UML Class Diagrams Examples

Here we provide some examples of **class diagrams** and **object diagrams**:

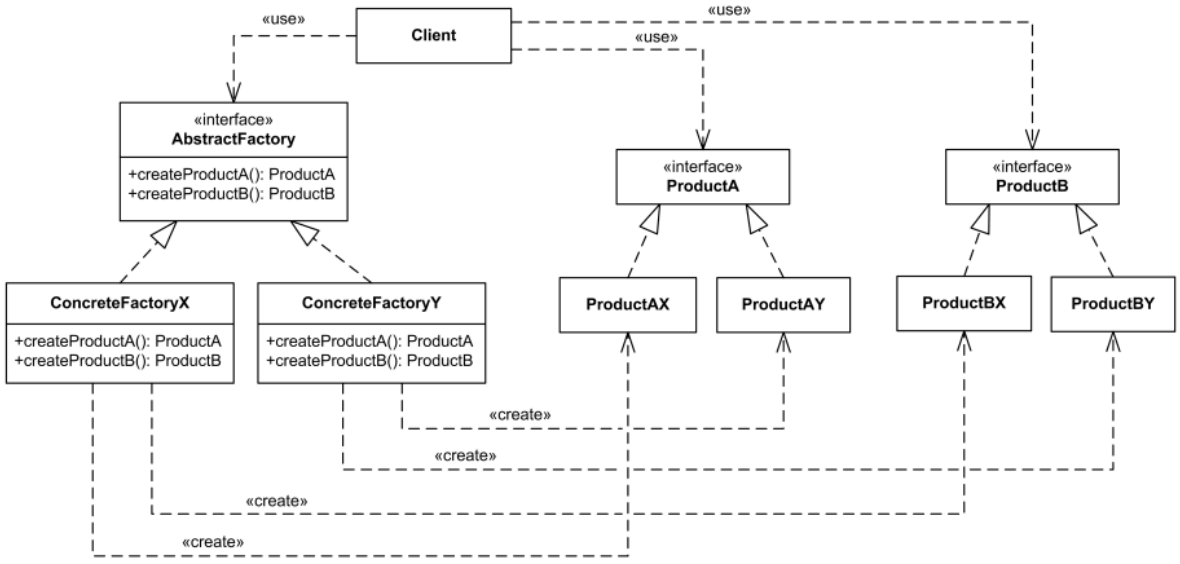
Abstract Factory Design Pattern

***Purpose**: Illustrate Abstract Factory design pattern.*

Abstract Factory is creational software **design pattern**. This pattern provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

Client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the family of objects. The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products.

Use of this pattern makes it possible to interchange families of concrete classes without changing the code that uses them. It separates details of implementation of a set of objects from their usage.



UML class diagram example for the Abstract Factory Design Pattern.

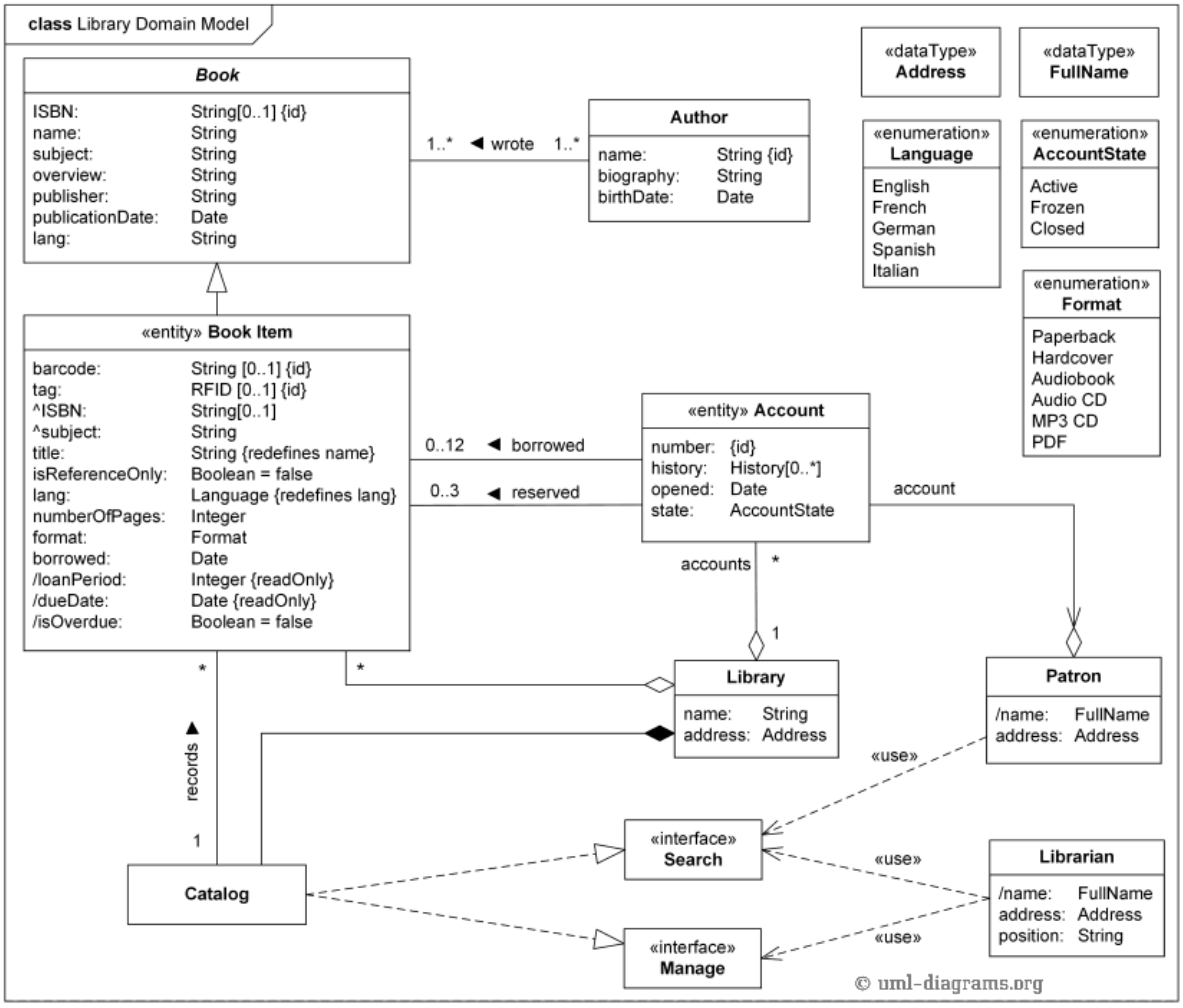
Library domain model

***Purpose**: Describe domain area for an Integrated Library System (ILS), also known as a Library Management System (LMS) - Library, Catalog, Book, Patron, Account.*

Library Domain Model describes main classes and relationships which could be used during analysis phase to better understand domain area for **Integrated Library System** (ILS), also known as a **Library Management System** (LMS).

Each physical library item - book, tape cassette, CD, DVD, etc. could have its own item number. To support it, the items may be **barcoded**. The purpose of barcoding is to provide a unique and scannable identifier that links the barcoded physical item to the electronic record in the **catalog**. Barcode must be physically attached to the item, and barcode number is entered into the corresponding field in the electronic item record.

Barcodes on library items could be replaced by **RFID tags**. The RFID tag can contain item's identifier, title, material type, etc. It is read by an RFID reader, without the need to open a book cover or CD/DVD case to scan it with barcode reader.



UML class diagram example of the Library Domain Model.

Library book attributes *ISBN* and *subject* are **inherited** from *Book* and shown with prepended caret '^' symbol.

The *title* attribute explicitly **redefines** *name*. While type of the attributes is the same, name is different. The *lang* attribute is explicitly redefined with different type. Original type was free text String, while redefined attribute is more specific (e.g. enumerated) *Language* class. We used explicit redefinition in this case because attribute types *String* and *Language* are not related. *Language* is **enumeration** type.

Library has some rules on what could be *borrowed* and what is for *reference only*. Rules are also defined on how many books could be borrowed by *patrons* and how many could be reserved.

Library book attributes *loanPeriod*, *dueDate*, and *isOverdue* are **derived**. Length of time a library book may be borrowed (loan period) depends on library policy and varies based on a kind of book and who is borrowing it. For example, in a university library undergraduates could borrow book for 30 days, graduate students for a quarter, and faculty staff for a year. In a public library normal loan period for a book could be 3 weeks, while it could be lowered to 2 weeks for new books. Book return due date will be calculated based on the borrow date and loan period. If due date is past the current date, *isOverdue* Boolean flag which is false by default will be set to true.

Library *Catalog* provides access for the library patrons and staff to all sources of information about library items, allows to search by a particular author, on a particular topic, or in a particular format, that the library has. It tells the user where materials meeting their specific needs can be found.

Online shopping domain model

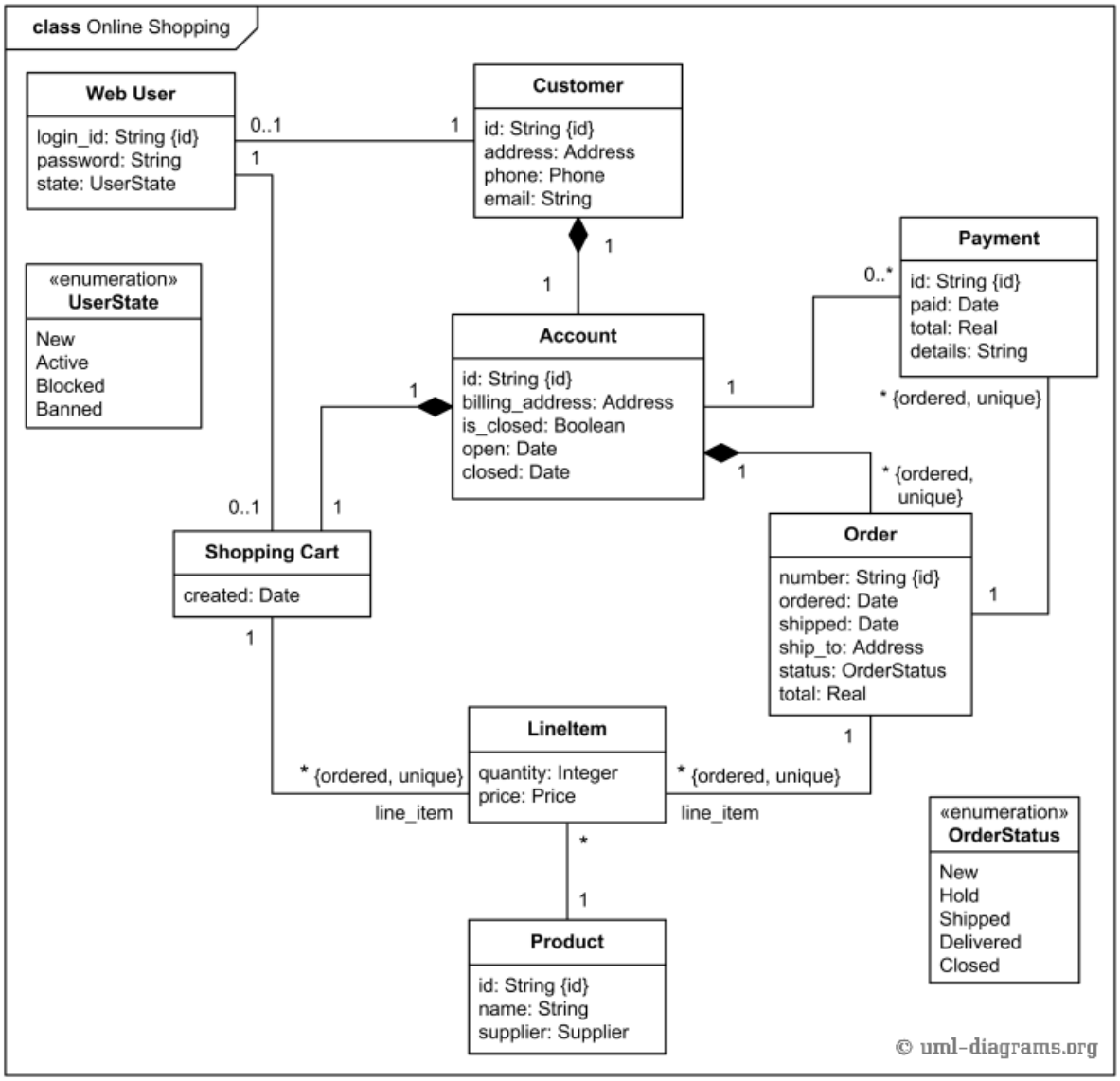
***Purpose**: Show some domain model for online shopping - Customer, Account, Shopping Cart, Product, Order, Payment.*

Here we provide an example of UML **class diagram** which shows a domain model for online shopping. The purpose of the diagram is to introduce some common terms, "dictionary" for online shopping - Customer, Web User, Account, Shopping Cart, Product, Order, Payment, etc. and relationships between. It could be used as a common ground between business analysts and software developers.

Each customer has unique id and is linked to exactly one **account**. Account owns shopping cart and orders. Customer could register as a web user to be able to buy items online. Customer is not required to be a web user because purchases could also be made by phone or by ordering from catalogues. Web user has login name which also serves as unique id. Web user could be in several states - new, active, temporary blocked, or banned, and be linked to a **shopping cart**. Shopping cart belongs to account.

Account owns customer orders. Customer may have no orders. Customer orders are sorted and unique. Each order could refer to several **payments**, possibly none. Every payment has unique id and is related to exactly one account.

Each order has current order status. Both order and shopping cart have **line items** linked to a specific product. Each line item is related to exactly one product. A product could be associated to many line items or no item at all.



Online shopping domain UML class diagram example.

Bank account class diagram example

***Purpose**: Domain model describing common types of bank accounts.*

This is an example describing some types of **Bank Accounts** using UML **generalization sets**.

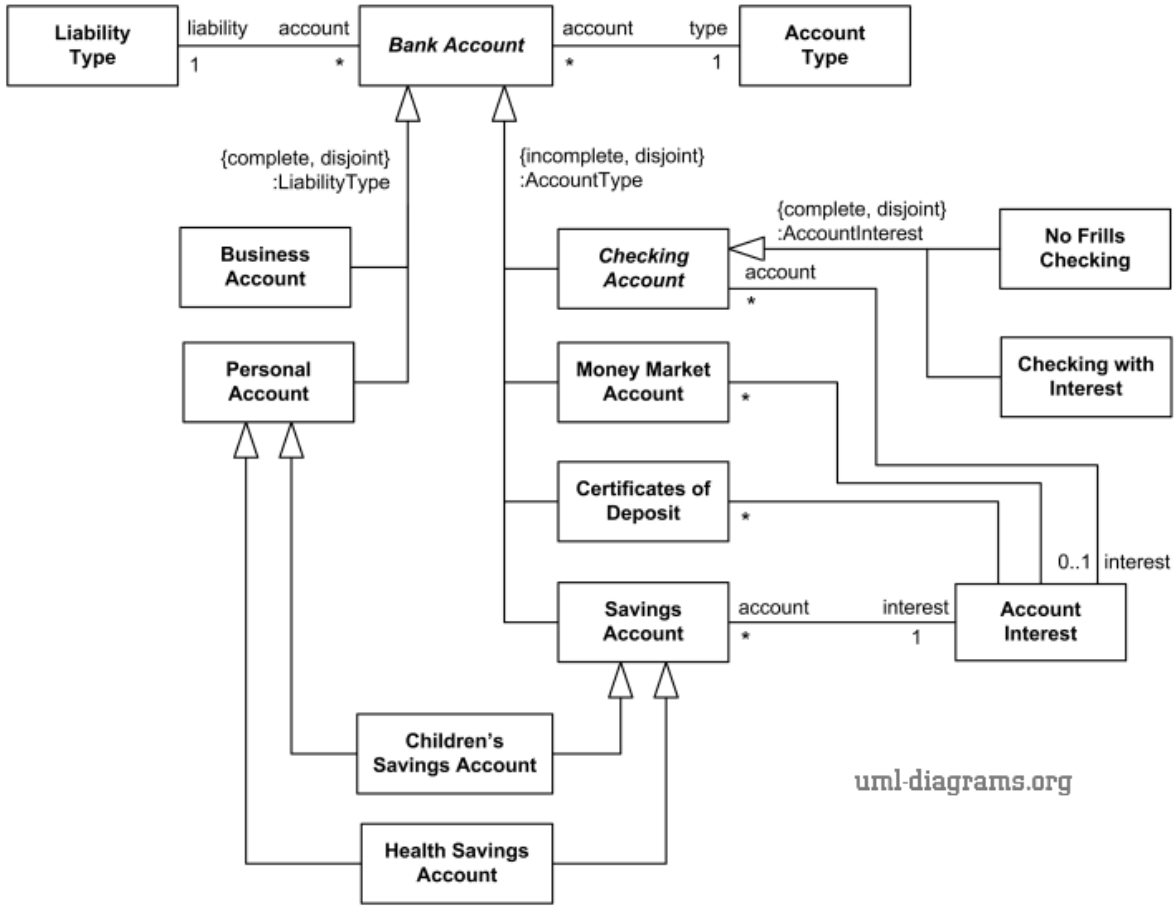
Bank accounts could be grouped into UML generalization sets based on different criteria. Example diagram below shows bank accounts split by liability type and account type. These two orthogonal dimensions also have corresponding **power types** - LiabilityType and AccountType.

Bank account could be used either for personal or for business purposes. To show that it assumes complete coverage and there is no overlapping, we have liability type constraints shown as **{complete, disjoint}**.

Note, that business owners still may use a personal bank account for their business purposes, but it is not recommended primarily because it can affect legal liability of business owner. From the bank's point of view, e.g. when opening an account, these two are two different kinds of accounts.

Another classification of bank accounts is based on related options and features and is shown below as account type generalization set. To show that it this set is incomplete but still there is no overlapping, we have account type constraints shown as **{incomplete, disjoint}**.

Note, that it is possible to have bank accounts with different combinations of account liability and account type, for example, personal savings account or business money market account.



Bank account taxonomy UML class diagram example with generalization sets and power types.

The purpose of **savings account** is to allow us to save money. Account holder can make some limited number of deposits and withdrawals per month, while account provides no checks. Bank usually pays interest rate that is higher than that of a checking account, but lower than a money market account or CDs.

A **checking account** is a bank account that uses checks as a way to withdraw or transfer money from the account - pay bills, buy items, transfer or loan money. Usually banks allow account holders to make withdrawals and deposits through automatic teller machines (ATM). **Basic checking** accounts, sometimes called **No frills** accounts, offer a limited set of services. They usually do not pay interest, have lower required minimum balance, may restrict writing and/or depositing more than a certain number of checks per month. **Checking accounts with interest** have higher required minimum balance but pay interest (based on the average balance maintained), and usually offer a better service, like allowing to write unlimited number of checks. These accounts are sometimes referred to as negotiable order of withdrawal (NOW) accounts.

Money market account or money market deposit account (MMDA) pays interest at a higher rate than the rate paid on savings or checking accounts with interest. Market accounts usually require a higher minimum balance for the account to start earning interest, as compared to checking or savings account. Fund withdrawals allowed per month are very limited.

Certificates of deposit (CDs) also known as **time deposits** are bank accounts that require the account holder to make a relatively large deposit and leave funds in the account for some agreed amount of time, usually several months or years. There is a penalty for early withdrawal of funds. Because of these restrictions, the interest paid on a CD is usually higher than the interest paid with other types of bank accounts.

Two special cases shown on this UML diagram are **Children's Savings Account** and **Health Savings Account** (HSA). These two accounts are both **Personal Accounts** as well as **Savings Accounts**. It is shown as **multiple inheritance**.

Children's Savings Account is a personal savings account that allows children to learn about money savings, interest rates and see what this means in relation to their savings. Some banks require some monthly fee or minimum balance and could charge fees, if an account is inactive or there are too many small deposits.

Health Savings Account (HSA) is a personal savings account that allows individuals covered by high-deductible health plans to receive tax-preferred treatment of money saved for future medical expenses.

Health insurance policy UML class diagram example

***Purpose**: Domain model describing various types of health insurance policies.*

This is an example of UML domain diagram describing some **Health Insurance Policy**.

Various health insurance choices and options for a job based, public, and private coverage, together with health insurance plans are described on a website of the U.S. Department of Health and Human Services HealthCare.gov.

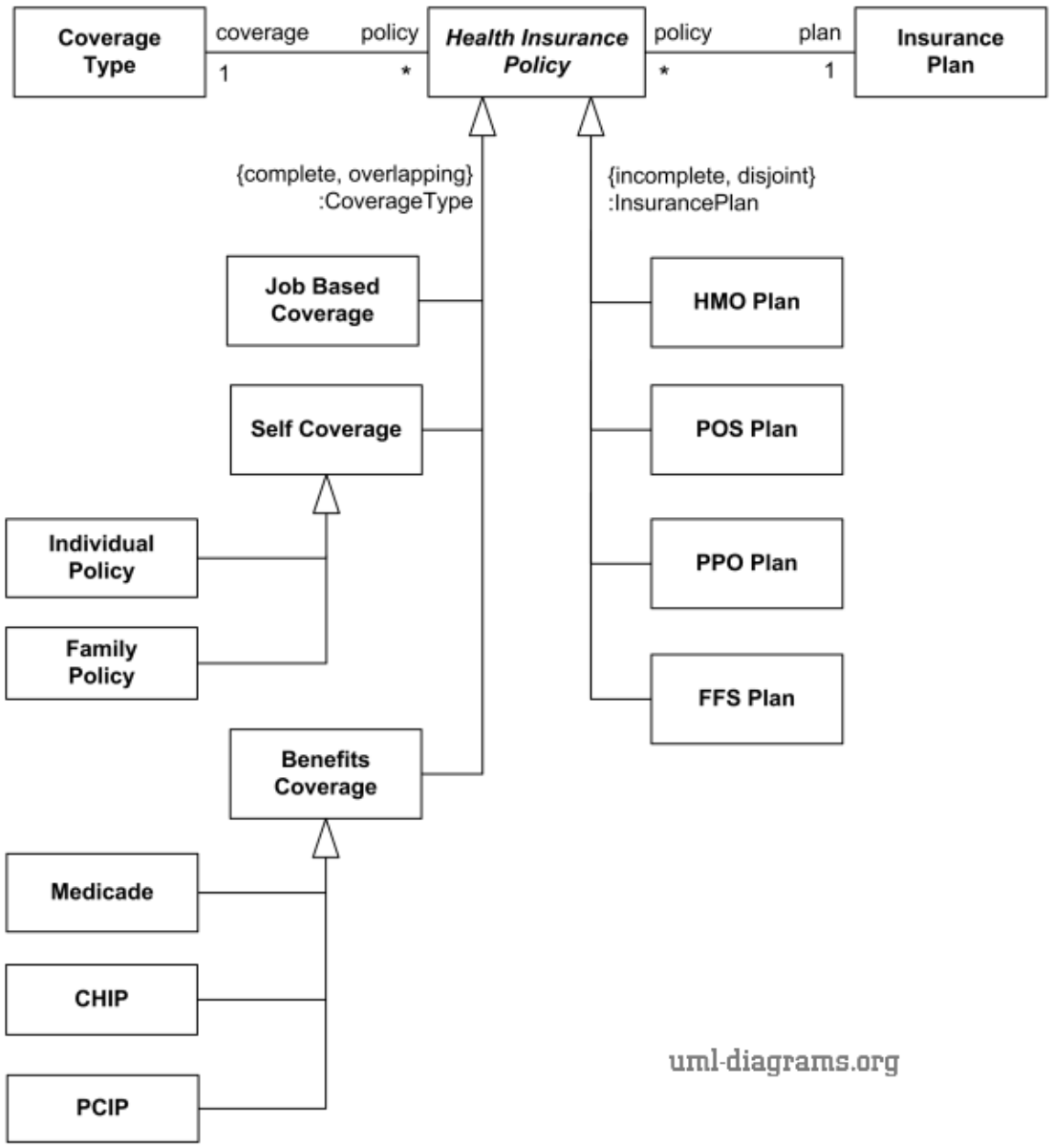
Several generalizations of the **Health Insurance Policy** could be grouped into **generalization set** **Coverage Type**. This set includes policies grouped by a coverage type - Job Based Coverage, Self Coverage, and Benefits Coverage.

People that have a job may be eligible for a health insurance coverage through work – either their own job or of their spouse or parent. If a person can't get insurance through the employer, he or she have a number of other options either through self coverage and/or by using some benefits coverage.

If person cannot get health insurance through work, he or she may be able to buy a health insurance policy for self or his/her family (individual or family insurance policies).

People who have private health insurance can also be eligible for Medicaid. This is one of the reasons for the CoverageType generalization set to have **{overlapping}** constraint.

Each US state operates a **Medicaid** program that provides health coverage for lower income people, families and children, the elderly, and people with disabilities. All states provide coverage for eligible children through Medicaid and the Children's Health Insurance Program (**CHIP**). People who have a pre-existing health condition and have been uninsured for the past six months, may qualify for the Pre-Existing Condition Insurance Plan (**PCIP**) created under the Affordable Care Act.



Health insurance policy example UML class diagram with generalization sets and power types.

Another generalization set for the **Health Insurance Policy** could be grouped by the **insurance plan**. Some common types of health insurance plans are Health Maintenance Organization (**HMO**), Point Of Service (**POS**), Participating Provider Option (**PPO**), and Fee For Service (**FFS**). Because this list is incomplete, as there are other insurance plans, Insurance Plan generalization set has **{incomplete}** constraint. Usually there is no overlapping in insurance plans, that is the reason for another **{disjoint}** constraint.

The **HMO** is one of the most affordable and common as a family health insurance choice. Usually it restricts patients to receive health care from certain "in-network" doctors and hospitals (health care providers). The **PPO** is another popular and flexible choice for families, as it provides both coverage from preferred in-network providers, while also allowing to get help from out-of-network health care providers. The **POS** plan is a combination of HMO and PPO. The **FFS** plan usually provides the same coverage from all available health care providers, while it does not work with any health care provider networks. Most services are covered because it is the most expensive health insurance plan.

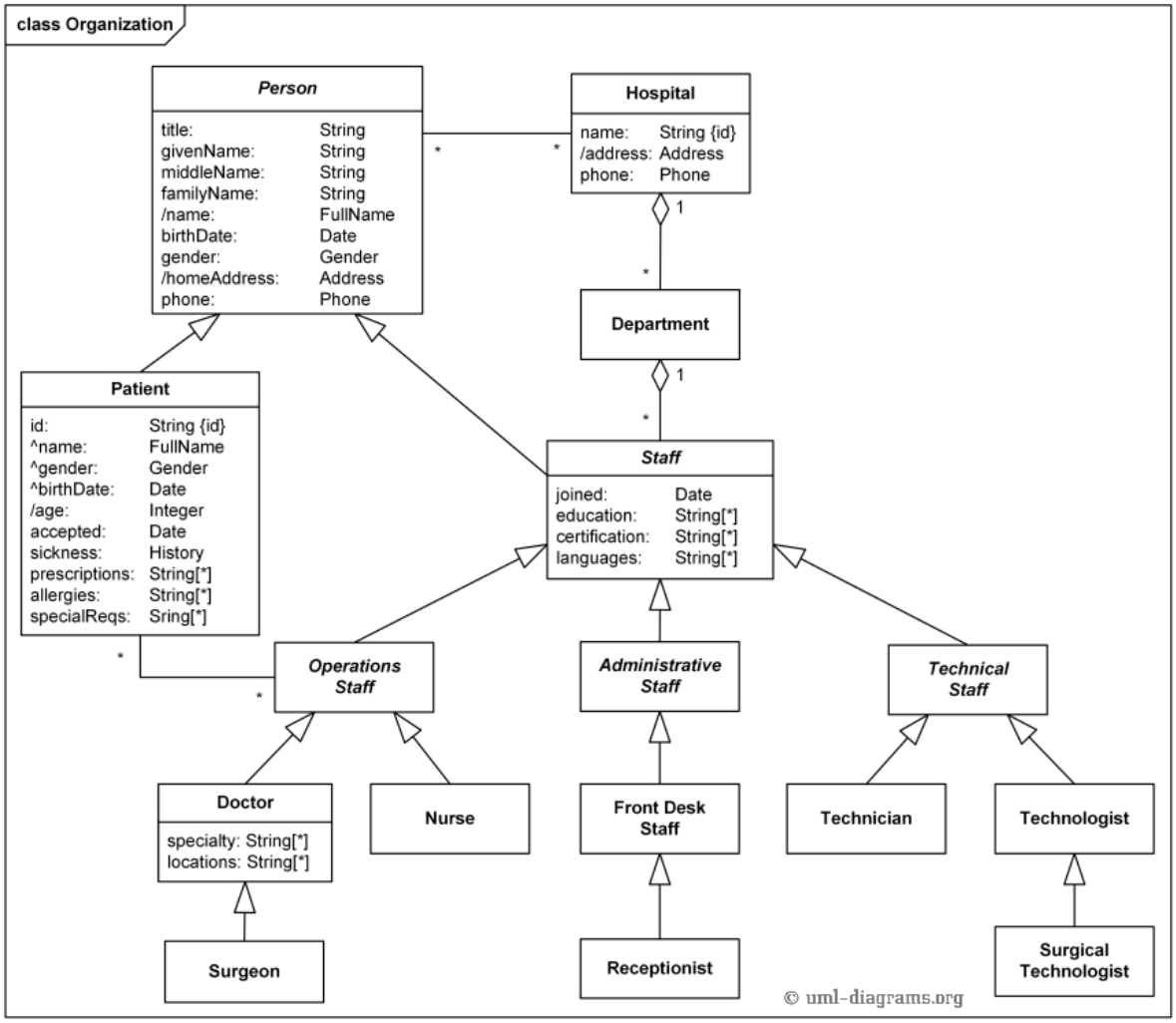
Hospital domain UML class diagram example

Purpose: *Domain model for a hospital to show and explain hospital structure, staff, relationships with patients, and patient treatment terminology.*

This is an example of a hospital domain model diagram. The domain model for the **Hospital Management System** is represented by several **class diagrams**. The purpose of the diagram is to show and explain hospital structure, staff, relationships with patients, and patient treatment terminology.

On the diagram below a *Person* could be associated with different *Hospitals*, and a Hospital could employ or serve multiple Persons. *Person* class has **derived attributes** *name* and *homeAddress*. Name represents full name and could be combined from title, given (or first) name, middle name, and family (or last) name. *Patient* class has derived attribute *age* which could be calculated based on her or his birth date and current date or hospital admission date.

The *Patient* class inherits attributes from the *Person* class. Several **inherited attributes** *name*, *gender*, and *birthDate* are shown with prepended *caret* '^' symbol (new notation introduced in UML 2.5).



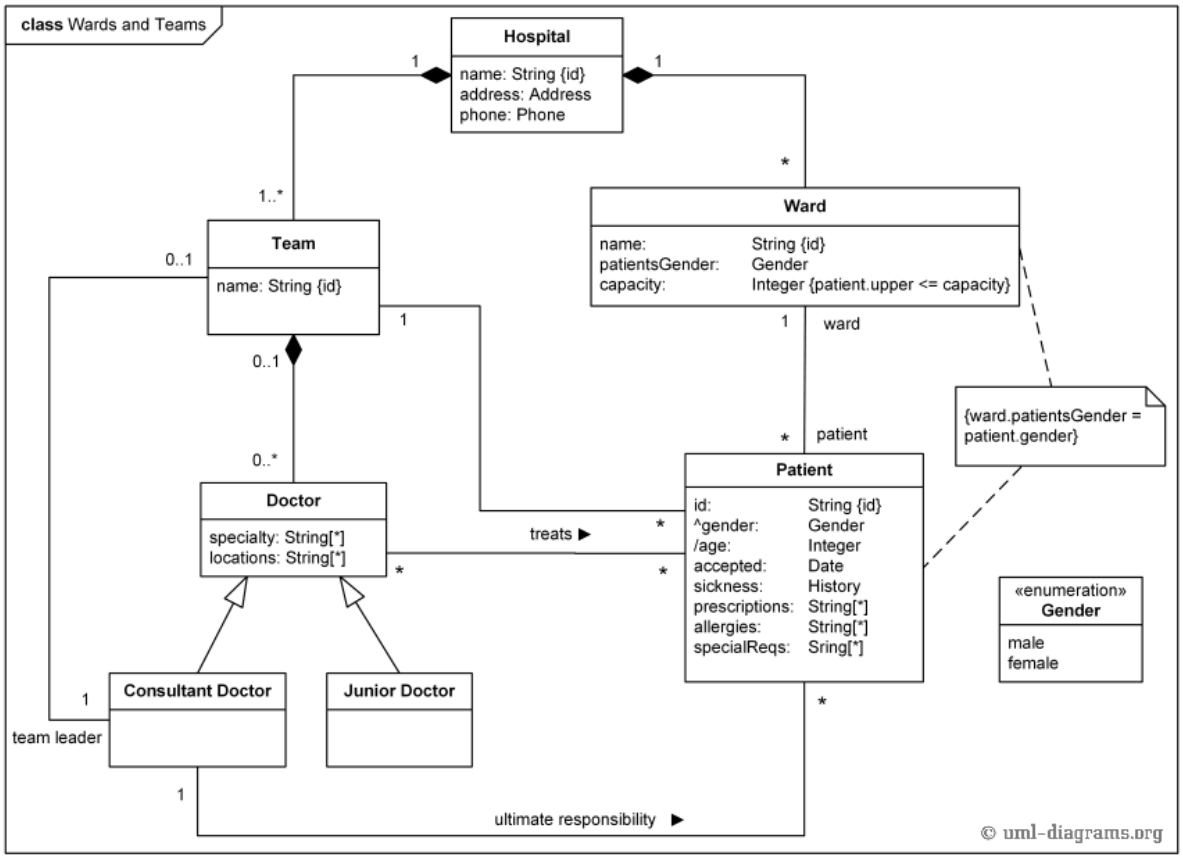
Hospital organization domain model - Patient, Hospital, Staff - Operations, Administrative, Technical.

Ward is a division of a hospital or a suite of rooms shared by patients who need a similar kind of care. In a hospital, there are a number of wards, each of which may be empty or have on it one or more **patients**. Each ward has a unique name. Diagram below shows it using **{id}** modifier for ward's name.

Wards are differentiated by **gender** of its patients, i.e. male wards and female wards. A ward can only have patients of the gender admitted to it. Gender is shown as enumeration. Ward and patient have constraint on Gender.

Every ward has a fixed capacity, which is the maximum number of patients that can be on it at one time (i.e. the capacity is the number of beds in the ward). Different wards may have different capacities.

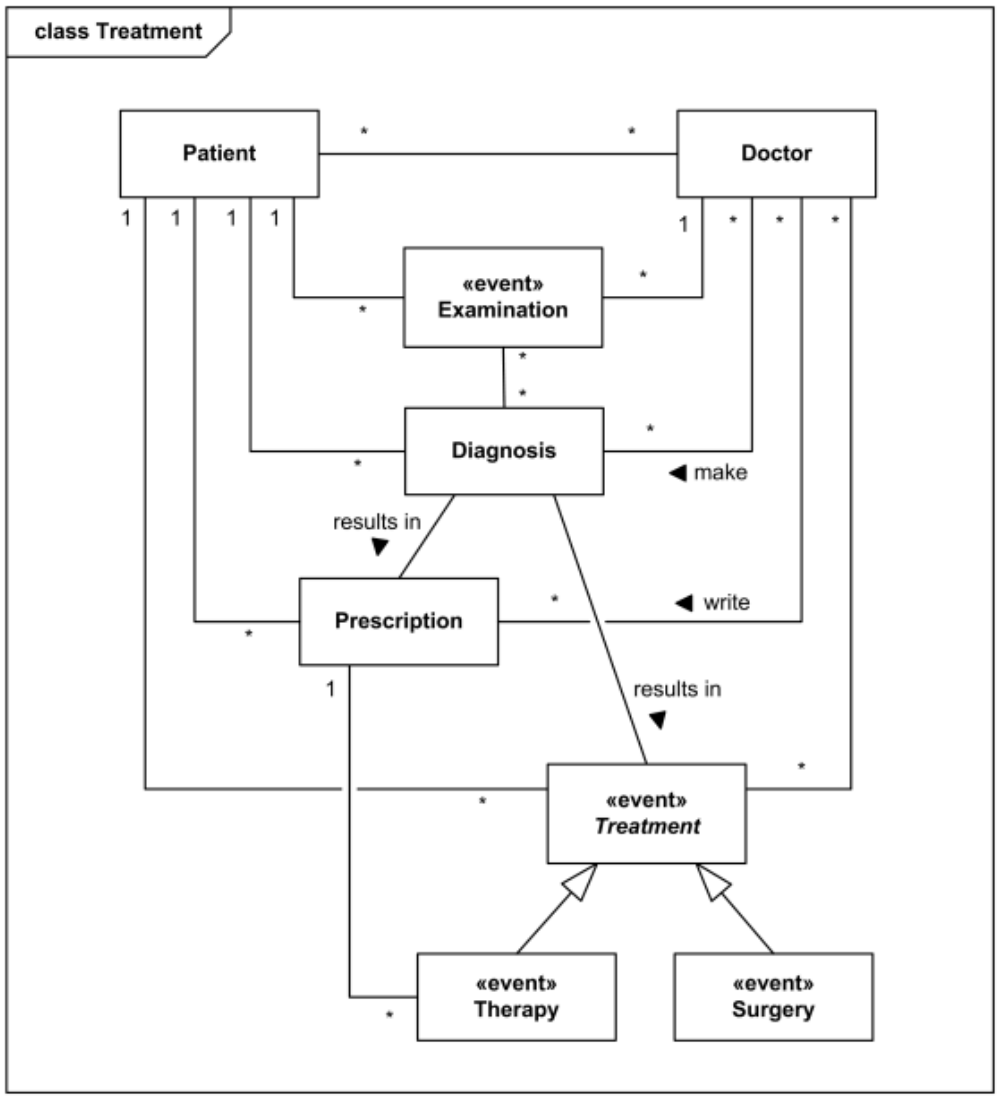
The doctors in the hospital are organised into **teams** (also called **firms**). Each team has a unique name or code (e.g. Orthopaedics or Pediatrics) and is headed by a **consultant doctor** (in the UK, Republic of Ireland, and parts of the Commonwealth) or **attending physician** (also known as staff physician) (in the United States). Consultant doctor or attending physician is the senior doctor who has completed all of his or her specialist training, residency and practices medicine in a clinic or hospital, in the specialty learned during residency. She or he can supervise fellows, residents, and medical students. The rest of the team are all junior doctors. Each doctor could be a member of no more than one team.



Hospital wards, teams of doctors, and patients.

Each **patient** is on a single ward and is under the care of a single team of doctors. A patient may be treated by any number of doctors, but they must all be in the team that cares for the patient. A doctor can treat any number of patients. The team leader accepts ultimate responsibility, legally and

otherwise, for the care of all the patients referred to him/her, even with many of the minute-to-minute decisions being made by subordinates.



Domain model - Patient, Doctors and Treatments.

Digital imaging in medicine - DICOM model of the real world

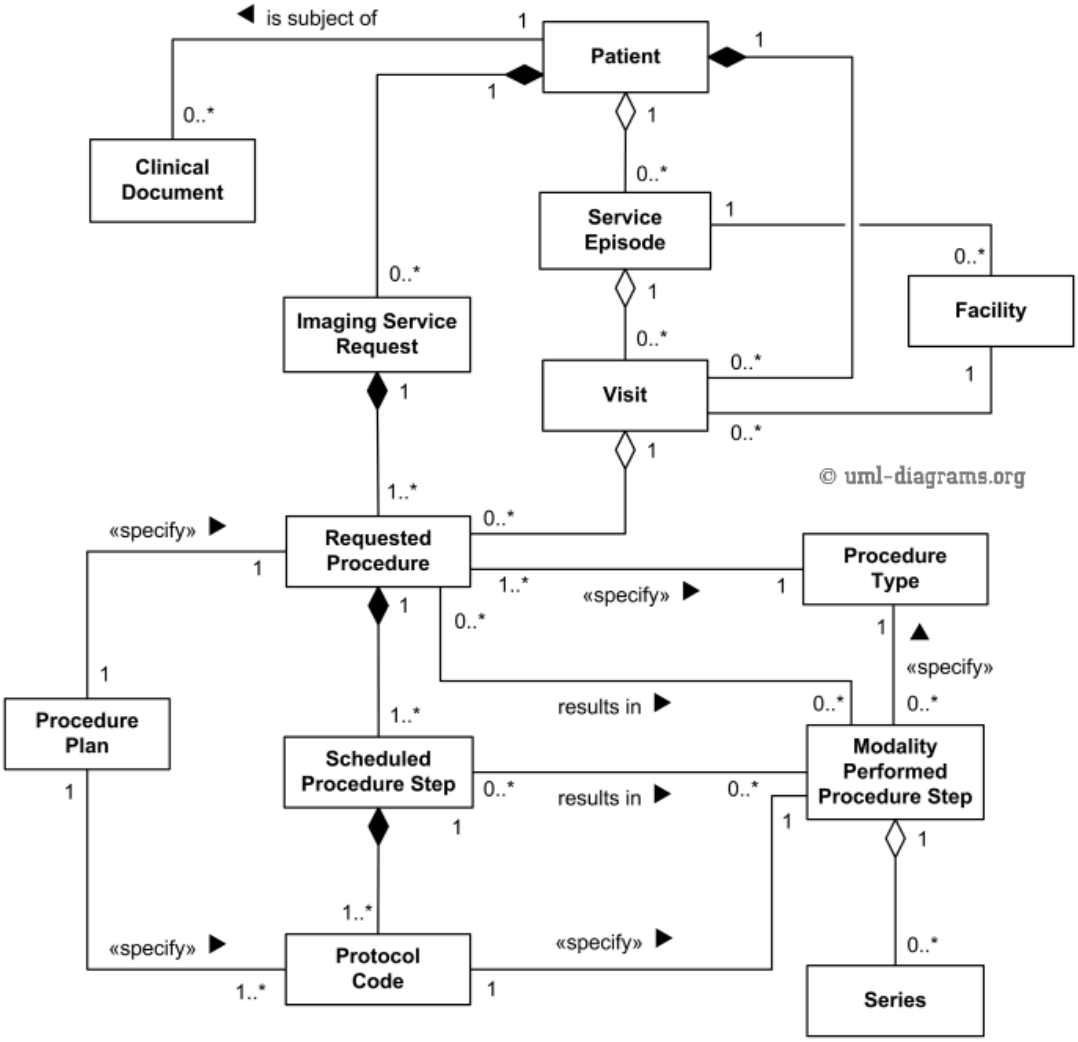
***Purpose**: Represent domain model ("model of the real world") for Digital Imaging and Communications in Medicine (**DICOM**) - Patient, Visit, Facility, Imaging Service Request, Scheduled Procedure Step, Modality Performed Procedure Step.*

An example of class diagram representing domain model ("model of the real world") for Digital Imaging and Communications in Medicine (**DICOM**). This diagram is based on E-R model Fig.7-3 of DICOM standard Part 3 - PS 3.3-2009.

Diagram represents DICOM extended domain, abstract description of the real-world objects used in the Modality-IS Interface. Modality is a piece of imaging equipment, e.g. computed tomography (CT) or ultrasound (US).

A **Patient** is a human or an animal receiving, or registered to receive, healthcare services, or is the subject of research studies. A **Clinical Document** is a part of the medical record of a patient. It is a documentation of clinical observations and services provided to the patient.

A **Service Episode** is a collection of events, context in which the treatment or management of an arbitrary subset of a Patient's medical conditions occurs. Service episode is entirely arbitrary; it may include a single outpatient visit or a hospitalization, or extend over significant period, e.g., the duration of a pregnancy, or an oncology treatment regimen. A service episode may involve one or more **Healthcare Organizations** (administrative entities that authorize **Healthcare Providers** to provide services within their legal administrative domain, e.g. hospitals, private physician's offices, multispecialty clinics, nursing homes).



DICOM domain model for Modality-IS interface UML class diagram example.

Visit is a part of service episode, collection of events that fall under the accountability of a particular Healthcare Organization in a single facility. A visit may be associated with one or more physical locations (e.g. different rooms, departments, or buildings) within the same facility.

An **Imaging Service Request** is a set of one or more **Requested Procedures** selected from a list of **Procedure Types**. An Imaging Service Request is submitted by one authorized imaging service requester to one authorized imaging service provider in the context of one **Service Episode**. An Imaging Service Request may be associated with one or more **Visits** that occur within the same Service Episode.

A modality **Scheduled Procedure Step** is an arbitrarily defined scheduled unit of service, that is specified by the Procedure Plan for a Requested Procedure. It prescribes **Protocol** which may be identified by one or more **Protocol Codes**. A modality Scheduled Procedure Step involves equipment (e.g. imaging equipment, surgical equipment, etc.), human resources, supplies, location, and time.

A **Modality Performed Procedure Step** is an arbitrarily defined unit of service that has actually been performed (not just scheduled). It contains references to zero or more **Series** of **Images**.

Digital imaging in medicine - DICOM Application Hosting API

***Purpose**: An example of UML class diagram representing **DICOM Application Hosting API**, defined in Part 19 of DICOM Standard (PS 3.19-2011). The Application Hosting API describes interfaces between two software applications - **Hosting System** and **Hosted Application**, exchanging medical data while located on the same system.*

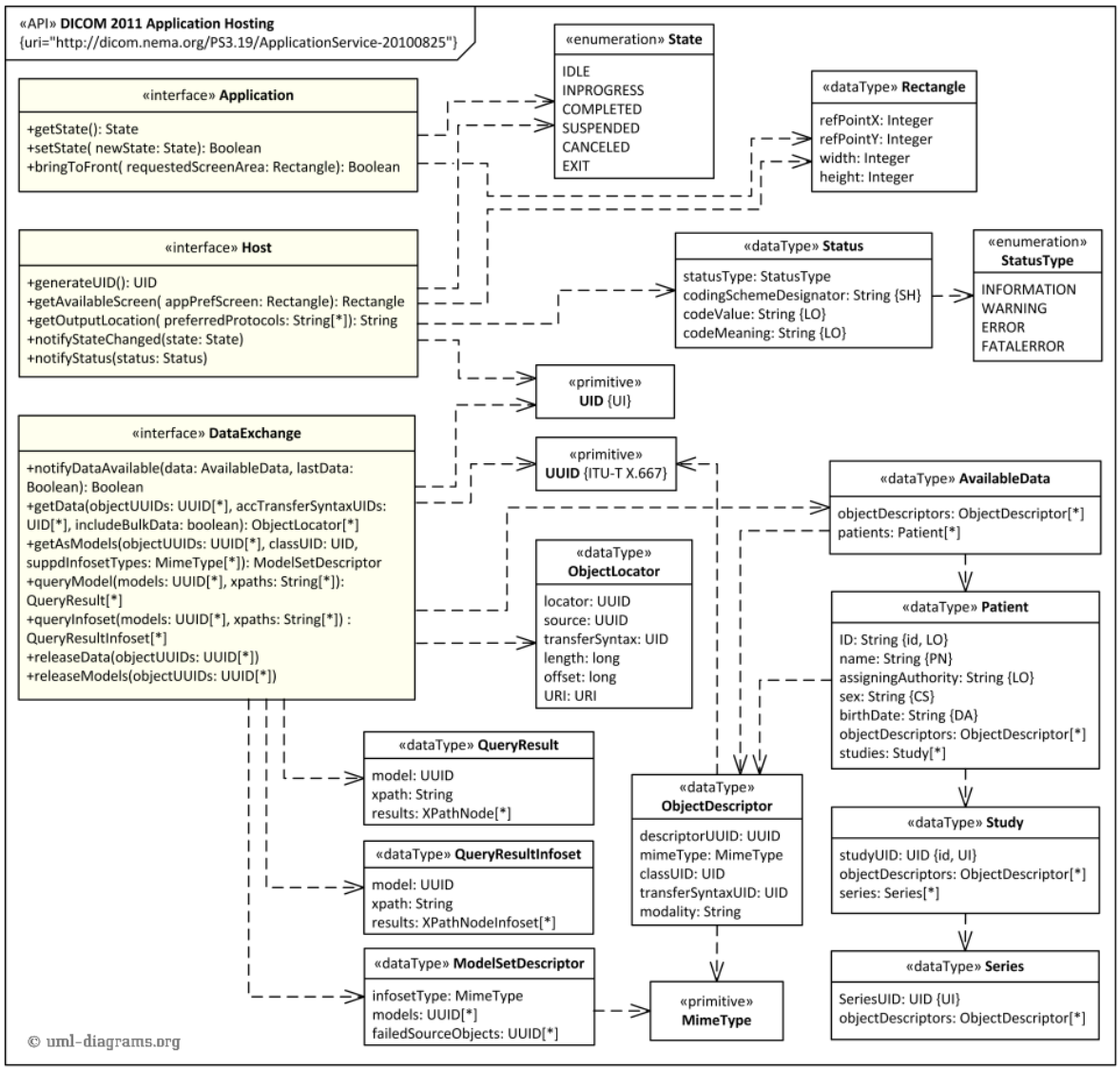
An example of class diagram representing **DICOM Application Hosting API**. The DICOM Application Hosting API was published in Digital Imaging and Communications in Medicine (DICOM) Standard, Part 19 Application Hosting (PS 3.19-2011). Example shown below is our own interpretation of the API, please refer to the DICOM Standard for official description and diagrams.

The Part 19 of the DICOM Standard defines an application programming interface (API) between two software applications - **Hosting System** and **Hosted Application**, both applications exchanging medical data while located on the same system.

The **Hosting System (Host)** (aka Host Application) is an application used to launch and control hosted applications. The Hosting System provides a variety of services such as DICOM object retrieval and storage for the hosted application. The hosting system provides the infrastructure in which the hosted application runs and interacts with the external environment. This includes network access, database and security. Host application provides the hosted one with data, such as a set of medical images and related metadata.

The **Hosted Application (Application)** is an application launched and controlled by a hosting system, which may also utilize some services offered by the hosting system. Hosted application processes provided medical data, potentially returning back some newly generated data, for example in the form of another set of images and/or structured reports, to the hosting application.

The hosting system has a mechanism to launch or connect to one or more hosted applications, verify that the hosted application has started successfully, and then pass the initial data objects. All interactions start in the hosting system.



DICOM Application Hosting API UML class diagram example.

There are three interfaces defined by the DICOM Application Hosting API.

The **Application** interface encapsulates the hosted application and is invoked by the hosting system to control the hosted application. The getState() method allows to query hosted application of its current state. Hosting system could request hosted application to switch to a new state using setState() method. By calling bringToFront() method, the hosting system could ask hosted application to make its GUI visible as the topmost window, and to gain focus.

The **Host** interface represents the hosting system and is utilized by the hosted application to request services from and to notify the Hosting System of events during the execution of the Hosted Application. For example, hosting system could generate new DICOM UID for the hosted application to use. The hosted application may inform the hosting system of notable events that occur during execution by invoking notifyStatus() method.

The **DataExchange** interface is an interface which is used by both the hosting system and the hosted application to exchange medical data directly or indirectly, and thus must be implemented by both. The data being exchanged between the hosting system and the hosted application can either be passed

as files or may be described in models that might be queried by the recipient. Model-based methods can work with a variety of models which can be either some abstraction of the data, or can be a model of some native format, including models defined by the DICOM Standard. Particular models are identified by a UID and described as XML Infosets, even though the original data might never be actually represented in XML form.

The DICOM Application Hosting API uses **ArrayOf[Type]** wrapper class to represent an array of a specific type "to enable cross-platform compatibility". Our example UML diagram provided here uses standard UML multiplicity instead.

Sentinel HASP software licensing domain UML class diagram example

***Purpose**: The purpose of the domain diagram is to show major "things" used during software licensing and protection process using Sentinel HASP, and relationships between those things.*

An example of UML **class diagram** which provides some simplified view of software licensing domain for the SafeNet **Sentinel HASP** Software Licensing Security Solution.

The purpose of the domain diagram is to show major "things" used during software licensing and protection using Sentinel HASP and relationships between those things.

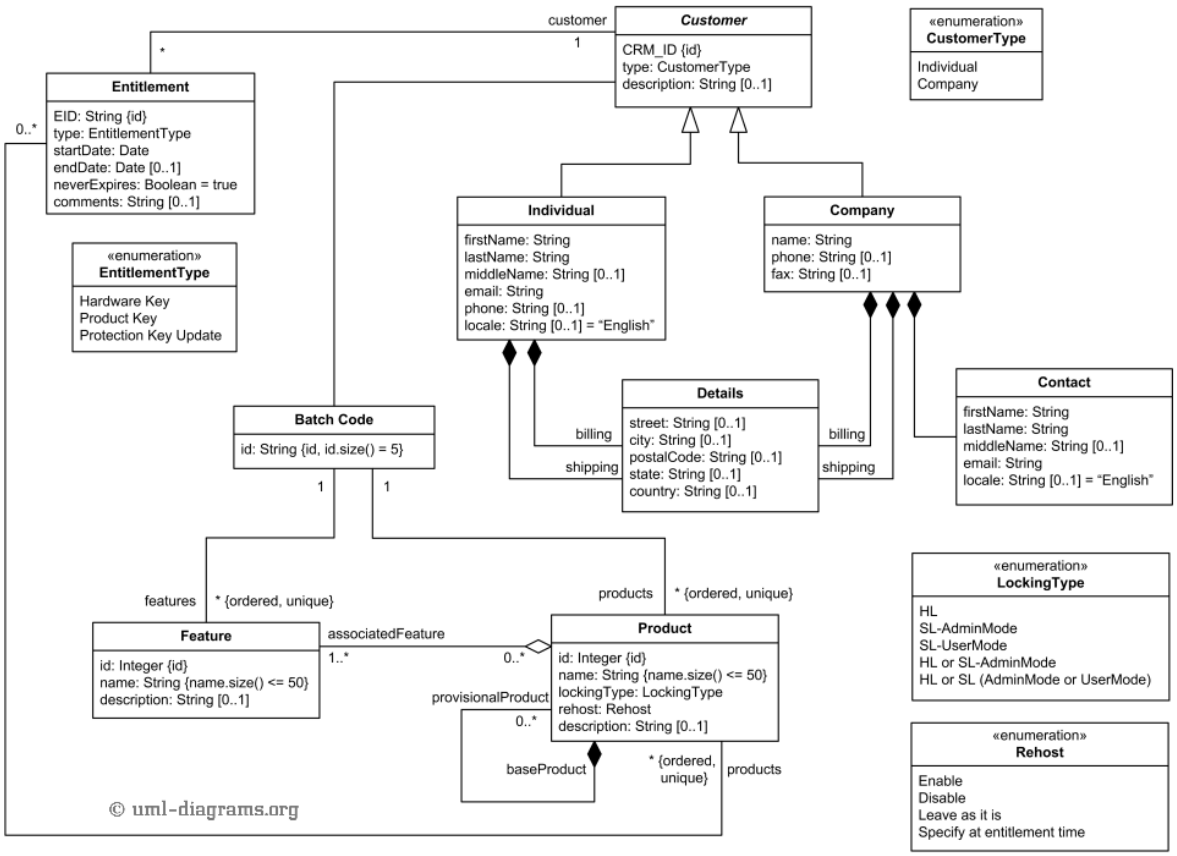
When software vendor purchases a Sentinel HASP License Development Kit (LDK), the vendor is also provided with a unique **batch code** and corresponding **vendor key** that contains unique vendor code specific to the company. The code is used by Sentinel LDK to communicate with protection keys, and to differentiate vendor keys from those of other software vendors.

A **batch code** consists of five characters that represent company's unique vendor code. When Sentinel protection keys are ordered from SafeNet, batch code is written to the keys before sending to the vendor. The batch code is also written on the outside of each key.

A **feature** is some identifiable functionality of a software application. Features may used to identify entire executables, software modules, .NET or Java methods, or a specific functionality such as print or save. Each feature is assigned unique identifier and is associated with a specific batch code. Feature id and feature name both should be unique in the associated batch. The maximum length for a feature name is 50 characters.

Each protected software **product** has some features and is associated with a batch code. A product name that identifies the product is unique in the selected batch. The maximum length for a product name is 50 characters. Product has some specific **locking type** (type type of protection) to be applied to the product.

The basic unit on which all products are created is the **base product**. A base product can contain all the product attributes such as features, licensing data and memory, and can be used as a product offered for sale. **Provisional products** could be also defined to be distributed as trialware. The properties for provisional products are not completely identical to those of the base product. After a product has been defined, it can be included in **entitlements** (orders).



An example of UML domain (class) diagram for Sentinel HASP Software Licensing Security Solution.

An **entitlement** is an order for products to be supplied with one or more Sentinel protection keys. Orders or sales department receives and fulfils entitlements. Order processing personnel process the entitlement details using Sentinel EMS. An entitlement can contain one or more products. When entitlement is defined in Sentinel EMS, they can specify the **customer** who placed the order. The customer could be either an **individual** customer or a **company**.

The locking type assigned to a product may determine the type of entitlement that can be produced. They cannot add a product defined only with the HL locking type and another product defined only with the SL locking type (whether AdminMode or UserMode) to the same entitlement:

- Products defined only with the HL locking type can be included in entitlements for HASP HL keys, product keys, or for protection key updates.

- Products defined only with the SL AdminMode or SL UserMode locking type can be included only in entitlements for product keys or for protection key updates.

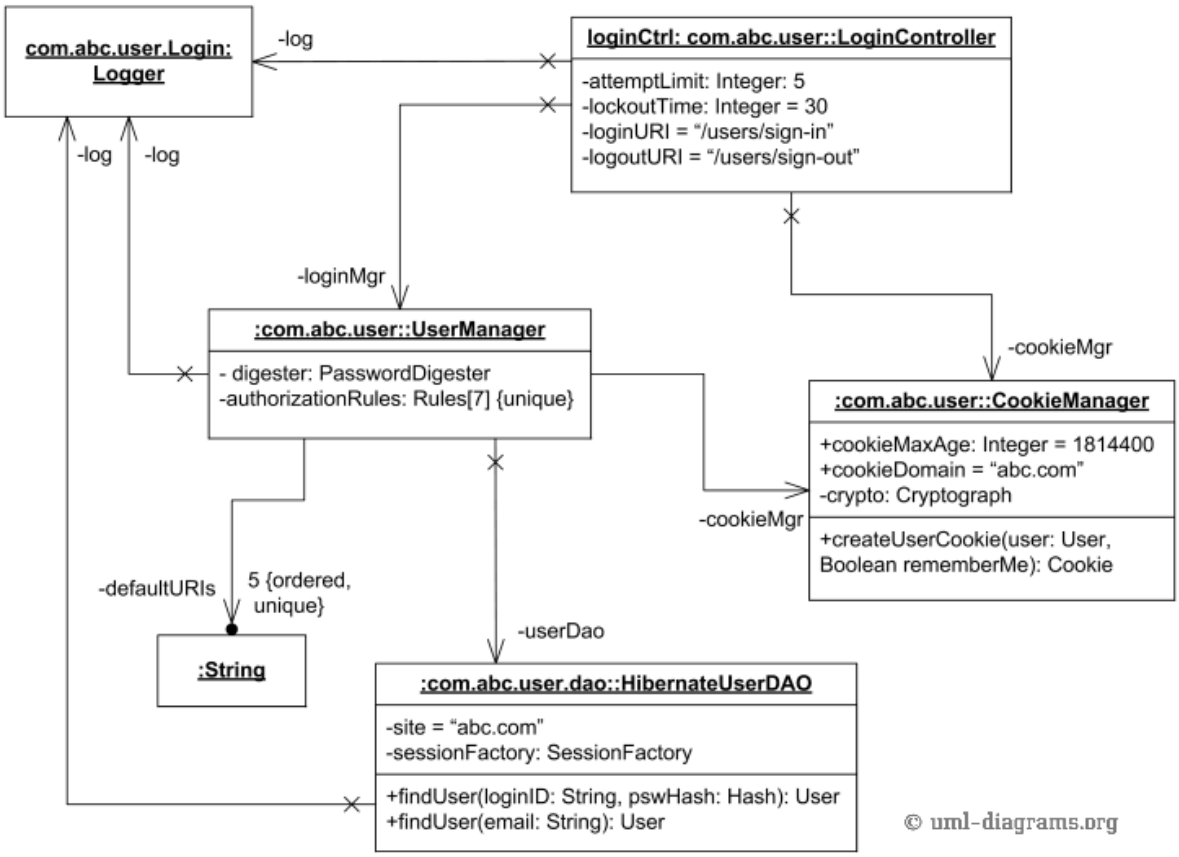
- Products defined with the HL or SL AdminMode or HL or SL AdminMode or SL User-Mode locking type can be included in entitlements for HASP HL keys, product keys, or for protection key updates.

Web application Login Controller object diagram

***Purpose**: An example of UML object diagram which shows some runtime objects involved in the login process for a web user.*

This is an example of **object diagram** which shows some runtime objects related to web user login process. Class **instance** loginCtrl of the LoginController has several **slots** with **structural features** of Integer and String types and corresponding value specifications.

The instance of LoginController is also associated with instances of UserManager, CookieManager, and Logger. LoginController, UserManager, and HibernateUserDAO (Data Access Object) share a single instance of Logger.



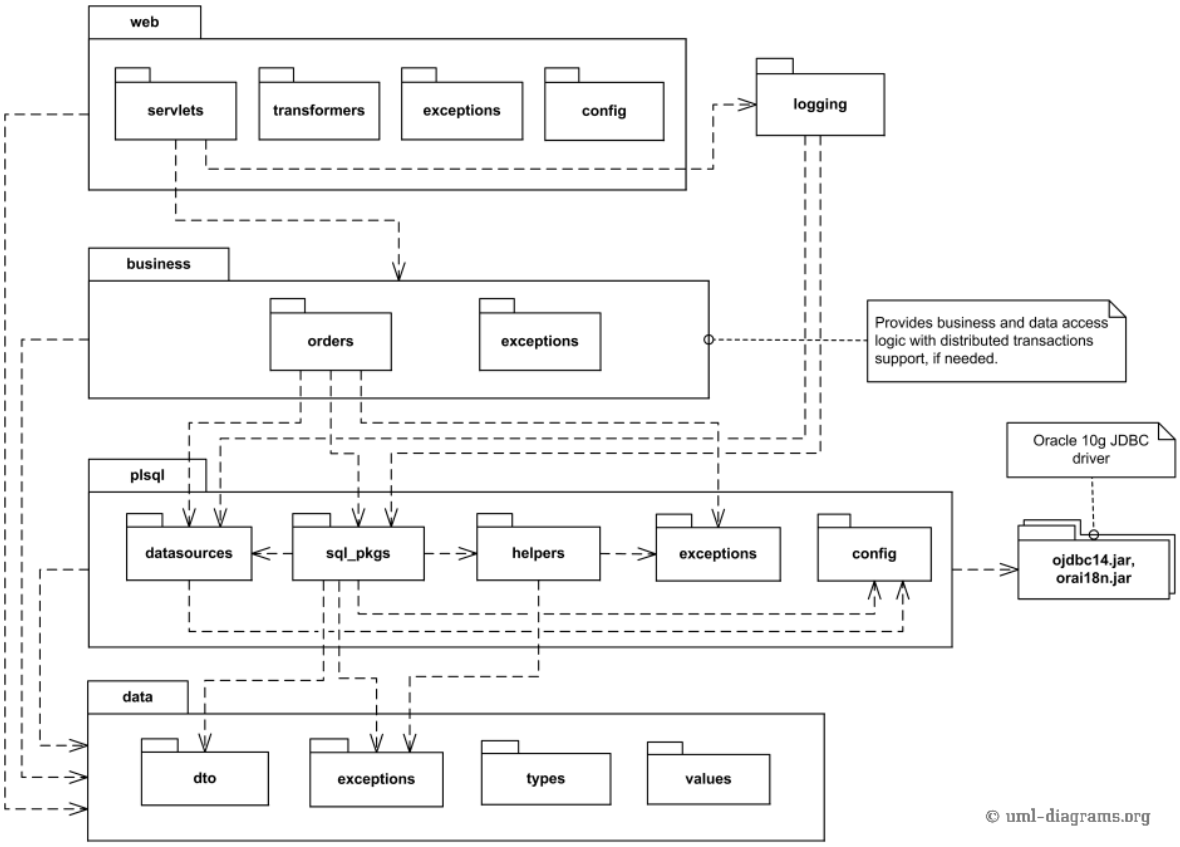
User login controller UML object diagram example.

UserManager has private attribute defaultURIs which is ordered collection (array) of 5 unique Strings. Instance of the CookieManager has two public structural features with specified values. Most **links** are non-navigable backwards.

Multi-Layered Web Architecture

UML Package Diagram Example

An example of UML **package diagram** representing some multi-layered web architecture. **Dependencies** between packages are created in such a way as to avoid circular dependencies between packages. Higher level packages depend on lower level packages. Packages belonging to the same level could depend on each other. Data transfer objects and common exceptions are used by packages at higher levels.



UML package diagram example of a multi-layered web architecture.