Chapter 6

Architectural Design
Topics covered

- The WHAT and WHY of architectural design
- Architectural design decisions
- Architectural views/perspectives
- Architectural styles/patterns
- (Generic) Application architectures
What is architectural design?

- An early stage of the system design process.
- The process of identifying the sub-systems making up a system and a framework for sub-system communication and control.
- A boot-strapping process undertaken in parallel with some specification activities.
- The output of this process is the software architecture.
Boot-strapping
Architectural abstraction

- **Architecture in the small**: concerned with the architecture of *individual programs*. (The way that an individual program is decomposed into components.)
  - Principal focus of this chapter...

- **Architecture in the large**: concerned with the architecture of complex, enterprise systems that include other (sub-)systems, programs, and program components. (These may be distributed over different computers which could be owned and managed by different companies.)
  - Chapters 18 and 19...
Advantages of explicit architecture design and documentation (Bass)

- **Stakeholder communication** — the architecture may be used as a focus of discussion by system stakeholders. (Requirements can be organized by sub-system.)

- **System analysis** — the feasibility of meeting critical non-functional requirements (e.g., performance, reliability, maintainability constraints) can be studied early-on.

- **Large-scale reuse** — the architecture may be reusable across a range of systems with similar requirements.
Architectural representations

- Simple, informal block diagrams showing components and relationships are the most frequently used...
- But these have been criticized because they do not show the types of relationships between components, nor the component’s externally visible properties.
- In general, model semantics depends on how the models are to be used...
Example of a simple block diagram: Packing robot control system

- Vision system
- Object identification system
- Arm controller
- Gripper controller
- Packaging selection system
- Packing system
- Conveyor controller
Architectural representations

- Simple, informal block diagrams showing components and relationships are the most frequently used...
- But these have been criticized because they do not show the types of relationships between components, nor the components’ externally visible properties.
- In general, model semantics depends on how the models are to be used...
Architectural representations

- Simple, informal block diagrams showing components and relationships are the most frequently used…
- But these have been criticized because they do not show the types of relationships between components, nor the components’ externally visible properties.
- In general, model semantics depends on how the models are to be used…
Two examples of how an architectural model might be used:

- **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 planners because it is not cluttered with 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.
Architectural design decisions

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

(cont’d)
Architectural design decisions (cont’d)

- Is there a *generic application architecture* that can be used?
- How will the system be *distributed*?
- What architectural *styles* are appropriate?
- What approach will be used to *structure* the system?
- How will the system be *decomposed into modules*?
- What *control strategy* should be used?
- How will the architectural design be *evaluated*?
- How should the architecture be *documented*?
Architecture reuse

- Systems in the same domain often have similar architectures reflecting domain characteristics or concepts.
- E.g., *application product lines* are often built around a core architecture reflecting a given domain with variants that satisfy particular end-user or customer requirements.
- Consider, for example the MS Office product line.
Architectural styles / patterns*

- The architecture of systems may be based on one or more architectural styles or patterns.
- Architectural styles / patterns are based on structural system organization models chosen in accordance with the attributes associated with particular non-functional system requirements…
- They represent “good design practices” for a given attribute that has been tried and tested in different environments.

* The terms style and pattern are used interchangeably in this context.
System attributes and (associated) architectural styles and structures

- **Performance** – localize operations by using fewer, large-grain components deployed on the same computer to minimize sub-system communication. (reflected in repository architecture model)

- **Security** – use a layered architecture with critical assets protected in inner layers. (reflected in the layered / abstract machine architecture model)

- **Safety** – isolate safety-critical components in one or just a few sub-systems.

(cont’d)
System attributes and (associated) architectural styles and structures (cont’d)

- **Availability** – include redundant components in the architecture.

- **Maintainability** – use (more) fine-grain, self-contained components; avoid shared data structures. *(reflected in the object-oriented program decomposition model)*
Architectural views/perspectives

- Different views or perspectives are useful when designing and documenting a system’s architecture.
- There are different opinions as to what views are required, however.
- Krutchen (‘95) suggests there should be 4 fundamental architectural views, which are related to one another using use cases or scenarios...
“4+1 view model” of software architecture (Kruchten ‘95)

1. **logical view**: shows the key abstractions in the system as objects or object classes.

2. **process view**: shows how, at run-time, the system is composed of interacting processes.

3. **development view**: shows how the software is decomposed for development.

4. **physical view**: shows the system hardware and how software components are distributed across the processors in the system.

+1. **Relate the above** to one another using use-cases or scenarios.
More on architectural patterns

- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments. (cf design patterns)
- They provide a means of representing, sharing and reusing knowledge.
- Patterns should include information about when they are and when they are not useful.
- Patterns may be represented using tabular and/or graphical descriptions.
# The Model-View-Controller (MVC) pattern

<table>
<thead>
<tr>
<th>Name</th>
<th>MVC (Model-View-Controller)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The <strong>Model component</strong> manages the system data and associated operations on that data. The <strong>View component</strong> defines and manages how the data is presented to the user. The <strong>Controller component</strong> manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.</td>
</tr>
<tr>
<td><strong>When used</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>Can involve additional code and code complexity when the data model and interactions are simple.</td>
</tr>
</tbody>
</table>

*(Generalization of the “observer” design pattern.)*
Organization of the Model-View-Controller (illustrated with UML packages)
Example of a Web application architecture using the MVC pattern
The Layered architecture pattern

- Also known as the *abstract machine* model.
- Used to model the interfacing of sub-systems.
- Organizes a system into a set of layers.
- Each layer provides a set of services used to implement the next layer.
- When a layer interface changes, only the adjacent layer is affected.
- However, it is often difficult / artificial to structure systems in this way.
### The Layered architecture pattern (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Layered architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Organizes the system into layers with related functionality associated with each layer. <em>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.</em> See Figure 6.6.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.</td>
</tr>
<tr>
<td><strong>When used</strong></td>
<td>Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsible for a layer of functionality; when there is a requirement for multi-level security.</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
A generic layered architecture

- User interface
- User interface management
  Authentication and authorization
- Core business logic/application functionality
  System utilities
- System support (OS, database etc.)
The Repository pattern

- Sub-systems must exchange info. This may be done \textit{(at the extremes)} in two ways:
  - Shared data is held in a central database or \textit{repository} and may be accessed by all sub-systems. \textit{(data is “global”)}
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are used, the repository model of sharing is commonly used.
## The Repository pattern (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</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>
<tr>
<td>Example</td>
<td>Figure 6.9 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.</td>
</tr>
<tr>
<td>When used</td>
<td>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.</td>
</tr>
<tr>
<td>Advantages</td>
<td>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.</td>
</tr>
<tr>
<td>Disadvantages</td>
<td>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.</td>
</tr>
</tbody>
</table>
A repository architecture for an Integrated Development Environment (IDE)
The Client-server pattern

- **Distributed system model** which **shows how data and processing are distributed across a range of processors.** (machines)

- **Major components:**
  - A set of **stand-alone servers** which provide specific services such as printing, file management, etc.
  - A set of **clients** which call on these services
  - A **network** which allows clients to access these services
## The Client-server pattern (cont’d)

<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>
<tr>
<td>Example</td>
<td>Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.</td>
</tr>
<tr>
<td>When used</td>
<td>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.</td>
</tr>
<tr>
<td>Advantages</td>
<td>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.</td>
</tr>
<tr>
<td>Disadvantages</td>
<td>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.</td>
</tr>
</tbody>
</table>
A client-server architecture for a film and picture library

![Diagram of client-server architecture]

- Client 1
- Client 2
- Client 3
- Client 4

Wide-bandwidth network

- Catalogue server
- Film clip files
- Digitized photographs
- Hypertext server

Catalogue

Video server

Picture server

Hypertext web

Wide-bandwidth network
The pipe and filter pattern

- Also known as the *data-flow* architecture.
- Functional transformations process inputs to produce outputs.
- Variants of this approach have a long history in software design. *(e.g., SA/SD, SADT, etc.)*
- When transformations are sequential, this is a *batch sequential model* which is extensively used in data processing systems.
- Not really suitable for interactive systems *(focus on input data streams vs. events)*
# The pipe and filter pattern (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Pipe and filter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The processing of the data in a system is organized so that each processing component (filter) is distinct 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><strong>Example</strong></td>
<td>Figure 6.13 is an example of a pipe and filter system used for processing invoices.</td>
</tr>
<tr>
<td><strong>When used</strong></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><strong>Advantages</strong></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><strong>Disadvantages</strong></td>
<td>The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unpars 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>
An example of the pipe and filter architecture (invoice processing system)
(Generic) application architectures

- Application systems are designed to meet organizational needs.
- 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 of a given type 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 organizing the work of a development team.
- As a means of assessing components for reuse.
- As a vocabulary for talking about application types.
Four examples of application architecture types

- **Data processing applications**: data driven applications that process data in batches without explicit user intervention during the processing.

- **Transaction processing applications**: data-centered applications that process user requests and update information in a system database.

(cont’d)
Four examples of application architecture types (cont’d)

- **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.
Four examples of application architecture types (cont’d)

- **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.

  (So what is the difference between a “Generic Application Architecture” and an “Architectural Pattern”?)
Application type examples

- Focus here is on transaction processing and language processing systems.
- 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 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 which are then processed by a transaction manager.
The structure of transaction processing applications
Example: dataflow architecture of an ATM system
Language processing systems

- Translate a natural or artificial language into another representation of that language, e.g.:
  - Programming language source code into machine code
  - XML data descriptions into database query commands
  - French into Norwegian

- May include an interpreter to execute instructions in the language being processed.
The architecture of a language processing system
Example: repository architecture for a language processing system

Repository-based model
A dataflow compiler architecture

Sequential function model
(batch processing oriented)
Compiler components

- A **lexical analyzer**, 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.

(cont’d)
Compiler components (cont’d)

- A **syntax tree**, which is an internal structure representing the program being compiled.
- 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.
So what’s the difference?

- **Architectural Patterns** are structural system organization models that are usually associated with competing emergent, non-functional system attributes (e.g., performance, maintainability, availability).

- **Generic Application Architectures** are logical system models specific to some common application type (e.g., data processing, transaction processing, event processing).

- A **Generic Application Architecture** may be successfully implemented for a *specific* application using various Architectural Patterns.
Key points

- At the highest level, a software architecture is just a description of how a software system is organized.
- Architectural design decisions include decisions on the type of application, the distribution of sub-systems, and the architectural styles to be used.
- Architectures may be documented from several different perspectives or views: e.g., a logical view, a process view, a development view, and a physical view.

(cont’d)
Key points (cont’d)

- *Architectural patterns* are a means of reusing knowledge about generic system architectures. They describe the architecture, when it may be used, and its advantages and disadvantages.

- Generic application architectures help us understand and compare applications, validate application system designs, and assess large-scale components for reuse.

(cont’d)
Key points (cont’d)

- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
- Language processing systems may be used to translate texts from one language into another and may include a translator and an abstract machine that executes the generated language.
Chapter 6

Architectural Design