Architectural Design

Chapter 6

Software Design

• Process of converting the requirements into the design of the system.
• Definition of how the software is to be structured or organized.

• For large systems, this is divided into two parts:
  - Architectural design defines main components of the system and how they interact.
  - Detailed design components are decomposed and described at a much finer level of detail.

Architectural Design

• Architectural Design:
  - the design process for identifying:
    - the subsystems making up a system and
    - the relationships between the subsystems

• what's a subsystem?
  - A subsystem is a sub-part of the system that provides (and/or consumes) services to other subsystems

• Architectural Design = subsystem decomposition
  - break the system into subparts with the goal of simplifying the overall system.

Services and subsystem interfaces

• A service is a set of related operations that share a common purpose.

• Subsystem interface: the set of one subsystems' operations that are available to other subsystems.
  - specification includes the signature of each operation:
    - name of the operation, types of each parameter.

• Why specify interfaces to components?
  - so components may be developed in parallel

• Helps promote separation/independence.
  - can make changes behind the interface without affecting components using that interface.
**Architectural Models**

- Simple box and line diagrams
- Each box is a component of the system (a subsystem)
- Boxes within boxes are subcomponents of a subsystem
- Arrows indicate data and/or messages are passed between components

**Example: Architecture of a packing robot control system**

The robot uses the vision system to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. It packs the object, and places it on another conveyor.

**Architectural Patterns**

- An architectural pattern is an abstract description of system organization that has been successful in previous projects (in various contexts)
- Patterns are a means of representing, sharing and reusing knowledge.
  - Architectural designers can browse pattern descriptions to identify potential candidates
- Each pattern description should indicate in which contexts it is and is not useful.
- Goals: simplicity, separation/independence

**Model-View-Controller (MVC) Pattern**

- Commonly used in desktop applications and web applications.
- Used to separate the data (the model) from the way it is presented to the user (the views)
- **Model objects** encapsulate the data.
- **View objects** present data to and receive actions from the user.
- **Controller** manages communication between Model and View (responds to user actions).
Model-View-Controller (MVC) Pattern Diagram

**Controller**
- Maps user actions to model updates
- Sends user events to controller

**View**
- Renders model
- Requests model updates
- Sends user events to controller

**Model**
- Encapsulates application state
- Notifies view of state changes

- State change (set values in model)
- State query (get values from model)
- Change notification

**User events**
- Change notifications

**Model-View-Controller (MVC) Pattern Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>MVC (Model-View-Controller)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Model component</strong> manages the system data and associated operations on that data.</td>
</tr>
<tr>
<td></td>
<td>• <strong>View component</strong> defines and manages how the data is presented to the user.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Controller component</strong> manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.</td>
</tr>
<tr>
<td>Example</td>
<td>Most web-based application systems, most desktop apps.</td>
</tr>
<tr>
<td>When used</td>
<td>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.</td>
</tr>
<tr>
<td>Advantages</td>
<td>Allows the data (model) to change independently of its representation (view) and vice versa. Supports presentation of the same data in different ways with user changes made in one representation shown in all of them.</td>
</tr>
<tr>
<td>Disadvantages</td>
<td>Can involve additional code and code complexity when the data model and interactions are simple.</td>
</tr>
</tbody>
</table>

Layered Architecture Pattern

- System functionality is organized into separate layers.
- Each layer relies only on facilities and services of layer immediately beneath it.
- Advantages:
  - Changes to one layer do not affect layers above
  - If interface changes, affects only layer above.
  - Easily replace one layer by another equivalent one (with same interface).
  - Portability: need to change only bottom layer to port to different machine(s).

**Layered Architecture Pattern Diagram**

- User interface
- User interface management
- Authentication and authorization
- Core business logic/application functionality
- System utilities
- System support (OS, database etc.)
Layered Architecture Pattern

Example: LIBSYS

- Allows controlled electronic access to copyrighted material from a group of university libraries.
- Example: A layered model of a system for sharing copyright documents held in different libraries: LIBSYS.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>Forms and query manager</td>
</tr>
<tr>
<td>Search</td>
<td>Rights manager</td>
</tr>
<tr>
<td>Retrieval</td>
<td>Accounting</td>
</tr>
<tr>
<td>Index</td>
<td>Print manager</td>
</tr>
</tbody>
</table>

Databases from different libraries:

DB1, DB2, DB3, DB4, DBn

Pipe and Filter Architecture

- A series of transformations on data.
- Composed of:
  - A set of “filters”, each one transforming some input stream into an output stream.
  - Pipes connecting the filters.
- Data is transformed as it moves through the system.
- Transformations can be run concurrently.
- Commonly used in batch processing systems and embedded control systems.
- Difficult to use for interactive systems.

Pipe and Filter Architecture

Example: Processing invoices

- Once a week, payments are reconciled against invoices (issued at the beginning of the month). For paid invoices, it issues a receipt. For unpaid, it issues a reminder.
**Pipe and Filter Example: Compiler (g++)**

Pipe and filter example diagram showing the processing stages:
- **Lexical analysis** produces a stream of tokens.
- Syntax is output of Syntactic analysis, which is transformed by Semantic analysis.

**Syntax tree** is the output of Syntactic analysis, which is transformed by Semantic analysis.

**Repository Architecture**

- Data is stored in a central shared repository.
- Components interact through the repository only.
- Suited to applications whose data is generated by one component and used by another.
- **Advantages:**
  - Components are independent/separate.
  - Changes to data are automatically available to other components.
- Communication between components may be inefficient.

**Pipe and Filter Architecture Description**

<table>
<thead>
<tr>
<th>Name</th>
<th>Pipe and filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Example</td>
<td>The pipe and filter system used for processing invoices.</td>
</tr>
<tr>
<td>When used</td>
<td>Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.</td>
</tr>
<tr>
<td>Advantages</td>
<td>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.</td>
</tr>
<tr>
<td>Disadvantages</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Repository Architecture Example: Compiler/IDE (eclipse)**

Diagram showing components and their interactions:
- **Lexical analyzer**
- **Syntax analyzer**
- **Semantic analyzer**
- **Editor**
- **Pretty-printer**
- **Repository**
- **Grammar definition**
- **Abstract syntax tree**
- **Symbol table**
- **Output definition**
- **Optimizer**
- **Code generator**

**Grammar definition:** syntax rules for input language.

**Output definition:** syntax rules for output language.
Repository Architecture

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td>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.</td>
</tr>
</tbody>
</table>

Example: An IDE where the components use a repository of system design information. Each software component generates information which is then available for use by other tools.

When used: *When large volumes of information are generated that has to be stored for a long time.*

*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.

---

Client-Server Architecture

- Commonly used organization for distributed systems.
- Composed of:
  - A set of servers that offer specific (unique) services to other components.
  - A set of clients that call on services offered by the servers
  - A network that allows the clients to access the services.
- Could run on a single computer: separation/independence.
- Clients make remote procedure calls to servers using a protocol like http, waits for reply.
- Several instances of client on different machines.

### Example: Film Library

Serves information, videos, still photos. Catalog server handles searching. Clients are multiple instances of a user interface (in a web browser).

---

Client-Server Architecture

<table>
<thead>
<tr>
<th>Name</th>
<th>Client-server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>

Example: The 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 servers.

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.