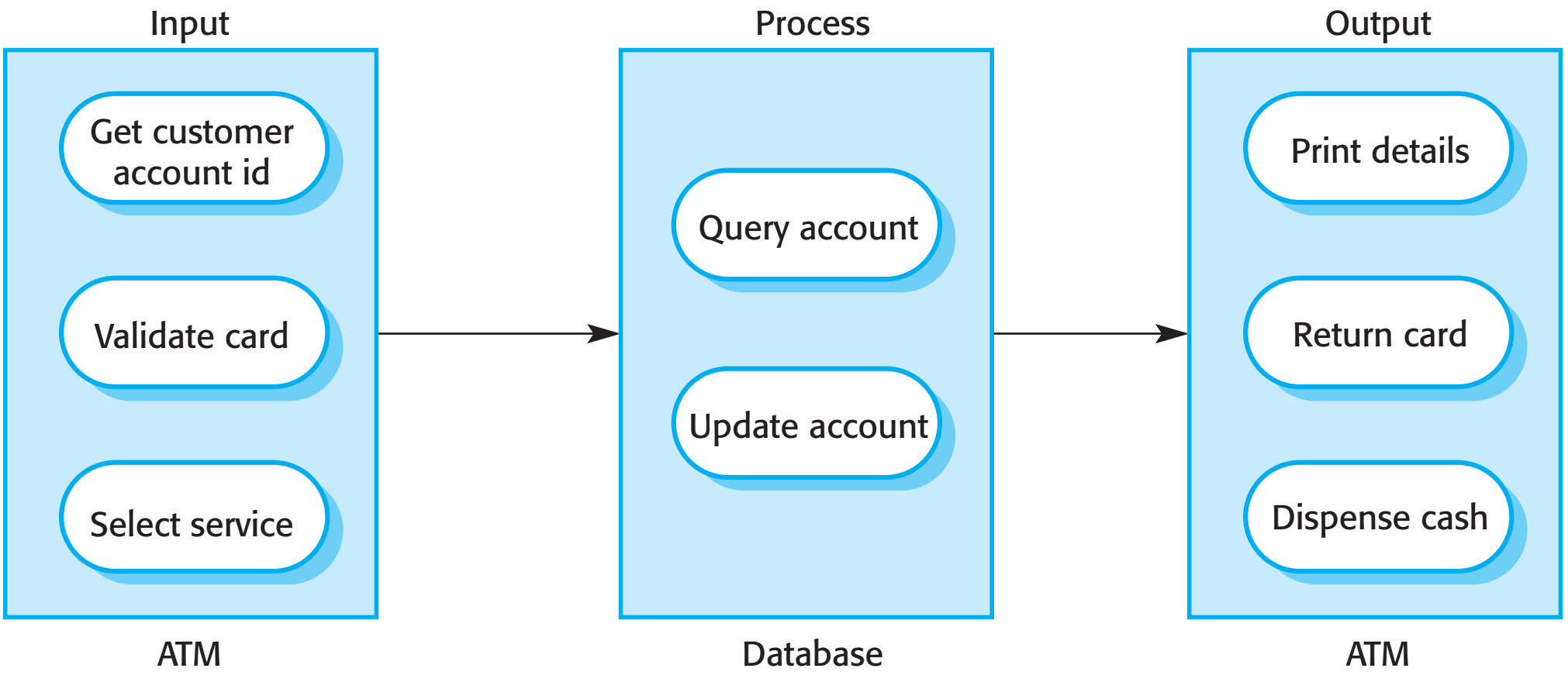
- As a starting point for architectural design.
- As a design checklist.
- As a way of organizing the work of the development team.
- As a means of assessing components for reuse.
- As a vocabulary for talking about application types

Transaction processing applications

- Transaction processing systems are designed to process user requests for information from a database, or requests to update a database.
- Technically, a database transaction is part of a sequence of operations and 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.
- Transaction processing systems are usually interactive systems in which users make asynchronous requests for service.

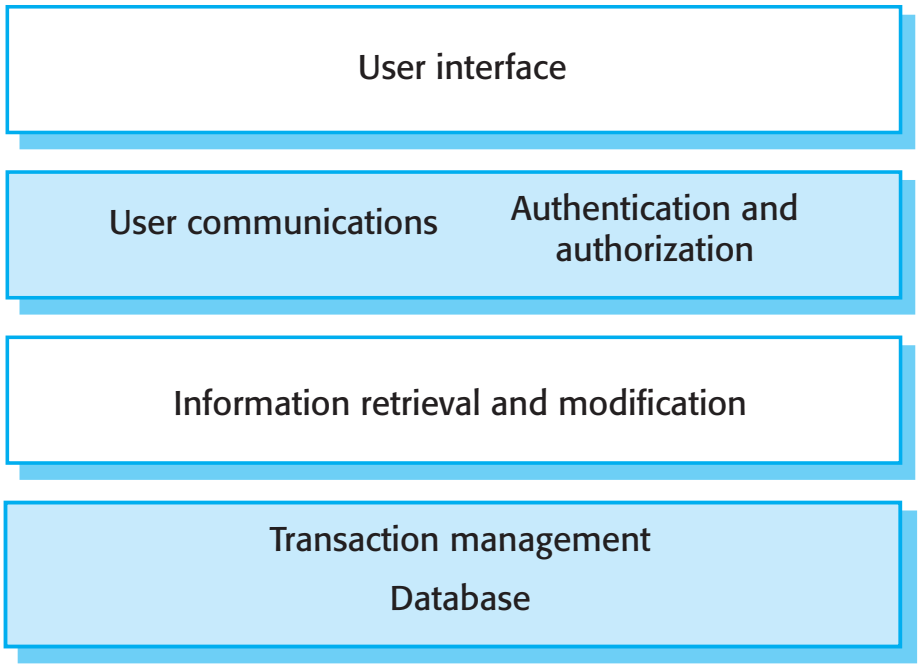


The software architecture of an ATM system

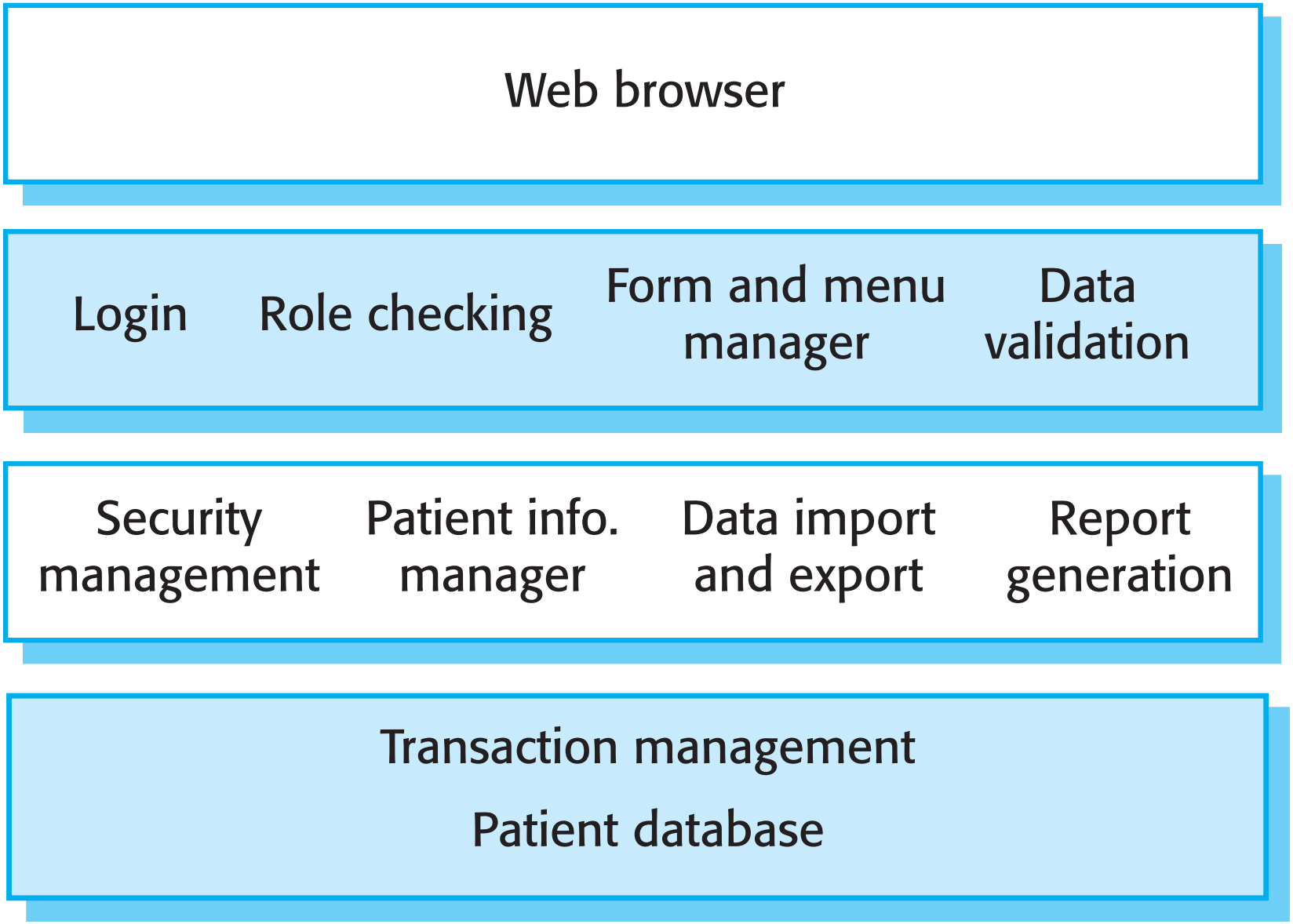


Information systems

- 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



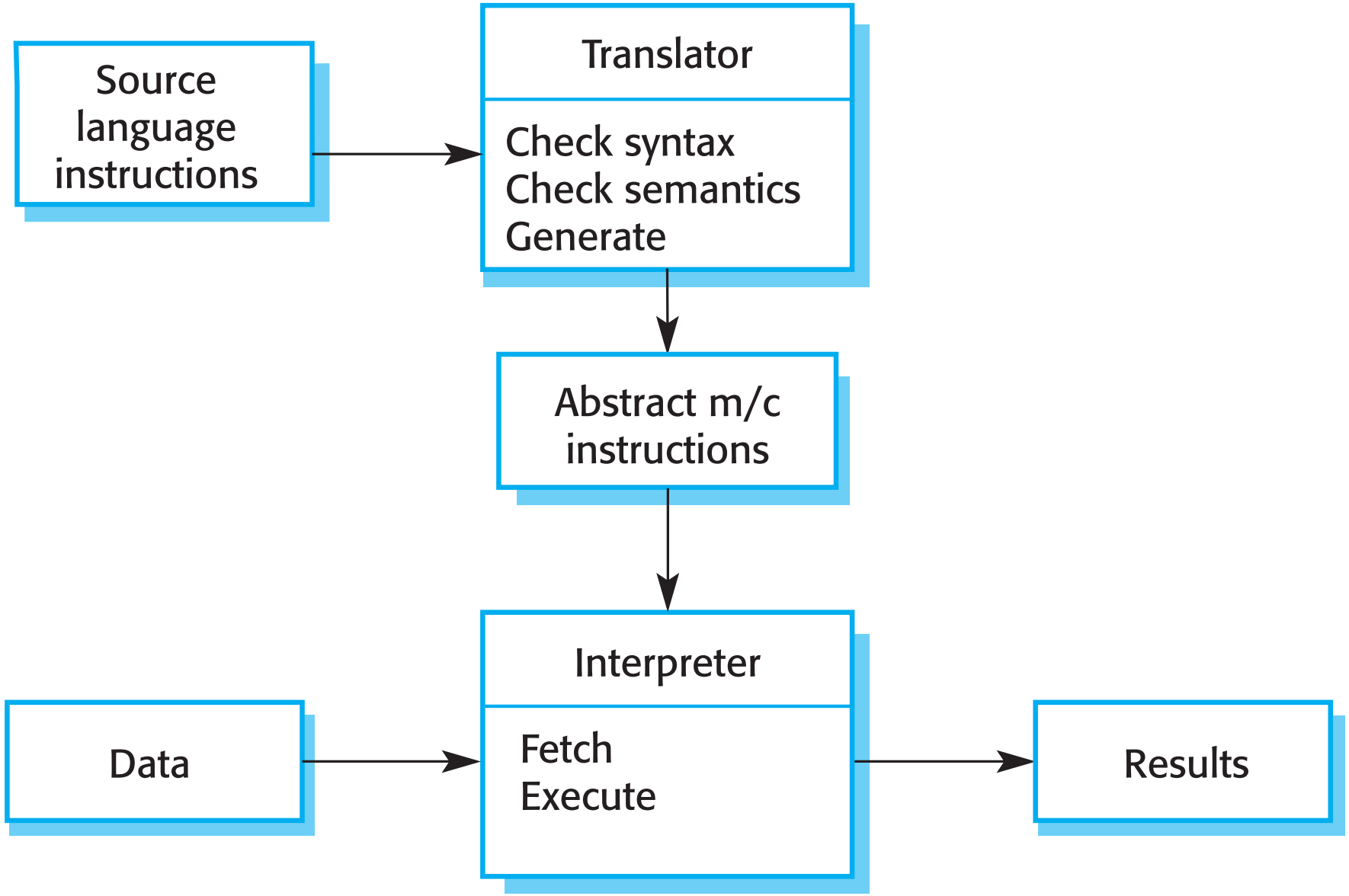
The architecture of the Mentcare system



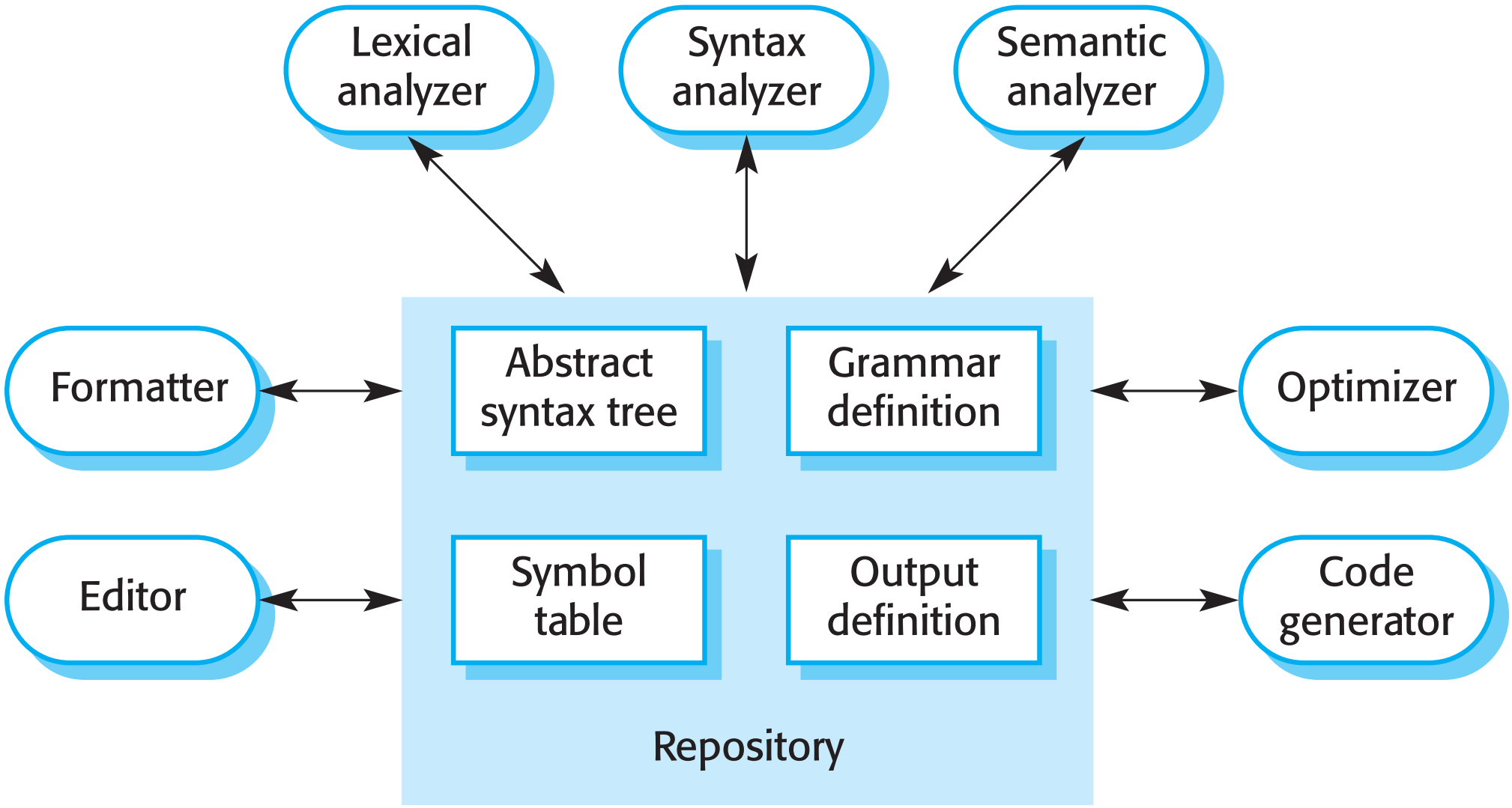
Language processing systems

- Language processing systems translate one language into an alternative representation of that language and, for programming languages, may also execute the resulting code.
  - Compilers translate a programming language into machine code.
  - Other language processing systems may translate an XML data description into commands to query a database or to an alternative XML representation.
  - Natural language processing systems may translate one natural language to another.

The architecture of a language processing system

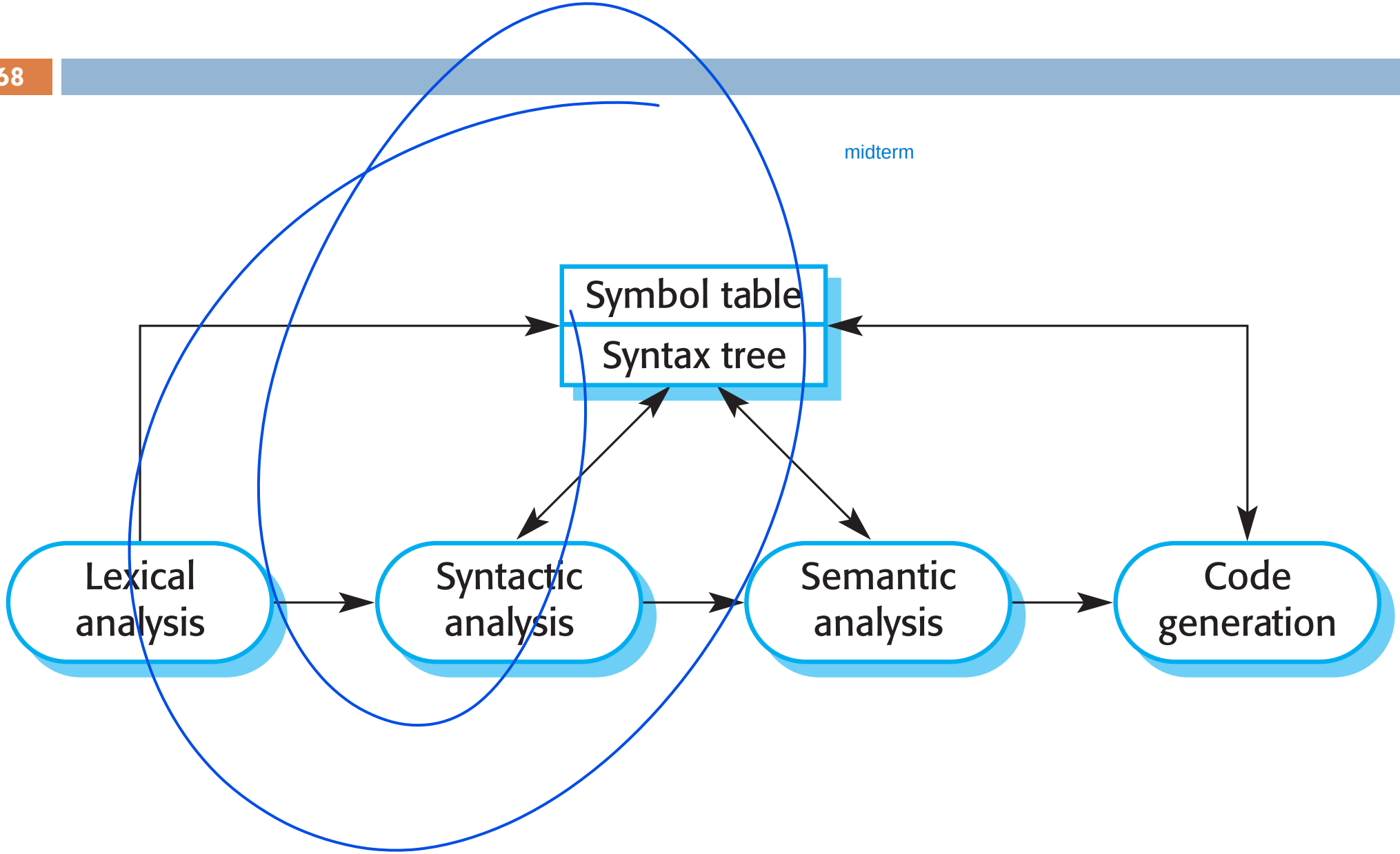


A repository architecture for a language processing system



A pipe and filter compiler architecture

midterm



Component-Level Design

Component-Level Design

- Component-level design occurs after the first iteration of architectural design has been completed.
- Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.
- The intent is to translate the design model into operational software.
- Component-level design bridges the gap between architectural design and coding.

Basic Design Principles

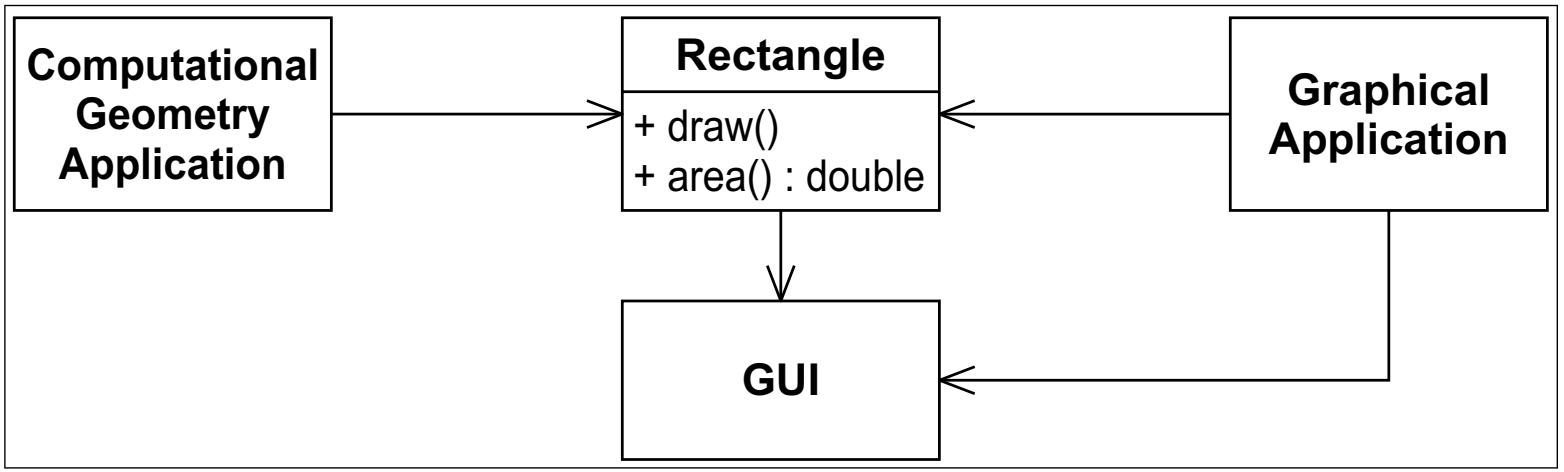
- **SOLID** Design
  - **S** : Single Responsibility Principle
  - **O**: Open-Closed Principle
  - **L** : Liskov Substitution Principle
  - **I** : Interface Segregation Principle
  - **D** : Dependency Inversion Principle

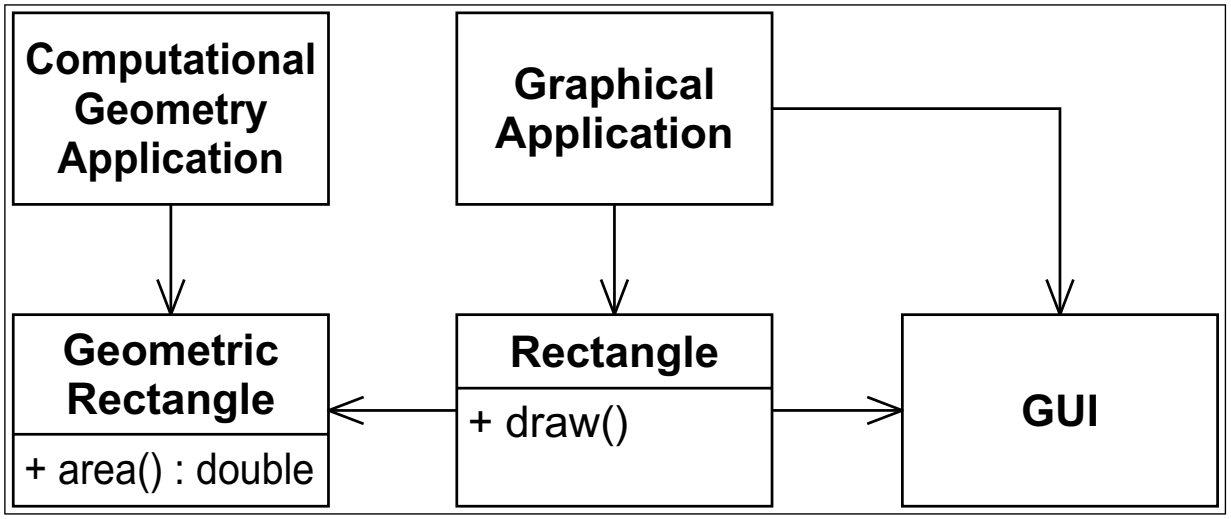
Single Responsibility Principle (SRP)

- A class should only have a single responsibility.
- If a class has more than one responsibility, then the responsibilities become coupled.
- Changes to one responsibility may impair or inhibit the ability of the class to meet the others.
- This kind of coupling leads to fragile designs that break in unexpected ways when changed.
- *Separation of concerns*

> *"A class should have one, and only one, reason to change."*
>
> By Robert C. Martin

One vs. Separated Responsibilities

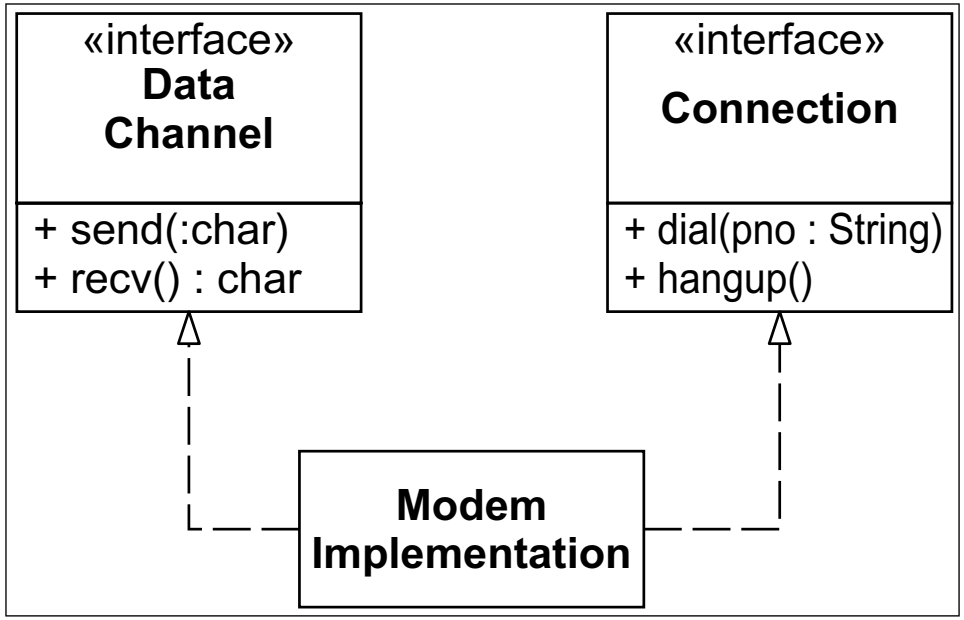




Rigidity vs. Needless Complexity

Modem.java -- SRP Violation

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```



Example in Python

Example in Python

Comply with SRP

- sry.py

Anti SRP (Violate SRP)

- srp_anti-pattern.py

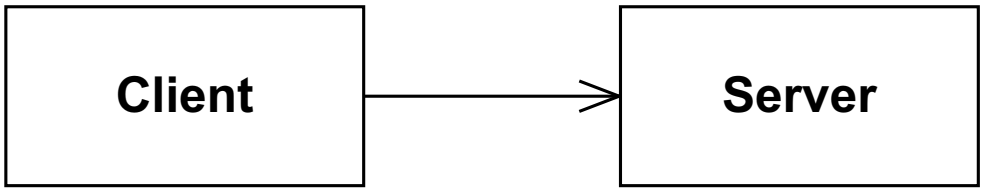
Open-Closed Principle (OCP)

"Software entities (classes, modules, functions, etc.) should be *open* for *extension*, but *closed* for *modification*."

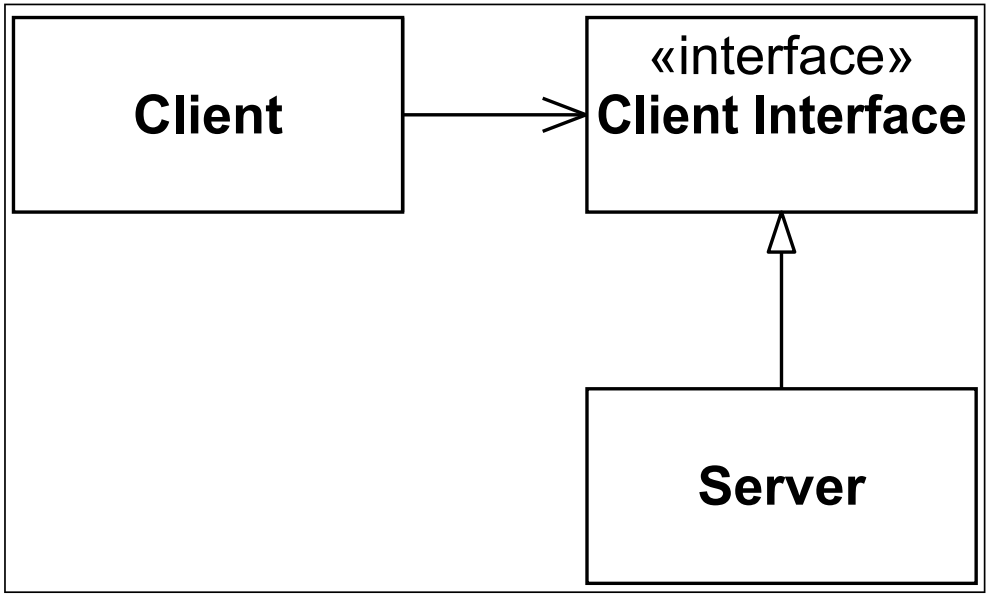
By Robert C. Martin

- "Open for extension."
  - This means that the behavior of the module can be extended.
  - As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes.
  - In other words, we are able to change what the module does.
- "Closed for modification."
  - Extending the behavior of a module does not result in changes to the source or binary code of the module.
  - The binary executable version of the module, whether in a linkable library, a DLL, or a Java .jar, remains untouched.

Using Abstraction to Gain Explicit Closure



Client is not open and closed



STRATEGY pattern: Client is both open and closed

```
//--shape.h-----------------------------------------
enum ShapeType {circle, square};
struct Shape
{
    ShapeType itsType;
};

//--circle.h----------------------------------------
struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};
void DrawCircle(struct Circle*);

//--square.h----------------------------------------
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};
void DrawSquare(struct Square*);

//--drawAllShapes.cc--------------------------------
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
                break;
            case circle:
                DrawCircle((struct Circle*)s);
                break;
        } }
}
```

```
class Shape {
  public:
    virtual void Draw() const = 0;
};
class Square : public Shape
{
  public:
    virtual void Draw() const;
};
class Circle : public Shape
{
  public:
    virtual void Draw() const;
};
void DrawAllShapes(vector<Shape*>& list)
{
  vector<Shape*>::iterator i;
  for (i=list.begin(); i != list.end(); i++)
    (*i)->Draw();
}
```

Open-Closed Principle (OCP)

- Anticipation and "Natural" Structure
  - There is no model that is natural to all contexts!
  - guess and construct abstractions -> expensive
- Putting the "Hooks" In
  - Fool Me Once...
    - We take the first bullet, and then we make sure we are protected from any more bullets coming from that gun.
  - Stimulating Change.

Example in Python

Example in Python

Comply with OCP

- ocp.py

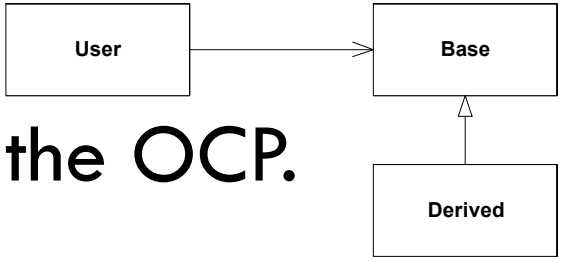
Anti OCP (Violate OCP)

- ocp_anti-pattern.py

Liskov Substitution Principle (LSP)

> *"Subtypes must be substitutable for their base types."*
>
> By Robert C. Martin

- Objects in a program should be able to be replaced with instances of their subtypes without altering the correctness of that program.
- In other words, a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.
- The LSP is one of the prime enablers of the OCP.



Example in Python

Example in Python

Anti LSP (Violate LSP)

•lsp_anti-pattern.py

Solution:

• Don’t implement any stricter validation rules on input parameters than implemented by the parent class.

• Apply at the least the same rules to all output parameters as applied by the parent class

IS-A

- IS-A is about Behavior
  - Behaviorally, a Square is not a Rectangle, and it is *behavior* that software is really all about.
  - the IS-A relationship pertains to behavior that can be reasonably assumed.

Design by Contract

- Design by Contract (DbD)
  - the author of a class explicitly states the contract for that class.
  - The contract informs the author of any client code about the behaviors that can be relied on.
  - The contract is specified by declaring preconditions and postconditions for each method.

"A routine redeclaration [in a derivative] may only replace the original *precondition* by one *equal* or *weaker*, and the original *postcondition* by one *equal* or *stronger*"

By Bertrand Meyer

Contracts in Unit Tests

- Specifying Contracts in Unit Tests
  - Contracts can also be specified by writing unit tests.
  - By thoroughly testing the behavior of a class, the unit tests make the behavior of the class clear.
  - Authors of client code will want to review the unit tests so that they know what to reasonably assume about the classes they are using.

Interface Segregation Principle (ISP)

> "Many *client-specific* interfaces are better than One *general-purpose* interface."
>
> By Robert C. Martin

- This principle deals with the disadvantages of "fat" interfaces.
  - Classes that have "fat" interfaces are classes whose interfaces are not cohesive.
  - In other words, the interfaces of the class can be broken up into groups of methods.
  - Each group serves a different set of clients.
  - Thus, some clients use one group of member functions, and other clients use the other groups.
- Clients should not be forced to depend on methods that they do not use.

Example in Python

Example in Python

Comply with ISP

- isp.py

Anti ISP (Violate ISP)

- isp_anti-pattern.py

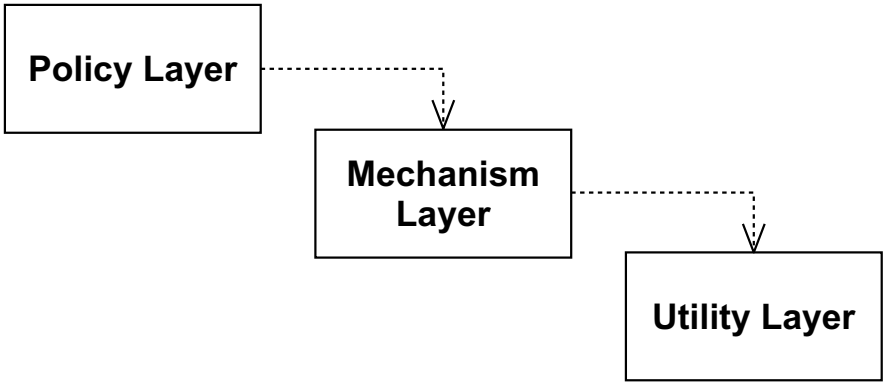
Dependency Inversion Principle (DIP)

"a. High-level modules should not depend on low-level modules.
Both should **depend on abstractions.**"
"b. Abstractions should not depend on details.
Details should depend on abstractions."

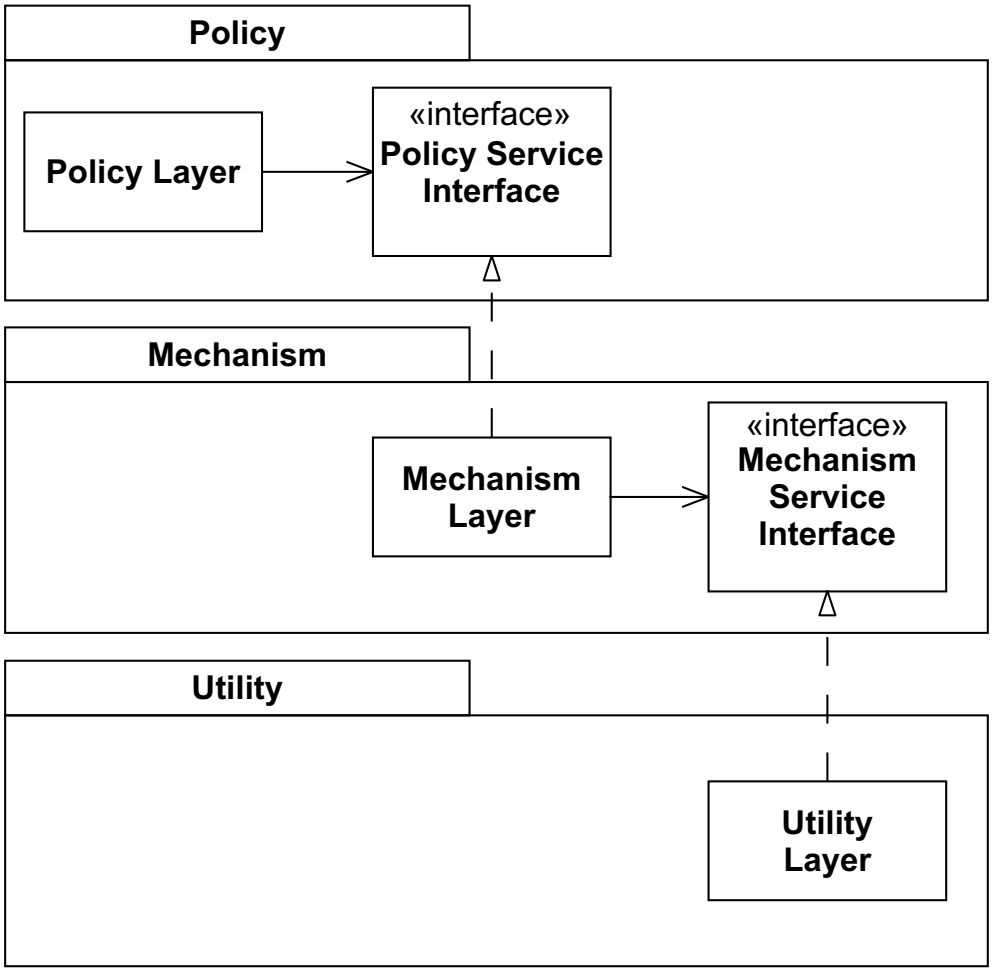
By Robert C. Martin

- Depend on abstractions.
- Do not depend on concretions.

Dependency Inversion Principle (DIP)



Naive layering scheme



Inverted Layers

Example in Python

Example in Python

Comply with DIP

- dip.py

Anti DIP (Violate DIP)

- dip_anti-pattern.py