**Chapter 5 – System Modeling**

**Topics covered**

* Context models
* Interaction models
* Structural models
* Behavioral models
* Model-driven engineering

**System modeling (mô hình hóa hệ thống)**

* System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective (khía cạnh) of that system.
* **System modeling has now come to mean representing a system using some kind of graphical notation,** which is now almost always based on notations in the Unified Modeling Language (UML) *(~Ngôn ngữ mô hình duy nhất).* 
  + - *UML: dùng ký hiệu để vẽ mô tả, để ai đọc vào đều hiểu duy nhất một ý tưởng.*
* System modelling helps the analyst to *understand the functionality* of the system and models are used *to communicate with customers.*

**Existing and planned system models**

* Models of the existing system are used during requirements engineering. They help clarify (sáng tỏ) what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
* Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
* In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

**System perspectives**

* An external perspective, where you model the context or environment of the system.
* An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
* A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
* A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

**UML diagram types**

* Activity diagrams, which show the activities involved in a process or in data processing .
* Use case diagrams, which show the interactions between a system and its environment.
* Sequence diagrams, which show interactions between actors and the system and between system components.
* Class diagrams, which show the object classes in the system and the associations between these classes.
* State diagrams, which show how the system reacts to internal and external events.

**Use of graphical models**

* As a means of facilitating discussion about an existing or proposed system
  + Incomplete and incorrect models are OK as their role is to support discussion.
* As a way of documenting an existing system
  + Models should be an accurate representation of the system but need not be complete.
* As a detailed system description that can be used to generate a system implementation
  + Models have to be both correct and complete.

**Context models**

**Context models**

* Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
* Social and organisational concerns may affect the decision on where to position system boundaries.
* Architectural models show the system and its relationship with other systems.

**System boundaries**

* System boundaries are established to define what is inside and what is outside the system.
  + They show other systems that are used or depend on the system being developed.
* The position of the system boundary has a profound effect on the system requirements.
* Defining a system boundary is a political judgment
  + There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.

**The context of the Mentcare system**

5.1 Mentcare context.eps

**Process perspective**

* Context models simply show the other systems in the environment, **not how the system being developed is used in that environment.**
* Process models reveal how the system being developed is used in broader business processes.
* UML activity diagrams may be used to define business process models.

**Process model of involuntary detention**

5.2 Detention Process.eps

*Chấm tròn: Điểm đầu*

*Chấm tròn viền: Điểm cuối*

*Đường ngang / dọc đậm: Các công việc* ***đồng thời*** *thực hiện (E.g. có quyết định –> có 02 việc cùng xảy ra: thông báo cho bệnh nhân & lưu lại bản ghi); có 2 dạng: vào 1 ra nhiều, vào nhiều ra 1.*

*Hình thoi: Rẽ nhánh (~điều kiện)*

**Interaction models**

**Interaction models**

* Modeling user interaction is important as it helps to identify user requirements.
* Modeling system-to-system interaction highlights the communication problems that may arise.
* Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
* Use case diagrams and sequence diagrams may be used for interaction modeling.

**Use case modeling**

* Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
* Each use case represents a discrete task that involves external interaction with a system.
* Actors in a use case may be people or other systems.
* Represented diagramatically to provide an overview of the use case and in a more detailed textual form.

**Transfer-data use case**

* A use case in the Mentcare system

5.3 UseCase.eps

*Actor: Có thể là con người/* ***có thể là hệ thống*** *(đi tương tác với các hệ thống khác)*

*Use case mô tả giữa 2 actors với nhau (cụ thể là con người và hệ thống)*

*Con người:* ***cần 1 giao diện*** *để thao tác*

**Tabular description of the ‘Transfer data’ use-case**

|  |  |
| --- | --- |
| **MHC-PMS: Transfer data** | |
| Actors | Medical receptionist, patient records system (PRS) |
| Description | A receptionist may transfer data from the Mentcase system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient’s diagnosis (chẩn đoán) and treatment. |
| Data | Patient’s personal information, treatment summary |
| **Stimulus (lệnh tương tác đến hệ thống/ Kích hoạt)** | User command issued by medical receptionist |
| Response (Đảm bảo thành công) | Confirmation that PRS has been updated |
| Comments | The receptionist must have appropriate security permissions to access the patient information and the PRS. |

**Use cases in the Mentcare system involving the role ‘Medical Receptionist’**

5.5 RecepUseCases.eps

**Sequence diagrams**

* Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.
* A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.
* The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
* Interactions between objects are indicated by annotated arrows.

**Sequence diagram for View patient information**

5.6 ViewInfo Seq Diag.eps

Đường đi (truyền dữ liệu) là đường nét liền, đường trả về là đường nét đứt

P là thể hiện của lớp PatientInfo

**Sequence diagram for Transfer Data**

A diagram of a medical receptionist

Description automatically generated

**Structural models**

**Structural models**

* Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
* Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
* You create structural models of a system when you are discussing and designing the system architecture.

**Class diagrams**

* Class diagrams are used **when developing an object-oriented system model** to show the classes in a system and the associations between these classes.
* An object class can be thought of as a general definition of one kind of system object.
* An association is a link between classes that indicates that there is some relationship between these classes.
* When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

**UML classes and association**

5.8 ClassAssoc.eps

*1 bệnh nhân chỉ có duy nhất 1 hồ sơ bệnh án*

**Classes and associations in the MHC-PMS**

5.9 MHCPMS-classes.eps

**The Consultation class**

5.10 Consultation Class.eps

***Vẽ lớp chi tiết:*** *Chia thành 3 phần: phần đầu là tên lớp, phần giữa là thuộc tính của lớp (+: public, - private, #: protected), phần cuối là phương thức của lớp*

**Generalization (Tổng quát hóa)**

* Generalization is an everyday technique that we use to manage complexity.
* Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.
* This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.
* In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.
* In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.
* In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.
* The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

**A generalization hierarchy**

5.11 GeneralizationHierarchy.eps

**A generalization hierarchy with added detail**

5.12 GeneralisationDetail.eps

**Object class aggregation models**

* An aggregation model shows how classes that are collections are composed of other classes (1 lớp được cấu thành từ các lớp khác) (E.g. PC aggregation các thành phần của máy tính <CPU, …>).
* Aggregation models are similar to the part-of relationship in semantic data models.

**The aggregation association**

5.13 Aggregation.eps

*Một class này có thể là aggregation của nhiều lớp khác*

**Behavioral models**

**Behavioral models**

* Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
* You can think of these stimuli as being of two types:
  + Data Some data arrives that has to be processed by the system.
  + Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

**Data-driven modeling**

* Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
* Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
* They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

**An activity model of the insulin pump’s operation**

5.14 PumpDFD.eps

**Order processing**

5.15 OrderSeq.eps

**Event-driven modeling**

* Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.
* Event-driven modeling shows how a system responds to external and internal events.
* It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

**State machine models**

* These model the behaviour of the system in response to external and internal events.
* They show the system’s responses to stimuli so are often used for modelling real-time systems.
* State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
* Statecharts are an integral part of the UML and are used to represent state machine models.

**State diagram of a microwave oven**

5.16 MWOvenStateDiag.eps

**Microwave oven operation**

5.18 Operate-state-mc.eps

**States and stimuli for the microwave oven (a)**

|  |  |
| --- | --- |
| **State** | **Description** |
| Waiting | The oven is waiting for input. The display shows the current time. |
| Half power | The oven power is set to 300 watts. The display shows ‘Half power’. |
| Full power | The oven power is set to 600 watts. The display shows ‘Full power’. |
| Set time | The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set. |
| Disabled | Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’. |
| Enabled | Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’. |
| Operation | Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding. |

|  |  |
| --- | --- |
| **Stimulus** | **Description** |
| Half power | The user has pressed the half-power button. |
| Full power | The user has pressed the full-power button. |
| Timer | The user has pressed one of the timer buttons. |
| Number | The user has pressed a numeric key. |
| Door open | The oven door switch is not closed. |
| Door closed | The oven door switch is closed. |
| Start | The user has pressed the Start button. |
| Cancel | The user has pressed the Cancel button. |

**Model-driven engineering**

**Model-driven engineering**

* Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
* The programs that execute on a hardware/software platform are then generated automatically from the models.
* Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

**Usage of model-driven engineering**

* Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.
* Pros
  + Allows systems to be considered at higher levels of abstraction
  + Generating code automatically means that it is cheaper to adapt systems to new platforms.
* Cons
  + Models for abstraction and not necessarily right for implementation.
  + Savings from generating code may be outweighed by the costs of developing translators for new platforms.

**Model driven architecture**

* Model-driven architecture (MDA) was the precursor of more general model-driven engineering
* MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
* Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

**Types of model**

* **A computation independent model (CIM)** 
  + These model the important domain abstractions used in a system. CIMs are sometimes called domain models.
* **A platform independent model (PIM)** 
  + These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
* **Platform specific models (PSM)** 
  + These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

**MDA transformations**

5.19 MDA-Transformations.eps

**Multiple platform-specific models**

5.20 Multiple PSMs.eps

**Agile methods and MDA**

* The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.
* The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering.
* If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

**Adoption of MDA**

* A range of factors has limited the adoption of MDE/MDA
* Specialized tool support is required to convert models from one level to another
* There is limited tool availability and organizations may require tool adaptation and customisation to their environment
* For the long-lifetime systems developed using MDA, companies are reluctant to develop their own tools or rely on small companies that may go out of business
* Models are a good way of facilitating discussions about a software design. Howeverthe abstractions that are useful for discussions may not be the right abstractions for implementation.
* For most complex systems, implementation is not the major problem – requirements engineering, security and dependability, integration with legacy systems and testing are all more significant.
* The arguments for platform-independence are only valid for large, long-lifetime systems. For software products and information systems, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
* The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

**Key points**

* A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system’s context, interactions, structure and behavior.
* Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
* Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
* Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
* Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.
* Activity diagrams may be used to model the processing of data, where each activity represents one process step.
* State diagrams are used to model a system’s behavior in response to internal or external events.
* Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

**Chapter 6 – Architectural Design**

**Topics covered**

* Architectural design decisions
* Architectural views
* Architectural patterns
* Application architectures

**Architectural design**

* Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
* Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
* The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

**Agility and architecture**

* It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
* Refactoring the system architecture is usually expensive because it affects so many components in the system

**The architecture of a packing robot control system**

6

**Architectural abstraction**

* Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
* Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

**Advantages of explicit architecture**

* Stakeholder communication
  + Architecture may be used as a focus of discussion by system stakeholders.
  + Stakeholder: những cá nhân/ tổ chức có thể ảnh hưởng (tốt/ xấu) đến dự án/ chương trình mà ta đang phát triển.
* System analysis
  + Means that analysis of whether the system can meet its non-functional requirements is possible.
* Large-scale reuse
  + The architecture may be reusable across a range of systems
  + Product-line architectures may be developed.

*UI/UX 🡪 là yêu cầu*

*UI = User Interface: button có hình gì, màu gì… - Giao diện trông ntn*

*UX = User Experience: để button tại chỗ đó có bất tiện/ hữu ích với người dùng không (điện thoại rung) – Giao diện hỗ trợ người dùng như thế nào*

**Architectural representations**

* Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
* But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
* Depends on the use of architectural models.The requirements for model semantics depends on how the models are used.

**Box and line diagrams**

* Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
* However, useful for communication with stakeholders and for project planning.

**Use of architectural models**

* 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 planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by 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 decisions**

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

6.2 Arch design questions.eps

**Architecture reuse**

* Systems in the same domain often have similar architectures that reflect domain concepts.
* Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
* The architecture of a system may be designed around one of more architectural patterns or ‘styles’.
  + These capture the essence of an architecture and can be instantiated in different ways.

**Architecture and system characteristics**

* Performance
  + Localise critical operations and minimise communications. Use large rather than fine-grain components.
* Security
  + Use a layered architecture with critical assets in the inner layers.
* Safety
  + Localise safety-critical features in a small number of sub-systems.
* Availability
  + Include redundant components and mechanisms for fault tolerance.
* Maintainability
  + Use fine-grain, replaceable components.

**Architectural views**

**Architectural views**

* What views or perspectives are useful when designing and documenting a system’s architecture?
* What notations should be used for describing architectural models?
* Each architectural model only shows one view or perspective of the system.
  + It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

6.3 Architectural views.eps

**4 + 1 view model of software architecture**

* A logical view, which shows the key abstractions in the system as objects or object classes.
* A process view, which shows how, at run-time, the system is composed of interacting processes (các thành phần tương tác với nhau như thế nào).
* A development view, which shows how the software is decomposed for development (chương trình được chia thành các thành phần nhỏ như thế nào).
* A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
* Related using use cases or scenarios (+1)

**Representing architectural views**

* Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
* I disagree with this as I do not think that the UML includes abstractions appropriate for high-level system description.
* Architectural description languages (ADLs) have been developed but are not widely used

**Architectural patterns**

**Architectural patterns**

* Patterns are a means of representing, sharing and reusing knowledge.
* An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
* Patterns should include information about when they are and when the are not useful.
* Patterns may be represented using tabular and graphical descriptions.

**The Model-View-Controller (MVC) pattern**

|  |  |
| --- | --- |
| **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 6.3. |
| **Example** | Figure 6.4 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** | Can involve additional code and code complexity when the data model and interactions are simple. |

**The organization of the Model-View-Controller**

6

**Web application architecture using the MVC pattern**

6

**Layered architecture**

* Used to model the interfacing of sub-systems.
* Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
* Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
* However, often artificial to structure systems in this way.

**The Layered architecture pattern**

|  |  |
| --- | --- |
| **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 6.6. |
| **Example** | A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7. |
| **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 multi-level security. |
| **Advantages** | 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. |
| **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. |

**A generic layered architecture**

6.6 LayeredArch.eps

**The architecture of the iLearn system**

6.9 iLearn architecture.eps

**Repository architecture**

* Sub-systems must exchange data. This may be done in two ways:
  + Shared data is held in a central database or repository and may be accessed by all sub-systems;
  + Each sub-system maintains its own database and passes data explicitly to other sub-systems.
* When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

**The Repository pattern**

|  |  |
| --- | --- |
| **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 6.9 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. |
| **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**

6.9 RepositoryIDE.eps

**Client-server architecture**

* Distributed system model which shows how data and processing is distributed across a range of components.
  + Can be implemented on a single computer.
* Set of stand-alone servers which provide specific services such as printing, data management, etc.
* Set of clients which call on these services.
* Network which allows clients to access servers.

**The Client–server pattern**

|  |  |
| --- | --- |
| **Name** | **Client-server** |
| **Description** | 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. |
| **Example** | Figure 6.11 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 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. |

**A client–server architecture for a film library**

6.11 ClientServerFilmPhoto.eps

**Pipe and filter architecture**

* Functional transformations process their inputs to produce outputs.
* May be referred to as a pipe and filter model (as in UNIX shell).
* Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
* Not really suitable for interactive systems.

**The pipe and filter pattern**

|  |  |
| --- | --- |
| **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 6.13 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 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. |

**An example of the pipe and filter architecture used in a payments system**

6.13 InvoiceProc.eps

**Application architectures**

**Application architectures**

* Application systems are designed to meet an organisational need.
* 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 for a type of software system 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 organising the work of the development team.
* As a means of assessing components for reuse.
* As a vocabulary for talking about application types.

**Examples of application types**

* Data processing applications
  + Data driven applications that process data in batches without explicit user intervention during the processing.
* Transaction processing applications
  + Data-centred applications that process user requests and update information in a system database.
* 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.

**Application type examples**

* Two very widely used generic application architectures are transaction processing systems 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 requests 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 for service which are then processed by a transaction manager.

**The structure of transaction processing applications**

6.14 TransactionProcSys.eps

**The software architecture of an ATM system**

6.15 ATMSystemArch.eps

**Information systems architecture**

* Information systems have a generic architecture that can be organised as a layered architecture.
* These are transaction-based systems as interaction with these systems generally involves database transactions.
* Layers include:
  + The user interface
  + User communications
  + Information retrieval
  + System database

**Layered information system architecture**

6.16 InfoSysArch.eps

**The architecture of the Mentcare system**

6.17 MHC-PMSArch.eps

**Web-based information systems**

* Information and resource management systems are now usually web-based systems where the user interfaces are implemented using a web browser.
* For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer*.*
* In an e-commerce system, the application-specific layer includes additional functionality supporting a ‘shopping cart’ in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

**Server implementation**

* These systems are often implemented as multi-tier client server/architectures (discussed in Chapter 17)
  + The web server is responsible for all user communications, with the user interface implemented using a web browser;
  + The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
  + The database server moves information to and from the database and handles transaction management.

**Language processing systems**

* Accept a natural or artificial language as input and generate some other representation of that language.
* May include an interpreter to act on the instructions in the language that is being processed.
* Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
  + Meta-case tools process tool descriptions, method rules, etc and generate tools.

**The architecture of a language processing system**

6.18 LangProcSys.eps

**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.
* 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.

**A repository architecture for a language processing system**

6.20 RepositoryLPS.eps

**A pipe and filter compiler architecture**

6.19 PipeFilterCompModel.eps

**Key points**

* A software architecture is a description of how a software system is organized.
* Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
* Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
* Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.
* Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
* 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 are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

**Chapter 7 – Design and Implementation**

**Topics covered**

* Object-oriented design using the UML
* Design patterns
* Implementation issues
* Open source development

**Design and implementation**

* Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
* Software design and implementation activities are invariably inter-leaved.
  + Software design is a creative activity in which you identify software components and their relationships, based on a customer’s requirements.
  + Implementation is the process of realizing the design as a program.

**Build or buy**

* In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users’ requirements.
  + For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
* When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

**Object-oriented design using the UML**

**An object-oriented design process**

* Structured object-oriented design processes involve developing a number of different system models.
* They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
* However, for large systems developed by different groups design models are an important communication mechanism.

**Process stages**

* There are a variety of different object-oriented design processes that depend on the organization using the process.
* Common activities in these processes include:
  + Define the context and modes of use of the system;
  + Design the system architecture;
  + Identify the principal system objects;
  + Develop design models;
  + Specify object interfaces.
* Process illustrated here using a design for a wilderness weather station.

**System context and interactions**

* Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
* Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

**Context and interaction models**

* A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
* An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

**System context for the weather station**

7.1 WeatherStatContext.eps

**Weather station use cases**

7.2 WS-UseCases.eps

**Use case description—Report weather**

|  |  |
| --- | --- |
| **System** | **Weather station** |
| Use case | Report weather |
| Actors | Weather information system, Weather station |
| Description | The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals. |
| Stimulus | The weather information system establishes a satellite communication link with the weather station and requests transmission of the data. |
| Response | The summarized data is sent to the weather information system. |
| Comments | Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future. |

**Architectural design**

* Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
* You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
* The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

**High-level architecture of the weather station**

7.4 WS-Architecture.eps

**Architecture of data collection system**

7.5 DataCollection.eps

**Object class identification**

* Identifying object classes is often a difficult part of object oriented design.
* There is no 'magic formula' for object identification. It relies on the skill, experience   
  and domain knowledge of system designers.
* Object identification is an iterative process. You are unlikely to get it right first time.

**Approaches to identification**

* Use a grammatical approach based on a natural language description of the system.
* Base the identification on tangible things in the application domain.
* Use a behavioural approach and identify objects based on what participates in what behaviour.
* Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

**Weather station object classes**

* Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  + Ground thermometer, Anemometer, Barometer
    - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
  + Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  + Weather data
    - Encapsulates the summarized data from the instruments.

**Weather station object classes**

7.6 WeatherStatObjs.eps

**Design models**

* Design models show the objects and object classes and relationships between these entities.
* There are two kinds of design model:
  + Structural models describe the static structure of the system in terms of object classes and relationships.
  + Dynamic models describe the dynamic interactions between objects.

**Examples of design models**

* Subsystem models that show logical groupings of objects into coherent subsystems.
* Sequence models that show the sequence of object interactions.
* State machine models that show how individual objects change their state in response to events.
* Other models include use-case models, aggregation models, generalisation models, etc.

**Subsystem models**

* Shows how the design is organised into logically related groups of objects.
* In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

**Sequence models**

* Sequence models show the sequence of object interactions that take place
  + Objects are arranged horizontally across the top;
  + Time is represented vertically so models are read top to bottom;
  + Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  + A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

**Sequence diagram describing data collection**

7.7 WS-SeqDiagram.eps

**State diagrams**

* State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
* State diagrams are useful high-level models of a system or an object’s run-time behavior.
* You don’t usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

**Weather station state diagram**

7.8 WS-StateModel.eps

**Interface specification**

* Object interfaces have to be specified so that the objects and other components can be designed in parallel.
* Designers should avoid designing the interface representation but should hide this in the object itself.
* Objects may have several interfaces which are viewpoints on the methods provided.
* The UML uses class diagrams for interface specification but Java may also be used.

**Weather station interfaces**

7.9 Interfaces.eps

**Design patterns**

**Design patterns**

* A design pattern is a way of reusing abstract knowledge about a problem and its solution.
* A pattern is a description of the problem and the essence of its solution.
* It should be sufficiently abstract to be reused in different settings.
* Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

**Patterns**

* *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*

**Pattern elements**

* Name
  + A meaningful pattern identifier.
* Problem description.
* Solution description.
  + Not a concrete design but a template for a design solution that can be instantiated in different ways.
* Consequences
  + The results and trade-offs of applying the pattern.

**The Observer pattern**

* Name
  + Observer.
* Description
  + Separates the display of object state from the object itself.
* Problem description
  + Used when multiple displays of state are needed.
* Solution description
  + See slide with UML description.
* Consequences
  + Optimisations to enhance display performance are impractical.

**The Observer pattern**

|  |  |
| --- | --- |
| **Pattern name** | **Observer** |
| Description | Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change. |
| Problem description | In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.  This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used. |
| Solution description | This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.  The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated. |
| Consequences | The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to thesubject may cause a set oflinkedupdates to observers to be generated, some of which may not be necessary. |

**Multiple displays using the Observer pattern**

7.11 MultipleDisplays.eps

**A UML model of the Observer pattern**

7.12 ObserverPatternUML.eps

**Design problems**

* To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  + Tell several objects that the state of some other object has changed (Observer pattern).
  + Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  + Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  + Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

**Implementation issues**

**Implementation issues**

* Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  + Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  + Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  + Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

**Reuse**

* From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  + The only significant reuse or software was the reuse of functions and objects in programming language libraries.
* Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
* An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

**Reuse levels**

* The abstraction level
  + At this level, you don’t reuse software directly but use knowledge of successful abstractions in the design of your software.
* The object level
  + At this level, you directly reuse objects from a library rather than writing the code yourself.
* The component level
  + Components are collections of objects and object classes that you reuse in application systems.
* The system level
  + At this level, you reuse entire application systems.

**Software reuse**

7.13 Software reuse.eps

**Reuse costs**

* The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
* Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
* The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
* The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

**Configuration management**

* Configuration management is the name given to the general process of managing a changing software system.
* The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
* See also Chapter 25.

**Configuration management activities**

* Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
* System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
* Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

**Configuration management tool interaction**

7.14 CM_activities.eps

**Host-target development**

* Most software is developed on one computer (the host), but runs on a separate machine (the target).
* More generally, we can talk about a development platform and an execution platform.
  + A platform is more than just hardware.
  + It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
* Development platform usually has different installed software than execution platform; these platforms may have different architectures.

**Host-target development**

7.15 Host-target development.eps

**Development platform tools**

* An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
* A language debugging system.
* Graphical editing tools, such as tools to edit UML models.
* Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
* Project support tools that help you organize the code for different development projects.

**Integrated development environments (IDEs)**

* Software development tools are often grouped to create an integrated development environment (IDE).
* An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
* IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

**Component/system deployment factors**

* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
* High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
* If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

**Open source development**

**Open source development**

* Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
* Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
* Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

**Open source systems**

* The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
* Other important open source products are Java, the Apache web server and the mySQL database management system.

**Open source issues**

* Should the product that is being developed make use of open source components?
* Should an open source approach be used for the software’s development?

**Open source business**

* More and more product companies are using an open source approach to development.
* Their business model is not reliant on selling a software product but on selling support for that product.
* They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

**Open source licensing**

* A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  + Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
  + Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
  + Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

**License models**

* The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
* The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
* The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

**License management**

* Establish a system for maintaining information about open-source components that are downloaded and used.
* Be aware of the different types of licenses and understand how a component is licensed before it is used.
* Be aware of evolution pathways for components.
* Educate people about open source.
* Have auditing systems in place.
* Participate in the open source community.

**Key points**

* Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
* The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
* A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
* Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.
* When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
* Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
* Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
* Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.