Advanced Software Design
Architecture & Patterns

Gary Stewart
gstewart@cs.uct.ac.za
slides taken from prof. edwin blake

ARCHITECTURE CENTRED APPROACH

Architecture Centred Approach

Introducing Patterns

Key Architecture Patterns

Layered Architecture

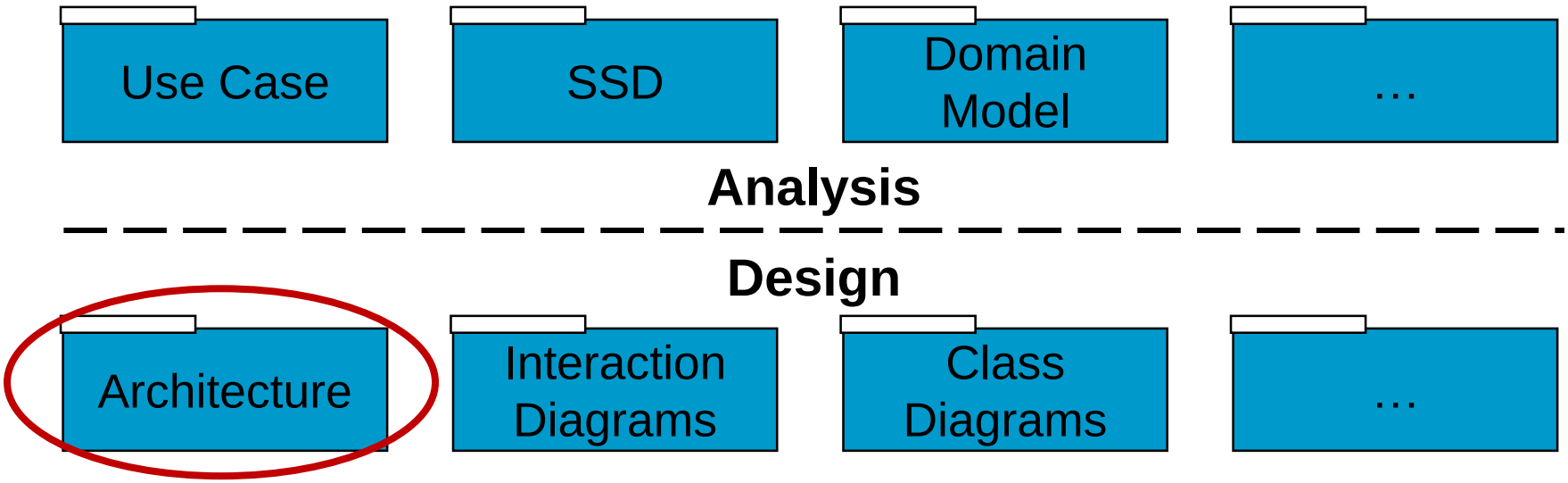
Patterns and Concurrency

Conclusion

Architecture Centred Approach

- We place an emphasis on design
  - Design pervades engineering activities
  - Important aspect of Computer Science
- An incorrect interpretation of the Agile development leads to inadequate levels of architectural design information
- Architecture Decisions are system-wide Strategic Decisions
  - Choice of programming paradigm, e.g. Object-Orientation
  - Choose overall architectural pattern (see later)
  - Choice of API(s)
- Local detailed tactical decisions e.g. algorithms are left for later

Requirements and Analysis leads to Design



- At first we are investigators learning about a problem
- Then we change roles and become designers producing a working solution
- (and then we change back iteratively!)

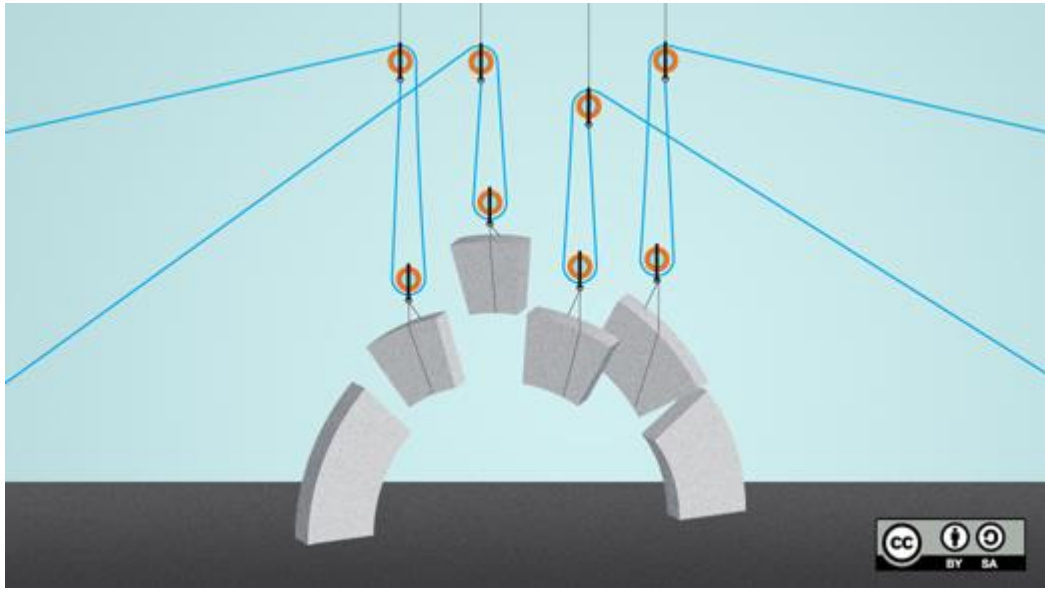
What is a Software Architecture then?

- Software architecture ⇒ *large scale*
  - big ideas
    - motivations
    - constraints
    - organization
    - patterns
    - responsibilities
    - connections
  - Very interested in the non-functional requirements
    - also called quality requirements
    - e.g. security, persistence, usability, etc.
    - ensure that these can be met
    - or if cannot be met, suggest trade- offs...
    - For example: trends vs cost vs time vs usability vs security.

Architecture is the Important Stuff!

1. Expert developers shared understanding of the system design
2. The set of design decisions that must be made early on

- Decisions that are hard to change
- The important stuff
  - Whatever that is!



INTRODUCING PATTERNS

Introducing Patterns

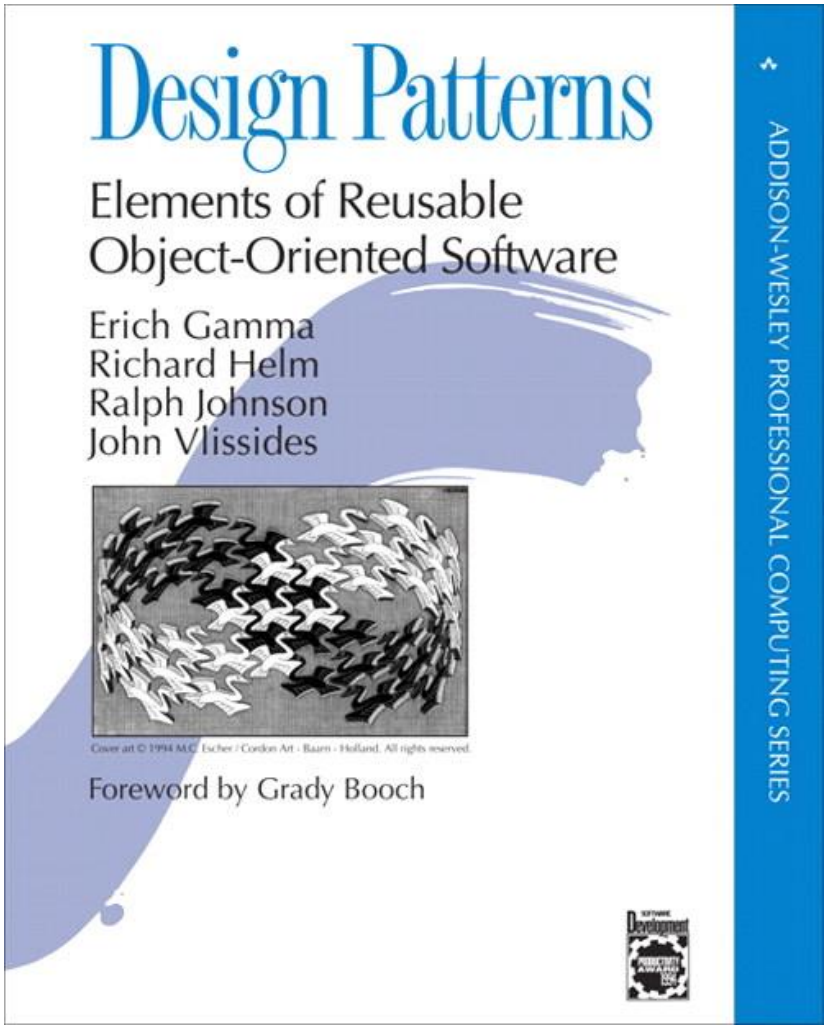
- Think about your own programming style.
  - A large portion of it is most likely the way you structure your solutions, rather than the minute details of your code.
  - If you could reuse the structure from one project to the next, you would gain all of the benefits of reuse.
- IT designer community has a growing body of past computer solutions for ideas and inspiration.
- Best described using patterns.

Patterns are Encapsulated Experience

- Software design does not begin with an empty slate.
- A new design is based on experience of previous “similar” designs.
- Software Design Patterns attempt to guide a new design with insight into typical problems.
- They attempt to encapsulate the basic approaches for similar types of problems.

Patterns applied to software development

- The idea of named patterns for software originated with Kent Beck (him of eXtreme Programming) in the mid 1980s.
- Software patterns really popularized with seminal book:
  - Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides
    - Frequently referred to as the Gang of Four or just *GoF*



Original Idea is from (real) Architecture

- Christopher Alexander, an architect, captured solutions to recurring problems.
  - These problems and solutions were described as patterns
- “Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice.”
- “Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.”
- Definition: A pattern is a solution to a problem in a context.
- OK, but apart from the obvious difference in domain of application, what is the difference between Architecture of buildings and of Software?

What's different about software architecture

- Software is not embedded in space
  - Often no constraining physical laws
  - Software is (infinitely) malleable
- Software has no obvious representation
  - E.g., no familiar geometric shapes
- Software does stuff (it is active)

Aims of Software Patterns

- The aim is to enhance reusability of object-oriented code
  - Well-structured object-oriented systems have recurring patterns of classes and objects
  - Knowledge of the patterns that have worked in the past allows:
  - a designer to be more productive and
  - the resulting designs to be more flexible and reusable.

Benefits of all Design Patterns

- Capture expertise and make it accessible to non-experts in an encapsulated design pattern
- Help communication amongst developers by providing a common language
  - Improve design understandability
- Make it easier to reuse successful designs and avoid alternatives that diminish reusability
- Facilitate design modifications
  - The design is more easily understood
- Improve design documentation
  - The system documentation starts with the UML design pattern

Definition: Architectural Patterns

An *architectural pattern* is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears

Architecture Model

- An artefact documenting some or all of the architectural design decisions about a system

Architecture Visualization

- A way of depicting some or all of the architectural design decisions about a system to a stakeholder

Architecture View

- A subset of related architectural design decisions

KEY ARCHITECTURE PATTERNS

Resources

Sommerville:

- Chapter 6 “Architectural Design” ← important.
  - Architectural patterns for control
  - Application architectures
  - Reference architectures

} Additional online material

- Chapter 13 “Dependability Engineering”
- Chapter 20 “Embedded Systems”

} Not in this course

- Chapter 28: Application Architectures (online only)

Bennett, McRobb & Farmer:

- Chapter 13 “System Design and Architecture”
- Chapter 15 “Design Patterns”

Resources II

Partha Kuchana, *Software Architecture Design Patterns in Java.*

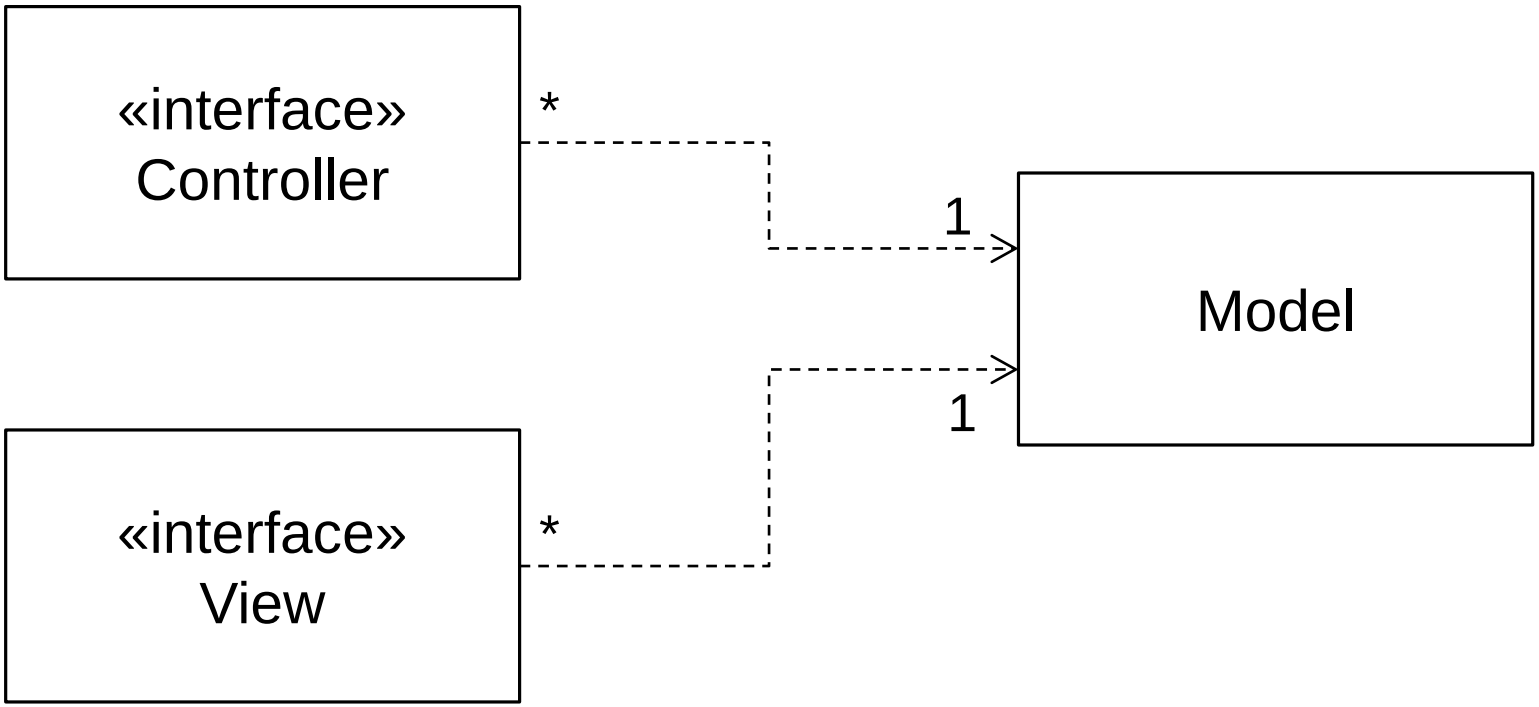
Monroe, R.T., Kompanek, A., Melton, R., Garlan, D., "Architectural styles, design patterns, and objects," *IEEE Software*, **14**, 1, 43–52, Jan/Feb 1997, doi: 10.1109/52.566427

Mary Shaw. 1996. "Some patterns for software architectures". In *Pattern languages of program design 2*. Addison-Wesley, 255–269. citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.710

Model-View-Controller (MVC) Pattern

Name	MVC (Model-View-Controller)
Problem	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.
Solution	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.
Pro	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.
Con	Can involve additional code and code complexity when the data model and interactions are simple.

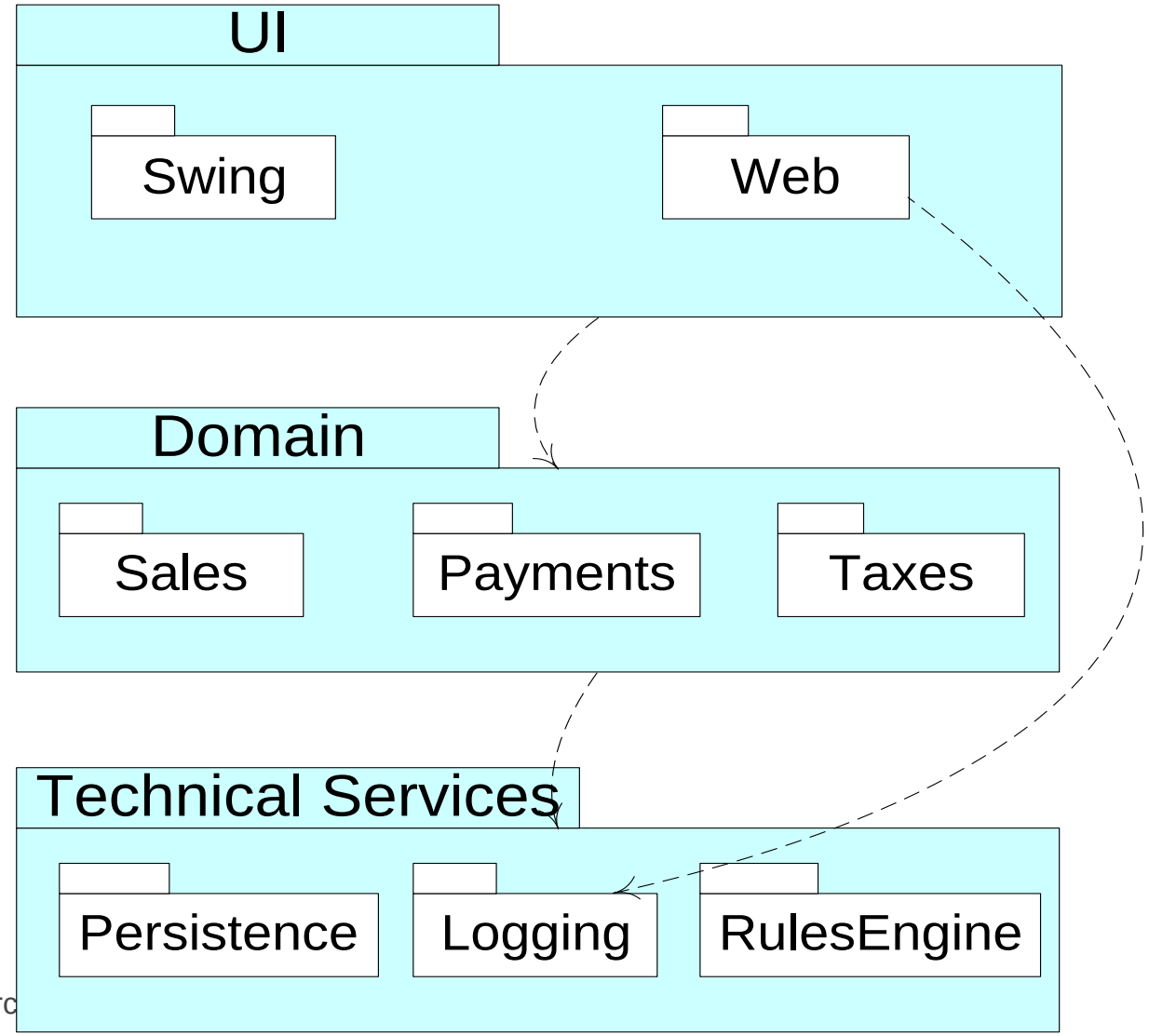
MVC Essential Dependencies Diagram



The Layered Architecture Pattern

Name	Layered architecture
Problem	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.
Solution	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.
Pro	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.
Con	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.

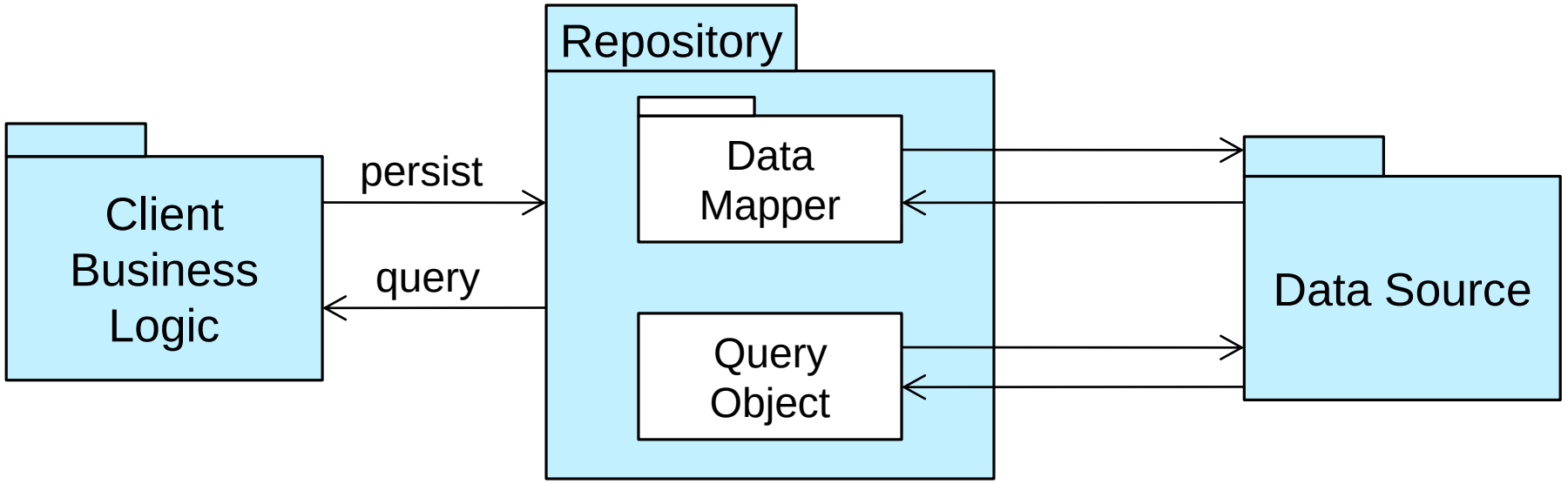
Layered Architecture Diagram



The Repository Pattern

Name	Repository
Problem	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.
Solution	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.
Pro	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.
Con	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.

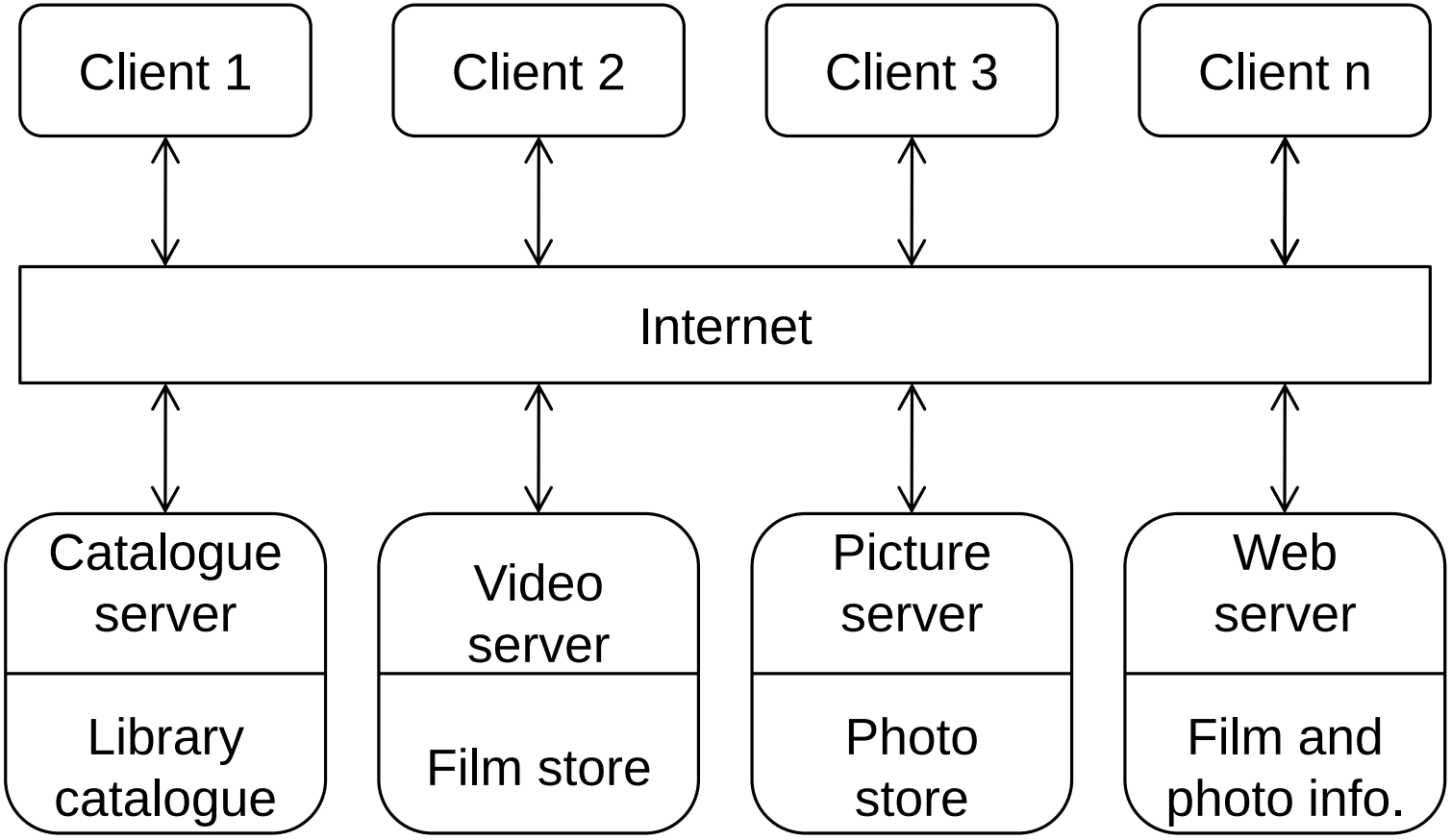
Repository Diagram



The Client–Server Pattern

Name	Client-server
Problem	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.
Solution	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.
Pro	The principal advantage of this model is that servers can be distributed across a network and servers can added or upgraded with minimal disruption. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all.
Con	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.

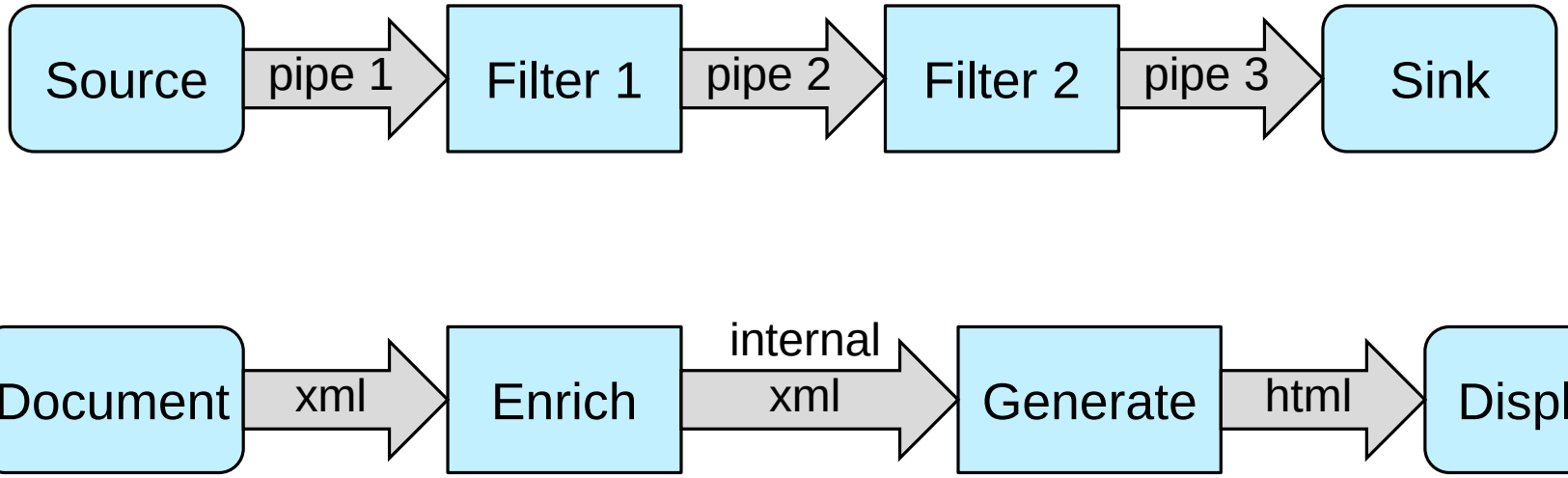
Client–Server Diagram for a Film Library



The Pipe and Filter Pattern

Name	Pipe and filter
Problem	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Solution	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.
Pro	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.
Con	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.

Pipes and Filters to Break Processing into Steps



Key Aim of these Patterns for Design

- Cohesion
  - degree to which communication takes place within the module
- Coupling
  - degree to which communication takes place between modules
- Minimize coupling while maximizing cohesion