ERIK JONSSON SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
The University of Texas at Dallas

UT DALLAS

CE/CS/SE 3354 Software Engineering

Software Architecture Design

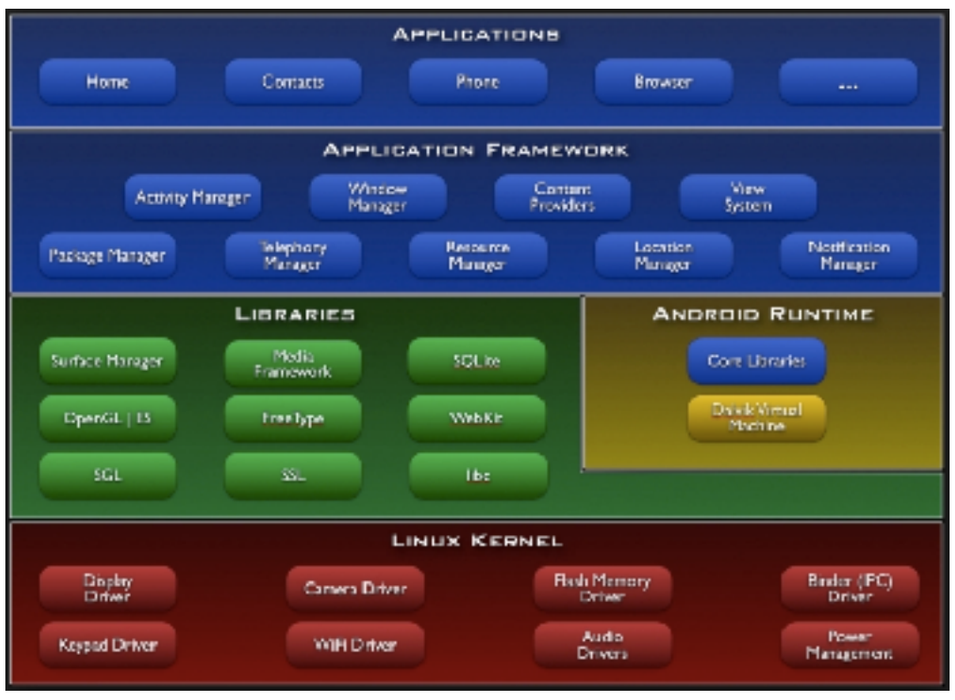
Software Design

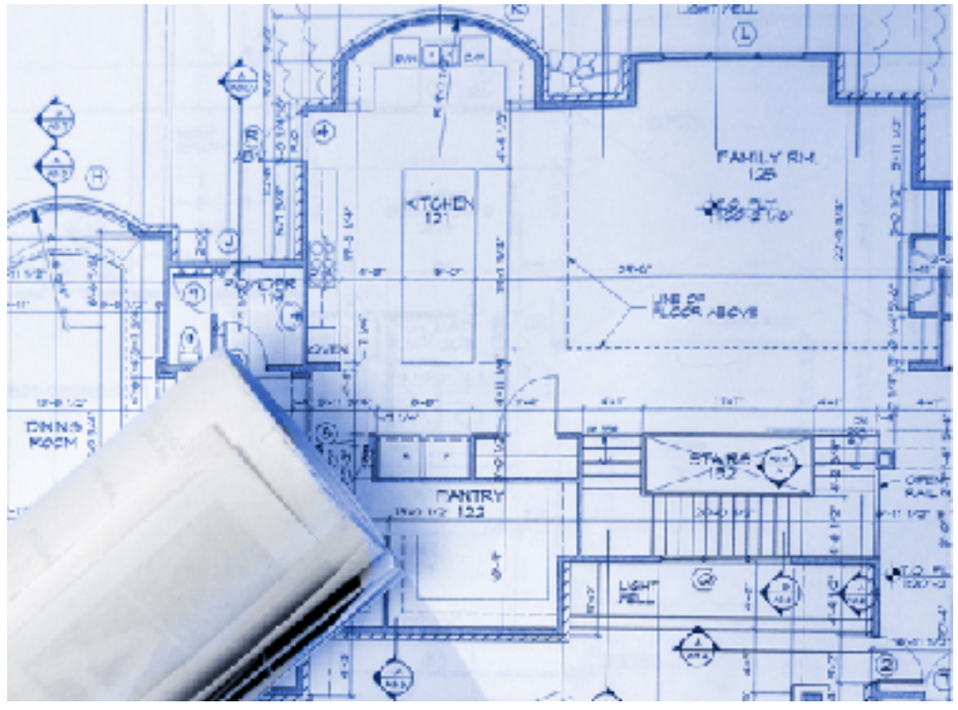
- Software design transforms software requirements specification into a description of the solution
  - Architecture design
  - Detail design

    Algorithms, data structures, etc
- Software design should satisfy all the requirements
  - Functional requirements
  - Nonfunctional requirements
- Software design process can be iterative

Software Architecture

- Software architecture is the structure of a software system – like the blue prints in building architecture





Software Architecture: Abstraction

- Software architecture abstracts the system into:
  - Software components
    - Details (data structure and algorithms) hidden
    - Services (how they use, are used by, relate to, and interact with other component) displayed
  - Relationships among the components
    - Data flows
    - Control flows

Abstraction - I

- One characterization of progress in software development has been the regular increase in abstraction levels, i.e., the conceptual size of software designer's building blocks

Abstraction - II

- 1950s: software was written in machine language
  - Instructions and data were placed individually and explicitly in the computer's memory
  - Machine code programming problems were solved by assembly language via adding a level of abstraction between the program and the machine

Abstraction - III

- Late 1950s: the emerging of the first high-level programming languages
  - Well understood patterns are created from notations that are more like mathematics than machine code

    evaluation of arithmetic expressions

    procedure invocation

    loops and conditionals

    followed with higher-levels of abstraction for representing data (types)

Abstraction - IV

- Late 1960s and 1970s: programmers shared an intuition that good data structure design will ease the development of a program
- This intuition was converted into theories of modularization and information hiding
  - Data and related code are encapsulated into modules
  - Interfaces to modules are made explicit

Software Architecture Issues

- Organization and global control structure
- Protocols of communication, synchronization, and data access
- Assignment of functionality to design elements
- Physical distribution
- Composition of design elements
- Scaling and performance
- Selection among design alternatives

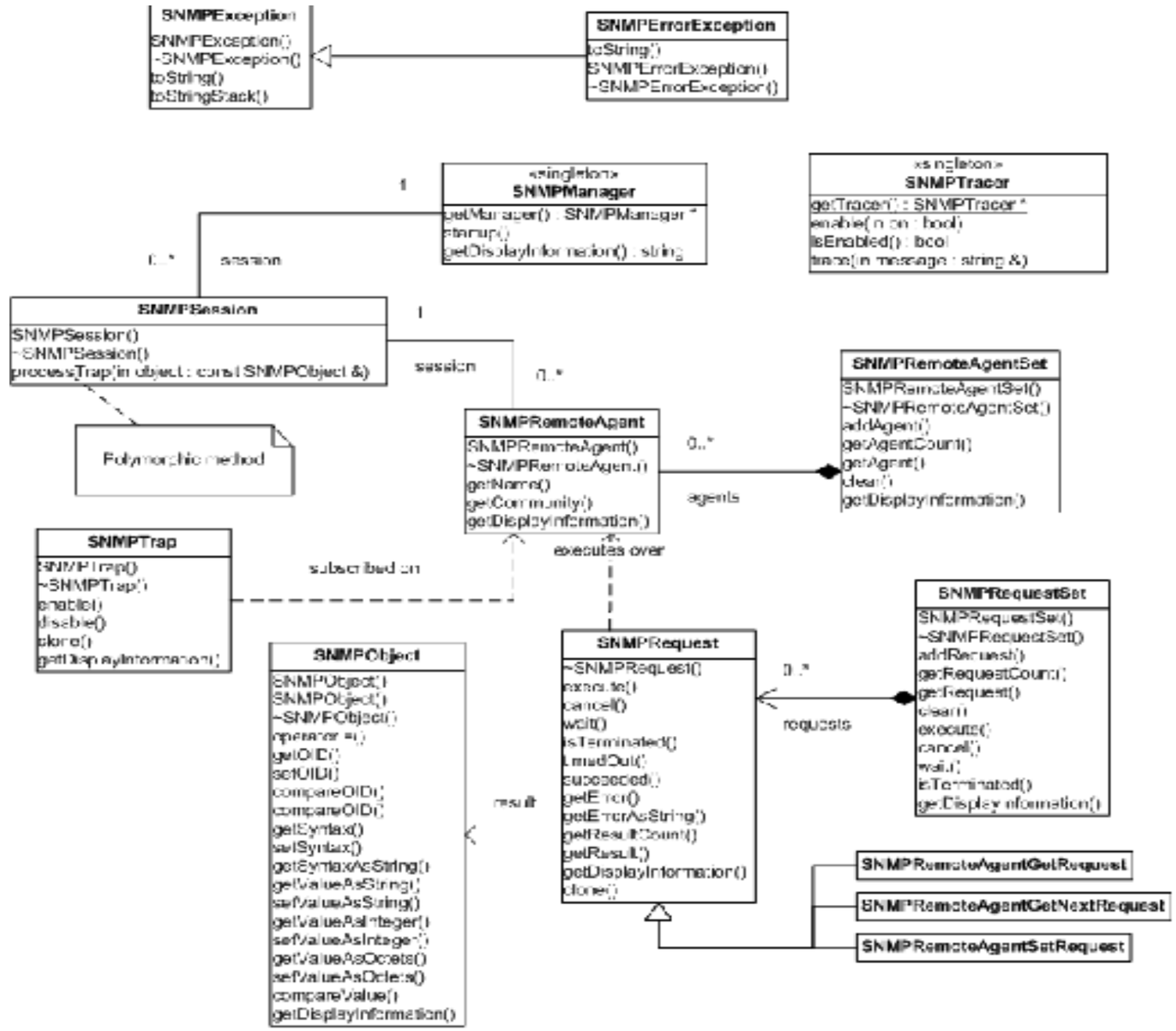
This Class

- Software Architecture
  - What?
  - Why?
  - How?

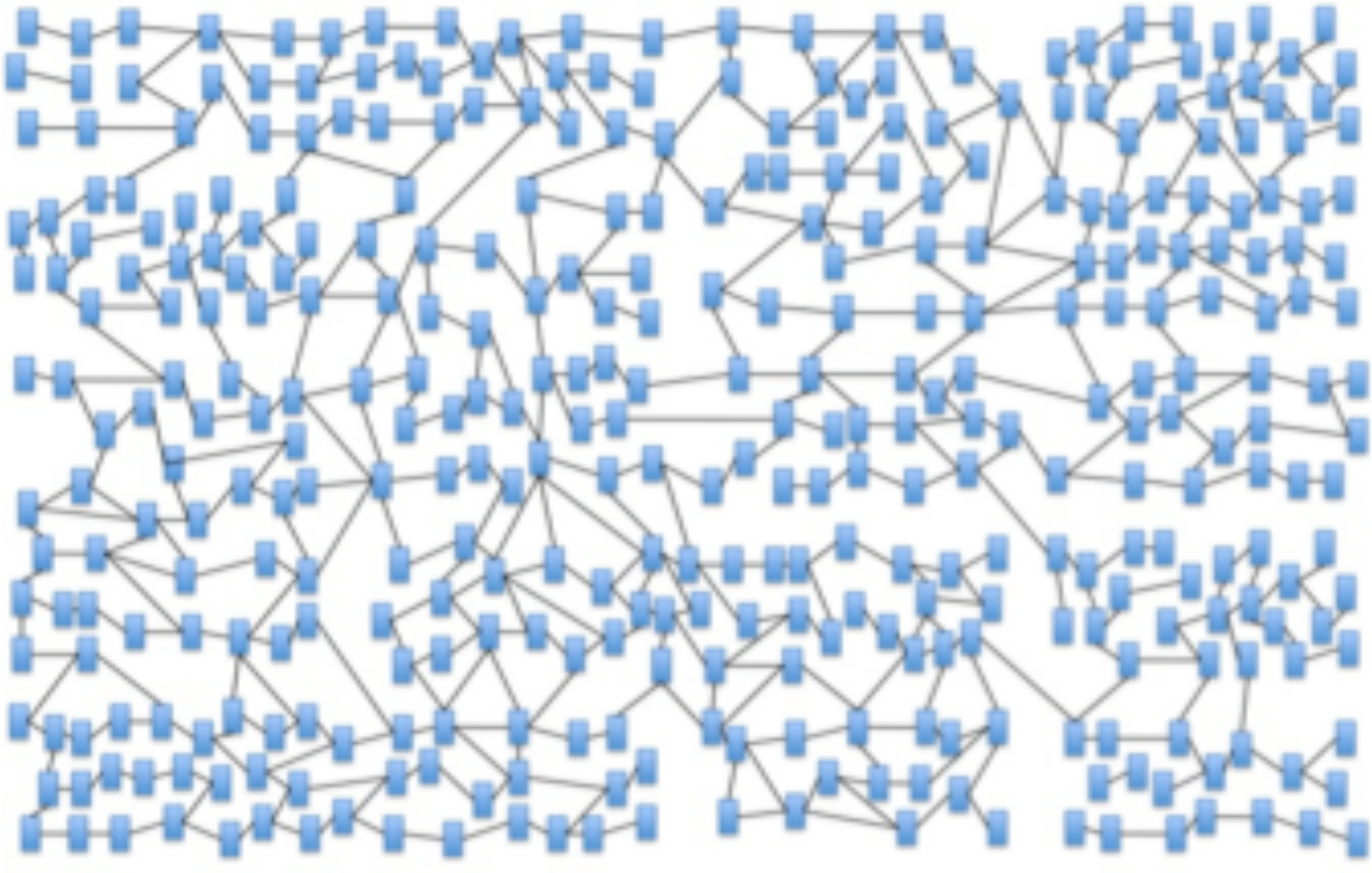
Why software architecture?

- Software becomes larger and larger
  - 50 lines of code: just write it
  - 5,000 lines of code: maybe class diagram
  - 500,000 lines of code: software architecture

A simple class diagram



A class diagram with 300 classes



So we need higher level abstraction

- Consider higher level concepts in architecture : components
- For OO approach, components are usually a collection of classes
- Consider the bookstore project:
  - For a small book store: books, shelves, clerks
  - But for barnes&noble: book stores, ads, finance, online stores, readers

  Bookstore is one component

Software Architectural Design

- Usually used for large software projects and software frameworks
- Early phase design
  - Important for task assignment
- At a higher level than class diagrams
  - Components and their relationship
  - Interface can be vague at first

Software Architectural Design

- Usually done by a small number of high-level people
- It can be both technical and artistic
- Experience and creativity can be major factors

- It is not something you can learn well from the book
- But there is something to learn about it

Software Architecture vs. Building Architecture

- Software architecture is similar to architecture styles for buildings
  - Certain way to organize components and their relationships
  - Can be reused in different software
  - Use different architecture styles according to the usages of software

Architecture Styles

Stadium

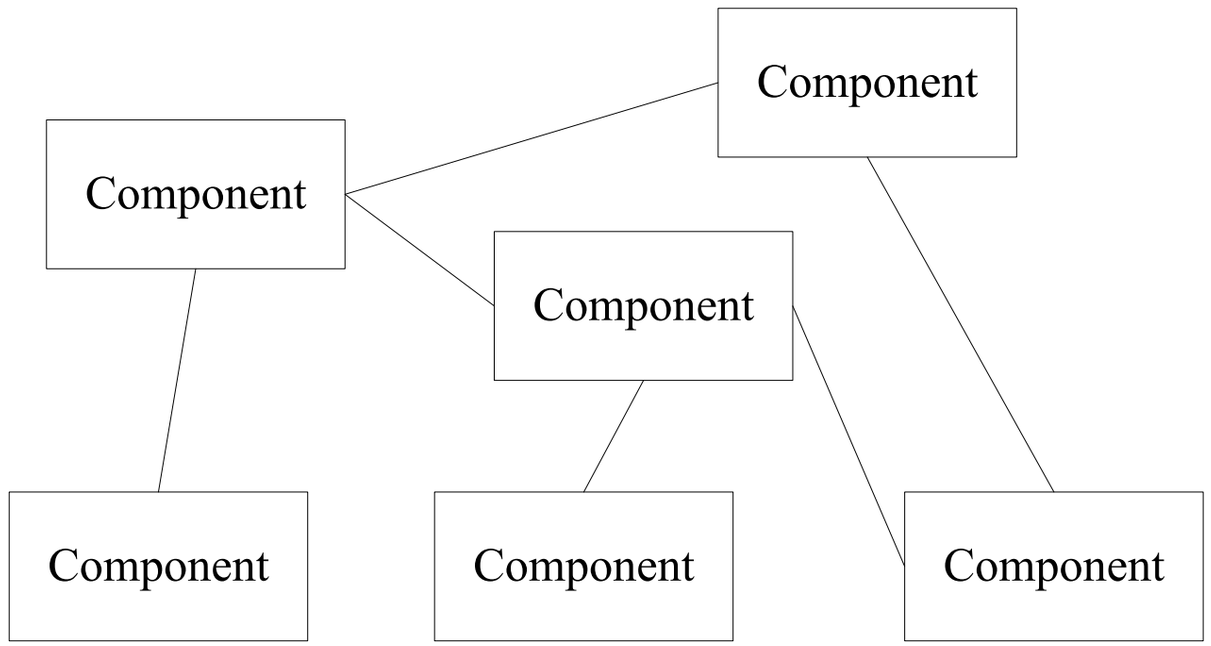
Architecture Styles

Skyscrapers

Some times people try new styles…

Why Software Architecture Styles

- Easier to do high level design
  - Barnes&Noble example
  - Consider an architecture like this:



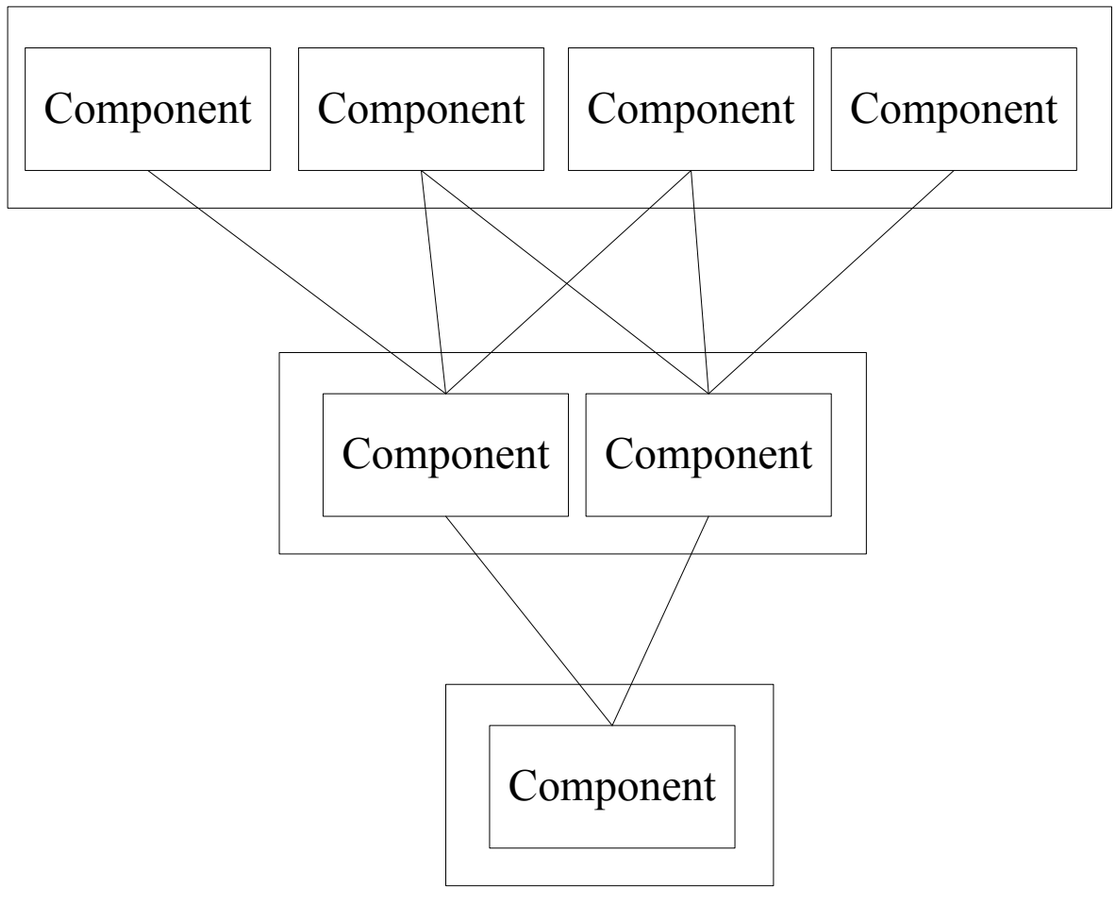
Why Software Architecture Styles

- Actually, most such information systems has an architecture like this.

Application

Service

Data



Why Software Architecture Styles

- Reduce high level design risks
  - People may have tried different styles and found this best (It is not as obvious as for buildings)
  - The drawbacks of commonly used architectures are well studied
- You may try new ones:
  - But not until you are a really good architect

This Class

- Software Architecture
  - What?
  - Why?
  - How?

Software Architecture Styles

- An architectural style defines a family of systems in terms of a pattern of structural organization
- It determines:
  - The vocabulary of components and connectors that can be used in instances of that style
  - A set of constraints on how they can be combined
    For example, one might constrain:

    The topology of the descriptions, e.g., no cycles

    Execution semantics, e.g., processes execute in parallel

Software Architecture Styles

- Software architectures are represented as graphs
- Nodes represent components:
  - procedures, modules, processes, tools, databases, etc
- Edges represent connectors:
  - procedure calls
  - event broadcasts
  - database queries
  - pipes

Popular architecture styles

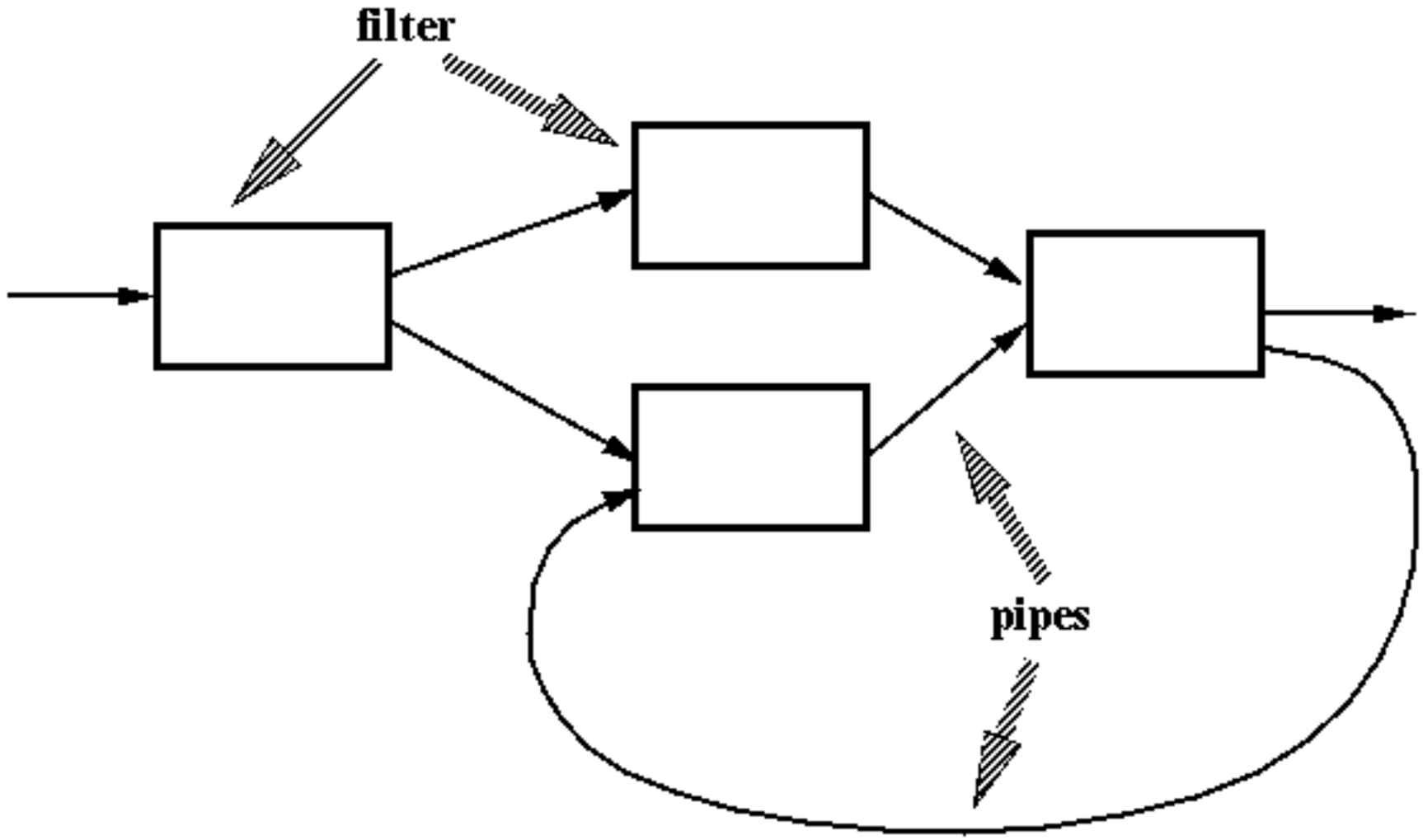
- Pipe and Filter
- Layered
- Model-View-Controller (MVC)
- Repository

Pipe and Filter

- Also denoted as data-flow style
  - A defined series of independent computations
  - Performed for data transformation

- A component reads streams of data on its inputs and produces streams of data on its outputs

Pipe and Filter



Pipe and Filter: Structure

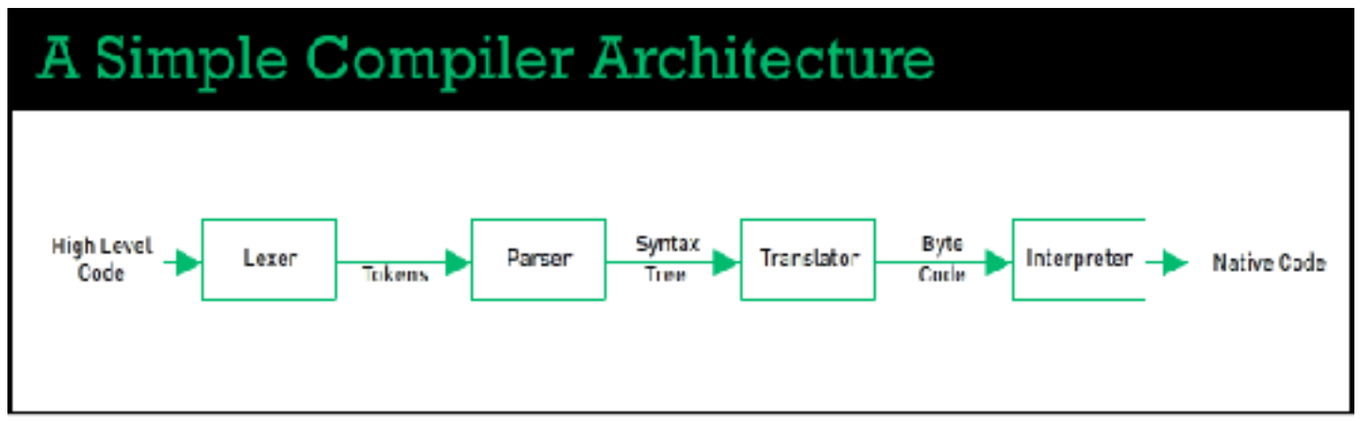
- Components: filter
  - Perform changes on the data input and generate output
  - Computation can be done with part of input, so that it can start before accepting all inputs
- Connectors: pipe
  - Simply data transfer between filters

Pipe and Filter: Characteristics

- Filters do not share state with other filters
- Filters do not know the identity of their upstream or downstream filters

Pipe and Filter Examples

- Unix shell scripts: provides a notation for connecting Unix processes via pipes.
  - e.g., cat file | grep err | wc
- Compilers: the phases in the pipeline include:
  - Lexical analysis
  - Parsing
  - Semantic analysis
  - Code generation



Pipe and Filter - Advantages

- Easy to understand
  - The overall input/output behavior of a system is a simple composition of the behaviors of individual filters
- Support reuse
  - Since any two filters can be hooked together, provided they agree on the data being transmitted between them
- Systems can be easily maintained and enhanced
  - Since new filters can be added to existing systems and old filters can be replaced by improved ones
- Support concurrent execution

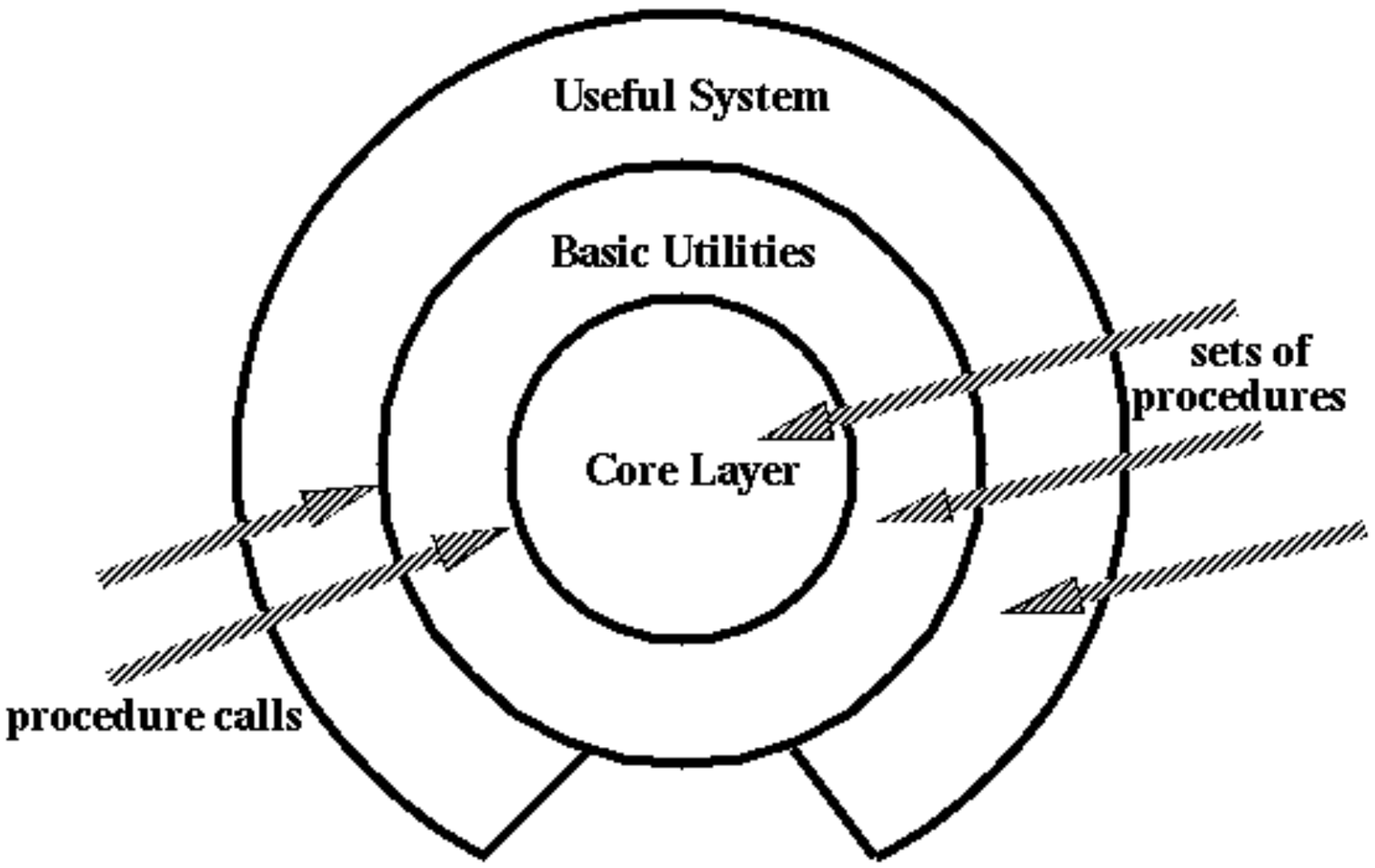
Pipe and Filter - Disadvantages

- Not good for handling reactive systems, because of their interactive event triggering
- Excessive parsing and un-parsing leads to loss of performance and increased complexity in writing the filters themselves

Layered Style

- The system is divided to multiple layers
- Each layer provides service to the layer above it and serves as a client to the layer below it
- Each layer may have multiple components, these components usually do not interact much with each other

Layered Style



Layered Style: Structure

- Components: are typically collections of procedures
- Connectors: the layer margin, are typically procedure calls under restricted visibility
- Layer Communication Rules:
  - Usually a component will talk only with the layer just beneath it
  - Only carefully selected procedures from the inner layers are made available (exported) to their adjacent outer layer

Layered Style: Examples

- Operating Systems
  - Unix
  - Windows
  - Android
  - …(almost any)
- Distributed Information Systems

Layered Style: Advantages

- Design: based on increasing levels of abstraction
- Security: inner layers are usually not directly accessed by outmost layers
- Maintainability: changes to the function of one layer affects at most two other layers
- Reuse: different implementations (with identical interfaces) of the same layer can be used interchangeably

Layered Style: Disadvantages

- There might be a negative impact on the performance as we have the extra overhead of passing through layers instead of calling a component directly
- The use of layers helps to control and encapsulate the complexity of large applications, but adds complexity to simple applications
- Changes to lower level interfaces tend to percolate to higher levels

Model-View-Controller (MVC) Pattern

- Context
  - user interface (UI) most frequently accessed and changed component of an interactive application
  - users frequently like to have multiple perspectives (views) into the system
- Problem
  - in a large complex system, separate the UI design from the rest of the system
  - allow changes to the interface with minimal impact on the rest of the system

Model-View-Controller (MVC) Pattern

- Solution
  - MVC pattern separates application functionality into three kinds of components

Model – internal state of the application

View – external representation of the model

Controller – coordinates updates of the view in response to user input or model changes

MVC Elements

- Model
  - interacts with the data model of the application
  - notifies the view when the state is updated allowing the view to change
- View
  - presentation layer for the application
  - gets information from the model to update the presentation
- Controller
  - defines the way UI reacts to user input
  - sends commands to the view to change the view's presentation
  - sends commands to the model to update the state of the model

Model-View-Controller (MVC) Pattern

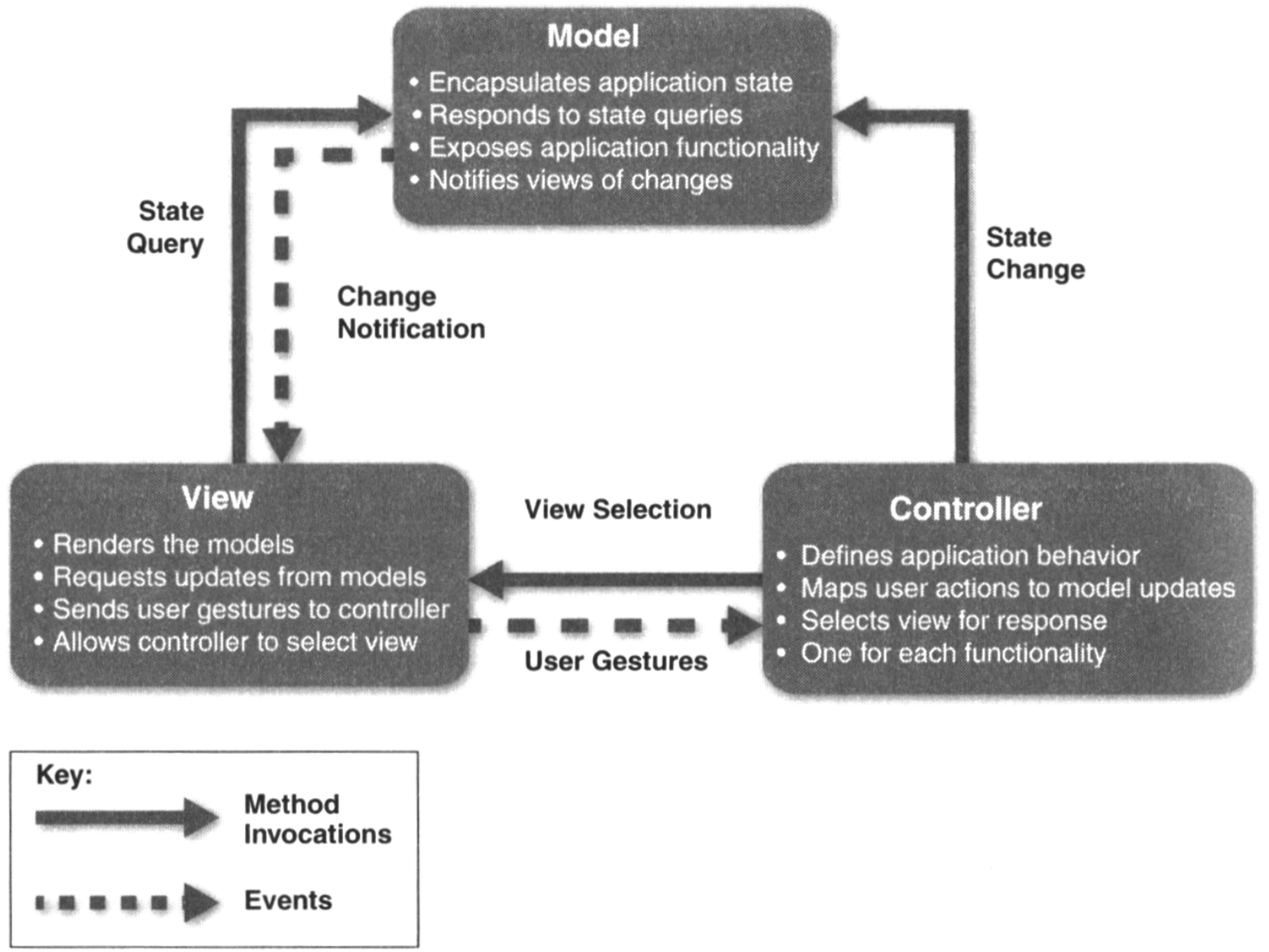


FIGURE 13.7 The model-view-controller pattern

MVC Examples

- Web applications
- Game development

MVC: Advantages

- Increase flexibility and reuse
  - separation of concerns
- Easily incorporate multiple views
- Promotes testability through defined interfaces

MVC: Disadvantages

- Some fundamental complexity
  - perhaps too complex for simple applications
- Variance of the pattern can be substantial

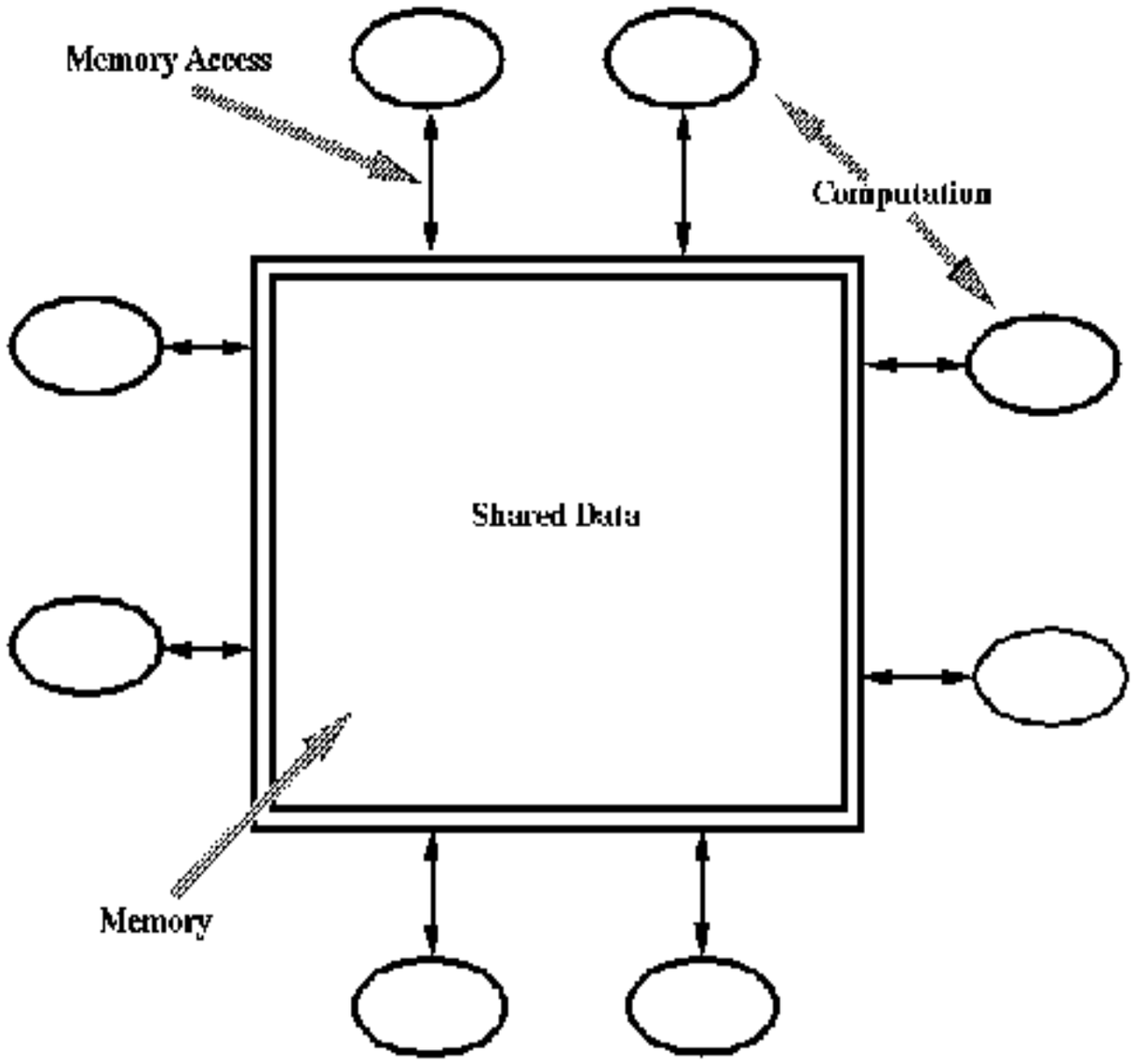
Repository Style

- Also denoted as data-centered style
  - Suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information

- Typically the information must be manipulated in a variety of ways

- Often long-term persistence of information is required

Repository Style



Repository Style: Structure

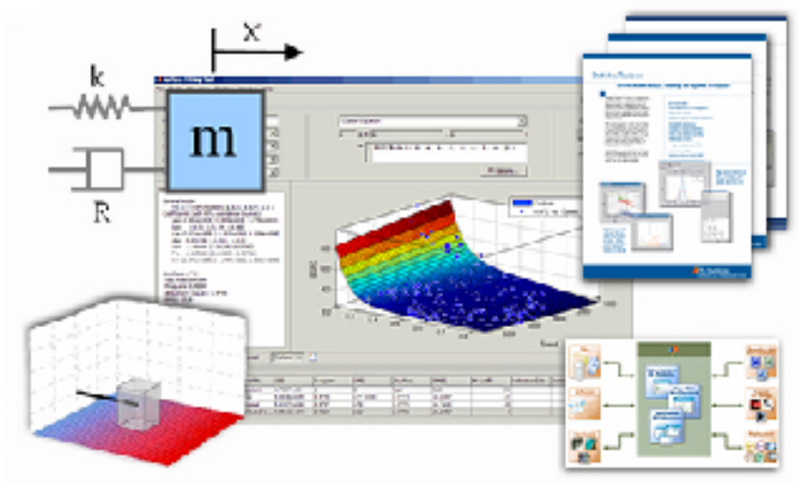
- Components:
  - A central data structure representing the correct state of the system
  - A collection of independent components that operate on the central data structure
- Connectors:
  - Typically procedure calls or direct memory accesses

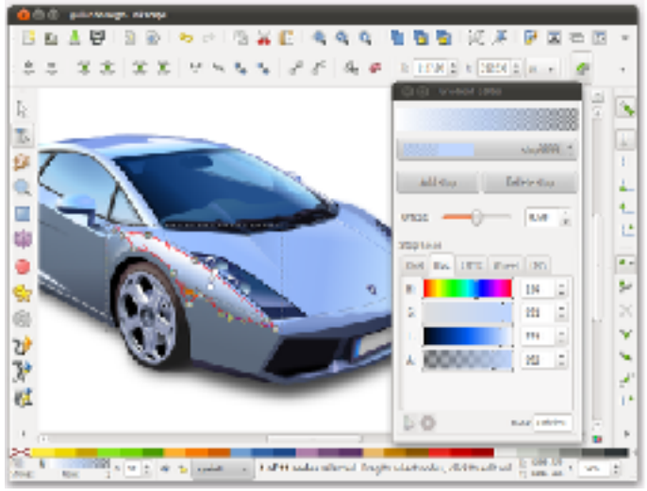
Repository Style: Characteristics

- Components only interact with the repository
- All components can access the repository
- The repository controls and manages the access to its data

Repository Style Examples

- Programming Environments
- Graphical Editors
- AI Knowledge Bases





Repository Style: Advantages

- Efficient way to store large amounts of data
- Centralized management:
  - backup
  - security
  - concurrency control

Repository Style: Disadvantages

- Must agree on a data model first
- Difficult to distribute data
- Data evolution is expensive

Software Architecture Style Summary

- We can understand what a style is by answering the following questions:
  - What is the structural pattern?
  - What is the underlying computational model?
  - What are some common examples of its use?
  - What are the pros and cons of using that style?
  - ...