UML Distilled Brief Guide by Martin Fowler

Ertale81

Table of Contents

1. What is UML?

- The Unified Modeling Language(UML) is a family of graphical notations that helps describing and designing software system, particularly software built using the object oriented(OO) style.

1.1. Ways of using the UML

1. **UML as sketch:**
    - **Forward engineering** draws a UML diagram before you wrote a code, while **reverse engineering** builds a UML diagram from existing code in order to help understand it.
    - The essence of sketching is selectivity. With **forward** sketching, you rough out some issues in code you are about to write. Your aim is to use the sketches to help communicate ideas and alternatives about what you are about to do. You don't talk about

all the code you are going to work on, only important issues that you want to run past your colleagues first or sections of the design that you want to visualize before you begin programming.

- With **reverse engineering**, you use sketches to explain how some parts of the system woks. You don't show every class, simply those that are interesting and worth talking about before you dig into the code.
- The tools used for sketching are lightweight drawing tools, and often people aren't too particular about keeping to every strict rule of the UML.
- In sketching the focus is communication rather than completeness.

2. **UML as blueprint:**
   - UML as blueprint is about completeness.
   - **In forward engineering**, the idea is that blueprints are developed by a designer whose)ob is to build a detailed design for a programmer to code up. That design should be sufficiently complete in that all design decisions are laid out, and the programmer should be able to follow it as a pretty straightforward activity that requires little thought. The designer may be the same person as the programmer, but usually the designer is a more senior developer who designs for a team of programmers. The inspiration for this approach is other forms of engineering in which Professional engineers create engineering drawings that are handed over to construction companies to build.
   - A common approach is for a designer to develop blueprint-level models as far as interfaces of subsystems but then let developers work out the details of implementing those details.
   - **In reverse engineering**, blueprints aim to convey detailed information about the code either in paper documents or as an interactive graphical browser . The blueprints can show every detail about a class in a graphical form that's easier for developers to understand.
   - Blueprints require much more sophisticated tools than sketches do in order to handle the details required for the task . Specialized CASE (computer-aided software engineering) tools fall into this category.
   - Forward-engineering tools support diagram drawing and back it up with a repository to hold the information. Reverse-engineering tools read source code and interpret from it to the repository and generate diagrams. Tools that can do both forward and reverse engineering like this are referred to as a **round-trip** tools.
   - The line between blueprints and Sketches is somewhat blurry, but the distinction, I think, rests an the fact that sketches are deliberately incomplete, highlighting important information, while blueprints intend to be comprehensive, often with the aim of reducing programming to a simple and fairly mechanical activity. In a sound bite, I'd say that sketches are explorative, while blueprints are definitive.
   - As you do more and more in the UML and the programming gets increasingly mechanical, it becomes obvious that the programming should be automated. Indeed, many CASE tools do some form of code generation, which automates building a significant part of a system. Eventually, however, you reach the point at which all the System can be specified in the UML, and you reach UML as programming language.
3. **UML as programming language:**

- In this environment, developers draw UML diagrams that are compiled directly into executable code, the UML becomes the source code.Obviously, this usage of UML demands particularly sophisticated tooling . (Also, the notions of forward and reverse engineering don't make any sense for this mode, as the UML and source code are the same thing.)

1.2. UML Diagrams

- UML 2 describes 13 official diagram types listed in the following table:

No	**Diagram**	**purpose**
1	Activity	Procedural and parallel behavior
2	Class	Class, features, and relationships
3	Communication	Interaction between objects; emphasis on links
4	Component	Structure and connections of components
5	Composite Structure	Runtime decomposition of a class
6	Deployment	Deployment of artifacts to nodes
7	Interaction overview	Mix of sequence and activity diagram
8	Object	Examples of configuration instances
9	Package	Compile-time hierarchic structure
10	Sequence	Interaction between objects; emphasis on sequence
11	State machine	How events change an object over its life
12	Timing	Interaction between objects; emphasis on timing

No	Diagram	purpose
13	Use case	How users interact with a system

- Those above diagrams can be categorized into to main diagrams:
    1. **Structure Diagram:**
        - This category includes: class diagram, component diagram, composite structure diagram, deployment diagram, object diagram and package diagram.
    2. **Behavior diagram:**
        - This category contains the following diagrams:
            - Activity diagram, use case diagram, state machine diagram
            - Interaction diagram: this category includes sequence diagram, communication diagram, timing diagram, and interaction overview diagram.
- Where to start with UML? Concentrate first on the basic forms of class diagrams and sequence diagrams. These are the most common and most useful diagram types.

2. Development Process

2.1. Iterative and Waterfall process

- The essential difference between the two is how you break up a project into smaller chunks. For example, if you have a project that you think will take a year, few people are comfortable telling the team to go away for a year and come back when done. Some break down is needed so that people can approach the problem and track the progress.
- The **waterfall** style breaks down a project based on activity. To build a software, you have to do certain activities: requirement analysis, design, coding, and testing. A 1-year project might thus have a 2-month analysis phase, followed by a 4-month design phase, followed by a 3-month coding phase, followed by a 3-month testing phase.
- The **iterative** style breaks down a project by subset of functionality. You may take a year and break into 3-month iterations. In the first iteration, you would take quarter of the requirements and do the complete software life cycle for that quarter: analysis, design, code and test. At the end of the first iteration, you would have a system that does a quarter of the needed functionality. Then you would do a second iteration, so that at the end of 6 months, you'd have a system that does half of the functionality.
- The above is a simplified description, but it is the essence of the difference.
- With waterfall development, there is usually some form of formal handoff between each phase, but there are often backflows. During coding, something may come up that causes you to revisit the analysis and design. You certainly should not assume that all design is finished when coding begins. It's inevitable that analysis and design decisions will have to be revisited in later phases. However, these backflows are exceptions and should be minimized as much as possible.
- With iteration, you usually see some form of exploration activity before the true iterations begin. You may well not put the system into production at the end of each iteration, but the

system should be production quality.

- Iterative development has come under many names: incremental, spiral, evolutionary, and jacuzzi spring. various people make distinctions among them, but the distinctions are neither widely agreed an nor that important compared to the iterative/waterfall dichotomy.
- You can have hybrid approaches. In a **staged delivery,** analysis and high-level design are done first, a waterfall style, and then coding and testing are divided up into iterations.
- A common technique with iteration is to use **time boxing**. This forces an iteration to be a fixed length of time. If it appears that you can't build all you intended to build during an iteration, you must decide to slip some functionality from the iteration; you must not slip the date of the iteration.
- One of the most common concerns about iterative development is the issue of rework. Iterative development explicitly assumes that you will be reworking and deleting existing code during the later iterations of a project. In many domains, such as manufacturing, rework is seen as a waste. But software isn't like manufacturing; as a result, it often is more efficient to rework existing code than to patch around code that was poorly designed. A number of technical practices can greatly help make rework be more efficient.
    - **Automated regression tests** help by allowing you to quickly detect any defects that may have been introduced when you are changing things. The xUnit family of testing frameworks is a particularly valuable tool for building automated unit tests. e.g., JUnit test. A good rule of thumb is that the size of your unit test code should be about the same size as your production code.
    - **Refactoring** is a disciplined technique for changing existing software. Refactoring works by using a series of small behavior-preserving transformations to the code base. Many of these transformations can be automated.
    - **Continuous integration** keeps a team in sync to avoid painful integration cycles. At the heart of this lies a fully automated build process that can be kicked off automatically whenever any member of the team checks code into the code base.
- All these technical practices have been popularized recently by Extreme Programming(XP).

2.2. Predictive and Adaptive planning

- Nothing is more frustrating than not having a clear idea how much it will cost to build some software and how long it will take to build it.
- One of the unique sources of complexity in software projects is the difficulty in understanding the requirements for a software system. The majority of software projects experience significant **requirements churn:** changes in requirements in the later stages of the project. These changes shatter the foundation of a predictive plan. You can combat these changes by freezing the requirements early on and not permitting changes, but this runs the risk of delivering a system that no longer meets the needs of its user.
- This problem leads to two very different reactions:
    1. Put more effort into the requirements process itself.
    2. requirement churn is unavoidable, that is too difficult for many projects to stabilize requirements sufficiently to use a predictive plan. This may be either owing to the sheer difficulty of envisioning what software can do or because market conditions force unpredictable changes. This thought advocates **adaptive planing**, whereby predictivity

is seen as an illusion.

- Instead of fooling ourselves with illusory predictability, we should face the reality of constant change and use a planning approach that treats change as a constant in a software project. This change is controlled so that the project delivers the best software it can; but although the project is controllable, it is not predictable.
- When people talk about a project that's doing well because it is going according to plan, that is predictive form of thinking. You can't say "according to plan" in adaptive environment, because the plan is always changing. This doesn't mean that adaptive projects don't plan; they usually plan a lot, but the plan is treated as a baseline to assess the consequences of change rather than as a prediction of the future.
- With a predictive plan, you can develop a fixed-price/fixed-scope contract. Such a contract says exactly what should be built, how much it will cost, and when it will be delivered. Such fixing isn't possible with an adaptive plan. You can fix a budget and a time for delivery, but you can't fix what functionality will be delivered. An adaptive contract assumes that the users will collaborate with the development team to regularly reassess what functionality needs to be built and will cancel the project if progress ends up being too slow. As such, an adaptive planning process can be fixed price/variable scope.
- Naturally, the adaptive approach is less desirable, as anyone would prefer greater predictability in a software project. However, predictability depends an a precise, accurate, and stable set of requirements. If you cannot stabilize your requirements, the predictive plan is based an sand and the chances are high that the project goes off course. This leads to two important pieces of advice:
    1. Don't make a predictive plan until you have precise and accurate requirements and are confident they won't significantly change.
    2. If you can't get precise, accurate, and stable requirements, use an adaptive planning styles.
- An adaptive plan absolutely requires an iterative process. Predictive planning can be done either way, although it easier to see how it works with waterfall or staged delivery approach.

2.3. Agile Processes

- *Agile* is an umbrella term that covers many processes that share a common set of values and principles as defined by the **Manifesto of Agile Software Development**. Examples of these processes are:
    - Extreme Programming(XP)
    - Scrum
    - Feature Driven Development(FDD)
    - Crystal
    - Dynamic Systems Development Method(DSDM)
- Agile processes are strongly adaptive in their nature. They are also very much people-oriented processes. Agile approaches assume that the most important factor in a project's success is the quality of the people on the project and how well they work together in human terms. which process they use and which tools they use are strictly second-order effects.
- Agile methods tend to use short, time-boxed iterations, most often of a month or less.

Because they don't attach much weight to documents, agile approaches disdain using the UML in blueprint mode.

2.4. Fitting a process into a project

2.5. Fitting UML into a process

2.5.1. Requirement Analysis

- The activity of requirement analysis involves trying to figure out what the users and customers of a software effort want the system to do. A number of UML techniques can come in handy here:
  - Use cases, which describes how people interact with system
  - A class diagram drawn from the conceptual perspective, which can be a good way of building up a rigorous vocabulary of the domain.
  - An activity diagram, which can show the work flow of the organization, showing how software and human activities interact.
  - A state diagram, which can be useful if a concept has an interesting life cycle, with various states and events that change that state.
- When working in requirements analysis, remember that the most important thing is communication with your users and customers.

2.5.2. Design

- When you are doing design, you can get more technical with your diagrams. You can use more notation and be more precise about your notation. Some useful techniques are:
  - Class diagrams from a software perspective. These show the classes in the software and how they interrelate.
  - Sequence diagrams for common scenarios. A valuable approach is to pick the most important and interesting scenarios from the use cases and use CRC cards or sequence diagrams to figure out what happens in the software.
  - Package diagrams to show the large-scale organization of the software.
  - State diagrams for classes with complex life histories.
  - Deployment diagrams to show the physical layout of the software.

2.5.3. Documentation

- Once you have built the software, you can use the UML to help document what you have done.

3. Class Diagrams: The Essentials

- The majority of UML diagrams we see are class diagrams.
- A **class diagram** describes the types of objects in the system and the various kinds of static relationship that exist among them. Class diagrams also show the properties and operations

of a class and the constraints that apply to the way objects are connected. The UML uses the term **feature** as a general term that covers properties and operations of a class.

3.1. Property

- Properties represent structural features of a class. You can think of properties as corresponding to fields in a class. Properties are a single concept, but they appear in two quite distinct notations: **attributes** and **associations**.
- **Attributes**: the attribute notation describes a property as a line of text within the class box itself. The full form of an attribute is: *visibility name: type multiplicity = default {property-string}*.
- An example of this is: **- name: String [1] = "Untitled" {readOnly}** Only the name is necessary.
    - The visibility marker indicates whether the attribute is public(+) or private(-)
    - The *name* of the attribute – how the class refers to the attribute – roughly corresponds to the name of the field in programming language.
    - The *type* of the attribute indicates a restriction on what kind of object may be placed in the attribute. You can think of this as the type of a field in programming language.
    - *Multiplicity* of a property is an indication of how many objects may fill the property.
    - The *default value* is the value for a newly created object if the attribute isn't specified during creation.
    - The {property-string} allows you to indicate additional properties for the attribute. In the example I used {readOnly} to indicate that clients may not modify the property. If this is missing, you can usually assume that the attribute is modifiable.
- **Associations**: this is another way to notate a property is an association. Much of the same information that you can show on an attribute appears on an association. An association is a solid line between two classes, directed from directed from the source class to the target class. The name of the property goes at the target end of the association, together with its multiplicity. The target end of the association links to the class that is the type of the property.
- **Multiplicity**: The multiplicity of a property is an indication of how many objects may fill the property. The most common multiplicities you will see are:
    - **1** (An order must have exactly one customer)
    - **0..1** (A corporate customer may or may not have a single sales rep.)
    - ***** (A customer need not place a Order and there is no upper limit to the number of Orders a customer may place– zero or more Orders)

3.2. Operations

- Operations are the actions that a class knows to carry out. Operations most obviously correspond to the methods on a class. Normally, you don't show those operations that simply manipulate properties, because they can usually be inferred.
- The full UML syntax for operation is: *visibility name (parameter-list) : return-type {property-string}*

    - The *visibility* marker is public(+) and private(-);
    - The *name* is a string
    - The *parameter-list* is the list of parameters for the operations
    - The *return-type* is the type of the returned value, if there is one.
    - The *property-string* indicates property values that apply to the given operation
- The parameters in the parameter list are notated in a similar way to attributes. The form is: *direction name: type = default value*
    - The *name, type,* and *default value* are the same as for attributes.
    - The *direction* indicates whether the parameter is input(*in*), output(*out*) or both(*inout*). If no direction is shown, it's assumed to be *in*. An example operation on account might be: *+ balanceOn (date: Date) : Money*
- With conceptual models, you shouldn't use operations to specify the interface of a class. Instead use them to indicate the principal responsibility of that class, perhaps a couple of words summarizing a CRC responsibility.
- I often find it useful to distinguish between operations that change the state of the system and those that don't. UML defines a **query** as an operation that gets a value from a class without changing the system state – in other words without side effects. You can mark such an operation with the property string {query}. I refer to operations that do change state as **modifiers**, also called **commands.** Strictly, the difference between query and modifiers is whether they change the observable state. The observable state is what can be perceived from the outside. An operation that updates a cache would alter the internal state but would have no effect that is observable from the outside.
- A common convention is to try to write operations so that modifiers don't return a value; that way, you can rely on the fact that operations that return a value are queries.
- Another distinction between operation and method. An **operation** is something that is invoked on an object – the procedure declaration – where as a **method** is the body of a procedure. The two are different when you have polymorphism. If you have a supertype with three subtypes, each of which overrides the supertype's getPrice operation, you have one operation and four methods that implement it.

3.3. Notes and Comments

- Notes are comments in the diagram. Notes can be stand on their own, or they can be linked with a dashed line to the elements they are commenting.

3.4. Dependency

- A dependency exists between two elements if changes to the definition of one element(the **supplier**) may cause changes to the other (the **client**). With classes, dependency exists for various reasons: One class send message to another; one class has another as part of its data; one class mentions another as a parameter to an operation. If a class changes its interface, any message sent to that class may no longer be valid.
- As computer systems grow, you have to worry more and more about controlling dependencies. If dependencies get out of control, each change to a System has a wide ripple effect as more and more things have to change. The bigger the ripple, the harder it is to

change anything.

- The UML allows you to depict dependencies between all sorts of elements. You use dependencies whenever you want to show how changes in one element might alter other elements.
- The basic dependency is not a transitive relationship. An example of **transitive** relationship is the "larger beard" relationship. If Jim has a larger beard than Grady, and Grady has a larger beard than Ivar, we can deduce that Jim has a larger beard than Ivar. Some kind of dependencies, such as substitute, are transitive, but in most cases there is a significant difference between direct and indirect dependencies.
- Many UML relationships imply a dependency. Selected dependency keywords:

Keyword	**Meaning**
<<call>>	The source calls an operation in the target
<<create>>	The source creates an instance of the target
<<derive>>	The source is derived from the target
<<instantiate>>	The source is an instance of the target(Note that if the source is a class) the class itself is the instance of the class class; that is the target class is a metaclass).
<<permit>>	The target allows the source to access the target's private features
<<realize>>	The source is an implementation of a specification or interface defined by the the target
<<refine>>	Refinement indicates a relationship between different semantic levels; for example: the source might be a design class and the target the corresponding analysis class
<<substitute>>	The source is substitutable for the target
<<trace>>	Used to track such things as requirements

Keyword	Meaning
	to classes or how changes in one model link to change elsewhere
<<use>>	The source requires the target for its implementation.

- Your general rule should be to minimize dependencies, particularly when they cross large areas of a system. In particular, you should be wary of cycles, as they can lead to a cycle of changes.
- Be selective and Show dependencies only when they are directly relevant to the particular topic that you want to communicate. To understand and control dependencies, you are best off using them with package diagrams.

3.5. Constraint Rules

- Much of what you doing in drawing a class diagram is indicating constraints. The UML allows you to use anything to describe constraints. The only rule is that you put them inside braces ({}). You can use natural language, programming language, or the UML's formal Object Constraint Language(OCL) which is based on predicate calculus.

3.6. When to use Class diagrams?

- Class diagrams are the backbone of UML, so you will find yourself using them all the time.
- The biggest danger with class diagrams is that you can focus exclusively an structure and ignore behavior. Therefore, when drawing class diagrams to understand software, always do them in conjunction with some form of behavioral technique.

4. Sequence Diagrams

- **Interaction diagrams** describe how groups of objects collaborate in some behavior. The UML defines several forms of interaction diagram, of which the most common is the sequence diagram.
- Typically, a sequence diagram captures the behavior of single scenario. The diagram shows a number of example objects and the messages that are passed between those objects within the use case.
- Each lifeline has an activation bar that shows when the participant is active in the interaction. This corresponds to one of the participant's methods being on the stack. Naming often is useful to correlate participants on the diagram.
- A fuller syntax for participant is *name: Class*, where both the name and class are optional, but you must keep the colon(:) if you use the class.

4.1. Creating and deleting Participants

- Sequence diagram show some extra notation for creating and deleting participants. To create a participant, you draw the message arrow directly into the participant box. Deletion of a participant is indicated by big X. A message arrow going into the X indicates one participant explicitly deleting another; an X at the end of lineline shows a participant deleting itself.

- In a garbage-collected environment, you don't delete objects directly, but it's still worth using the X to indicate when an object is no longer needed and is ready to be collected. It's also appropriate for close operations, indicating that the object isn't usable any more.

4.2. Loops, Conditionals, and the Like

- A common issue with sequence diagrams is how to show looping and conditional behavior. The first thing to point out is that this isn't what sequence diagrams are good at. If you want to show control structures like this, you are better off with an activity diagram or indeed with code itself. Treat sequence diagrams as a visualization of how objects interact rather than as a way of modeling control logic.
- Both loops and conditionals use **interaction frame**, which are a way of marking of a piece of a sequence diagram.
- In general, frames consist of some region of a sequence diagram that is divided into one or more fragments. Each frame has an operator and each fragment may have a guard.
- Here are list of common operators for interaction frames.

Operator	**Meaning**
alt	Alternative multiple fragments; only the one whose condition is true will execute
opt	Optional; the fragment executes only if the supplied condition is true Equivalent to *alt* with only one trace.
par	Parallel; each fragment is run in parallel
loop	Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration
region	Critical region; the fragment can have only one thread executing it at once
neg	Negative; the fragment shows an invalid interaction
ref	Reference; reference to an interaction defined on another diagram. The frame is

Operator	Meaning
	drawn to cover the lifelines involved in the interaction. You can define parameters and return value
sd	Sequence diagram; used to surround an entire sequence diagram, if you wish.

4.3. Synchronous and Asynchronous Calls

- In UML 2, **filled arrowheads** show a synchronous message, while **stick arrowheads** show an asynchronous message.
- If a caller sends a **synchronous message**, it must wait until the message is done, such as invoking a subroutine. If a caller sends an **asynchronous message**, it can continue processing and doesn't have to wait for a response. You see asynchronous calls in multithreaded applications and in message-oriented middle-ware. Asynchrony gives better responsiveness and reduces the temporal coupling but is harder to debug.

4.4. When to use Sequence diagrams?

- You should use sequence diagrams when you want to look at the behavior of several objects within a single use case. Sequence diagrams are good at showing collaborations among the objects; they are not so good at precise definition of the behavior.
- If you want to look at the behavior of a single object across many use cases, use a **state diagram**. If you want to look at behavior across many use cases or many threads, consider an **activity diagram.**
- Other useful forms of interaction diagrams are communication diagrams, for show connection; and timing diagrams, for showing timing constraints.

4.5. CRC Cards (Class-Responsibility-Collaboration Cards)

- A **responsibility** is a short sentence that summarizes something that an object should do: an action the object performs, some knowledge the object maintains, or some important decisions the object makes. The idea is that you should be able to take any class and summarize it with a handful of responsibilities. Doing that can help you think more clearly about the design of your classes.
- The second **C** refer to **collaborators**: the other classes that this class needs to work with. This gives you some idea of the links between classes – still at a high level.
- One of the chief benefits of CRC cards is that thev encourage animated, discussion among the developers.
- A comnion mistake 1 see people make is generating long lists of low-level responsibilities. But doing so misses the point. The responsibilities should easily fit an one card.

5. Class diagrams: Advanced concepts

5.1. Keywords

- One of the challenges of graphical language is that you have to remember what the symbols mean. With too many, users find it very difficult to remember what all the symbols mean. So the UML often tries to reduce the number of symbols and use keywords instead. If you find that you need a modeling construct that isn't in the UML but is similar to something that is, use the Symbol of the existing UML construct but mark it with a keyword to show that you have something different. An example of this is the **interface**. A UML interface is a class that has only public operations, with no method bodies. Because it's a special kind of class, it's show using a class icon with the keyword *«interface»*
- Some keywords, such as *{abstract}*, show up in curly brackets . It's never really clear what should technically be in guillemets and what should be in curlies.
- Some keywords are so common that they often get abbreviated : *«interface»* often gets abbreviated to *«I»* and *{abstract}* to *{A}*

5.2. Static Operations and Attributes

- The UML refers to an operation or an attribute that applies to a class rather than to an instance **static**. This is equivalent to static members in C-based languages. Static features are <u>underlined</u> on a class diagram.

5.3. Aggregation and Composition

- One of the most frequent sources of confusion in the UML is aggregation and composition. **Aggregation** is the part-of relationship. It's like saying that a car has an engine and wheels as its parts. But the difficult thing is considering what the difference is between aggregation and association.
- The "no sharing" rule is key to composition. Composition is a good way of showing properties that own by value, properties to value objects, or properties that have a strong and somewhat exclusive ownership of particular other components.

5.4. Derived Properties

- **Derived properties** can be calculated based on other values. When we think about a date range, we can think of three properties: the start date, the end date, and the number of days in the period. These values are linked, so we can think of the length(end date - start date) as being derived from the other two values.
- Derivation in software perspectives can be interpreted in a couple of different ways. You can use derivation to indicate the difference between a calculated value and a stored value.

5.5. Interfaces and Abstract Classes

- An **abstract class** is a class that cannot be directly instantiated. Instead, you instantiate an instance of a subclass. Typically, an abstract class has one or more operations that are **abstract**. An abstract operation has no implementation; it is pure declaration so that clients can bind to the abstract class.

- The most common way to indicate an abstract class or operation in UML is to *italicize* the name. You can also make properties abstract, indicating an abstract property or accessor methods. Italics are tricky to do an a white-boards, so you can use the label: {abstract}.
- An interface is a class that has no implementation; that is, all its features are abstract. You mark an interface with the keyword «interface».
- Classes have two kinds of relationships with interfaces: **providing** and **requiring**.
- A class **provides** an interface if it is substitutable for the interface. In Java and .NET, a class can do that by implementing the interface or implementing a subtype of the interface.
- A class requires an interface if it needs an instance of that interface in order to work. Essentially, this is having a dependency an the interface.

5.6. Read-Only and Frozen

- The {readOnly} keywords mark a property that can only be read by clients and that can't be updated. Similar yet different is the {frozen} keyword from UML 1. A property is **frozen** if it can't change during the life time of an object; such properties are often called **immutable**. Although it was dropped from UML 2, {frozen} is a very useful concept, so continue to use it. As well as marking individual properties as frozen, you can apply the keyword to a class to indicate that all properties of all instances are frozen.
- If you need to represent immutability or a similar concept in latest UML diagrams, you might need to use custom stereotypes or notes to convey that information.

5.7. Reference Objects and Value Objects

- One of the common things said about objects is that they have identity. This is true, but it is not quite as simple as that. In practice, you find that identity is important for reference objects but not so important for value objects.
- **Reference objects** are such things as Customer. Here, identity is very important because you usually want only one software object to designate a customer in the real world. Any object that references a Customer object will do so through a reference, or a pointer; all objects that reference this Customer will reference the same software object. That way, changes to a Customer available to all users of the Customer. If you have two references to a Customer and with to see whether they are the same, you usually compare their identities.
- **Value Objects** are such things as Date. You often have multiple value objects representing the same object in the real world. For example, it is normal to have hundreds of objects that designate 1-Jan-04. These are all interchangeable copies. New dates are created and destroyed frequently.
- If you have two dates and wish to see whether they are the same, you don't look at their identities but rather at the values they represent. This usually means that you have to write an equality test Operator, which for dates would make a test an year, month, and day-or whatever the internal representation is. Each object that references 1-Jan-04 usually has its own dedicated object, but you can also share dates.
- Value objects should be immutable; in other words, you should not be able to take a date object of 1-Jan-04 and change the same date object to be 2-Jan-04. Instead, you should create a new 2-Jan-04 object and use that instead. The reason is that if the date were shared, you would update another object's date in an unpredictable way, a problem referred to as **aliasing.**

5.8. Qualified Associations

- A **qualified association** is the UML equivalent of a programming concept variously know as associative arrays, maps, hashes, and dictionaries.

5.9. Classification and Generalization

- Many people consider subtyping as *is a* relationship. But be aware the phrase *is a* can have different meanings.
- Generalization is transitive; classification is not. I can combine a classification followed by a generalization but not vice versa.

5.10. Multiple and Dynamic Classification

- **Classification** refers to the relationship between an object and its type. Mainstream programming languages assume that an object belongs to a single class. But there are more options to classification than that.
- In **single classification**, an object belongs to a single type, which may inherit from supertypes. In **multiple classification**, an object may be described by several types that are not necessarily connected by inheritance.
- Multiple classification is different from multiple inheritance. Multiple inheritance says that a type may have many supertypes but that a single type must be defined for each object. Multiple classification allows multiple types for an object without defining a specific type for the purpose. For example, consider a person subtyped as either man or woman, doctor or nurse, patient or not. Multiple classification allows an object to have any of these types assigned to it in any allowable condition, without the need for types to be defined for all the legal combinations.
- If you use multiple classification, you need to be sure that you make it dear which combinations are legal. UML 2 does this by placing each generalization relationship into a generalization Set. On the class diagram, you Label the generalization arrowhead with the name of the generalization Set, which in UML 1 was called the discriminator. Single classification corresponds to a single generalization set with no name.
- Generalization sets are by default disjoint: Any instance of the supertype may be an instance of only one of the sub types within that set.
- To illustrate, note the following legal combinations of subtypes in the diagram: (Female, Patient, Nurse); (Male, Physiotherapist); (Female, Patient); and (Female, Doctor, Surgeon). The combination (Patient, Doctor, Nurse) is illegal because it contains two types from the role generalization set.
- Another question is whether an object may change its class. **Dynamic classification** allows objects to change class within the sub typing structure; **static classification** does not. With static classification, a separation is made between types and states; dynamic classification combines these notions.
- Should you use multiple, dynamic classification? In the vast majority of UML diagrams, you will see only single static classification, so that should be your default.

5.11. Association class

- Association classes allow you to add attributes, operations, and other features to associations.

5.12. Template(Parameterized) class

- his concept is most obviously useful for working with collections in a strongly typed language. This way, you can define behavior for sets in general by defining a template class Set.

5.13. Enumerations

- Enumerations are used to show a fixed set of values that don't have any properties other than their symbolic values. They are shown as the class with the «enumeration» keyword.

5.14. Active class

- An active class has instances, each of which executes and controls its own thread of control . Method invocations may execute in a client's thread or in the active object's thread. A good example of this is a command processor that accepts command objects from the outside and then executes the commands within its own thread of control.

5.15. visibility

- Any class has public and private elements. Public elements can be used by any other class; private elements can be used only by the owning class. UML provides four(4) abbreviations for visibility: + (public), - (private), ~(package), and # (protected).
- When you are using visibility, use the rules of the language in which you are working. When you are looking at a UML model from elsewhere, be wary of the meanings of the visibility markers, and be aware of how Chose meanings can change from language to language.

5.16. Messages

- Standard UML does not Show any Information about message calls an class diagrams. However, I've sometimes seen conventional diagrams like, arrows to the side of association. The arrows are labeled with the message that one object sends to another.

6. Object Diagrams

- An object diagram is a snapshot of the objects in the system at a point in time. Because it shows instances rather than classes, an object diagram is often called an **instance diagram**.
- You can use an object diagram to show an example configuration of objects.
- Each name takes *instance name: class name*. Both parts of the name are optional, so, John, :Person, and aPerson are legal names. If you use only the class name, you must include the colon(:).
- Strictly, the elements of an object diagram are instance specifications rather than true instances. The reason is that it's legal to leave mandatory attributes empty or to Show instance specifications of abstract classes. You can think of an instance specification as a

partly defined instance.

6.1. *When to use Object diagrams?*

- Object diagrams are useful for showing examples of objects connected together. In many situations, you can define a structure precisely with a class diagram, but the structure is still difficult to understand. In these situations, a couple of object diagram examples can make all the difference.

7. Package Diagrams

- Classes represent the basic form of structuring an object-oriented System. Although they are wonderfully useful, you need something more to structure large Systems, which may have hundreds of classes.
- A **package** is a grouping construct that allows you to take any construct in the UML and group its elements together into higher-level units. Its most common use is to group classes.
- In a UML model, each class is a member of a single package. Packages can also be members of other packages. A package can contain both subpackages and classes.
- Each package represents a **namespace**, which means that every class must have a unique name within its owning package.
- In diagrams, packages are shown with a tabbed folder. You can simply show the package name or show the contents too. At any point, you can use fully qualified names or simply regular names.
- The UML allows classes in a package to be public or private. A public class is part of the interface of the package and can be used by classes in other packages; a private class is hidden.
- A useful technique here is to reduce the interface of the package by exporting only a small subset of the operations associated with the package's public classes. You can do this by giving all classes private visibility, so that they can be seen only by other classes in the same package, and by adding extra public classes for the public behavior. These extra classes, called *facades*, then delegate public operations to their shyler companions in the package.
- How do you choose which classes to put in which packages? Two useful principles for this are the Common Closure Principle and Common Reuse principle. The Common closure principle says that the classes in the package should need changing for similar reasons. The Common Reuse Principle says that classes in a package should all be reused together. Many of the reasons for grouping classes in a packages have to do with the dependencies between the packages.

7.1. *Packages and Dependencies*

- A **package** diagrams shows packages and their dependencies. A good package structure has a clear flow to the dependencies, a concept that is difficult to define but often easier to recognize.
- Many authors say that there there should be no cycles in the dependencies(the Acyclic Dependency Principle).
- The more dependencies coming into a package, the more stable the package's interface

needs to be, as any change in its interface will ripple into all the packages that are dependent on it (the Stable Dependencies Principle).

- Often, you'll find that the more stable packages tend to have a higher proportion of Interfaces and abstract classes (the Stable Abstractions Principle).
- The dependency relationships are not transitive.

7.2. When to use Package diagrams?

- They are useful in large-scale systems to get a picture of the dependencies between major elements of the system.

8. Deployment Diagram

- Deployment diagrams show a system's physical layout, revealing which pieces of software runs on what piece of hardware. The main items on the diagram are nodes connected communication paths. A **node** is something that can host software. Nodes come in two forms: A **device** is hardware, it may be a computer or a simpler piece of hardware connected to a system. An **execution environment** is software that itself hosts or contains other software, examples are an operating system or a container process.
- The nodes contains **artifacts**, which are the physical manifestations of software: usually, files. These files might be executables (such as .exe files, binary files, DLLs, JAR files, assemblies, or scripts), or data files, configuration files, HTML documents, and so on. Listing an artifact within a node shows that the artifact is deployed to that node in the running system.
- You can show artifacts either as class boxes or by listing the name within a node.
- Artifacts are often the implementation of a component. To show this, you can use a tagged value in the artifact box.
- Communication paths between nodes indicate how things communicate. You can label these paths with information about the communication protocols that are used.

9. Use cases

- Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users a system and the system itself, providing a narrative of how the system is used.
- A **scenario** is a sequence of steps describing an interaction between an user and a system. So, if we a Web-based online store, we might have a Buy a Product scenario.
- A use case is a set of scenarios tied together by a common user goal. In use-case speak, the users are referred to as actors. An **actor** is a role that a user plays with respect to the system. Actors might include customer, customer service rep, sales manager, and product analyst. Actors carry out use cases. A single actor may perform many use cases; conversely, a use case may have several actors performing it. Usually, you have many customers, so many people can be the customer actor. Also, one person may act as more than one actor. An actor doesn't have to be human. If the system performs a service for another computer system, that other system is an actor.
- Each use case has a primary actor, which calls on the system to deliver a service. The

primary actor is the actor with the goal the use case is trying to satisfy and is usually, but not always, the initiator of the use case. There may be other actors as well with which the system communicates while carrying out the use case. These are known as secondary actors.

- Each step in a use case is an element of the interaction between an actor and the system. Each step should be a simple Statement and should clearly show who is carrying out the step. The step should show the intent of the actor, not the mechanics of what the actor does. Consequently, you don't describe the user interface in the use case. Indeed, writing the use case usually precedes designing the user interface.
- An extension within the use case names a condition that results in different interactions from those described in the main success scenario (MSS) and states what chose differences are.
- The use case structure is a great way to brainstorm alternatives to the main success scenario. For each step, ask, how could this go differently? and in particular, what could go wrong? It's usually best to brainstorm all the extension conditions first, before you get bogged down working out the consequences.
- A complicated step in a use case can be another use case. In UML terms we say that the first use case **includes** the second. There's no standard way to show an included use case in the text, but I find that underlining, which suggests a hyperlink, works very nicely and in many tools really will be a hyperlink.
- Included use cases can be useful for a complex step that would clutter the main scenario or for steps that are repeated in several use cases. However, don't try to break down use cases into sub-use cases and subsub-use cases using functional decomposition. Such a decomposition is a good way to waste a lot of time.
- As well as the steps in the scenarios, you can add some other common information to a use case:
    - A **pre-condition** describes what the system should ensure is true before the system allows the use case to begin.
    - A **guarantee** describes what the system will ensure at the end of the use case. Success guarantees hold after a successful scenario; minimal guarantee hold after any scenario.
    - A **trigger** specifies the event that gets the use case started.
- Although the diagram is sometimes useful, it isn't mandatory. In your use case work, don't put too much effort into the diagram. Instead, concentrate an the textual content of the use cases.
- The use case diagram shows the actors, the use cases, and the relationship between them:
    - which actor carry out which use cases
    - which use cases include other use cases

9.1. Levels of Use Cases

- A common problem with use cases is that by focusing an the interaction between a user and the System, you can neglect situations in which a change to a business process may be the best way to deal with the problem.
- There are often two general terms. **System use cases** and **business use cases**. A system use case is an interaction with the **software**, where as a business use case discuss how a business respond to an event.

- Some people suggests a scheme of levels of use cases. The core use cases are at "sea level." **Sea-level** use cases typically represent a discrete interaction between a primary actor and the system. Use cases that are there only because they are included by sea-level use cases are **fish level**. Higher, **kite-level** use cases show how the sea-level use cases fit into wider business interactions. Kite-level use cases are usually business use cases, where as sea and fish levels are system use cases. You should have most of your use cases at the sea level. It's preferred to indicate the level at the top of the use case.

10. State Machine Diagram

- State machine diagrams are techniques to describe the behavior of a system. In object-oriented approaches, you draw a state machine diagram for a single class to show the lifetime behavior of a single object.
- The **transition** indicates a movement from one state to another. Each transition has a label that comes in three parts: trigger-signature [guard]/activity. All the parts are optional. The **trigger-signature** is usually a single event that triggers a potential change of state. The **guard** , if present, is a Boolean condition that must be true for the transition to be taken. The **activity** is some behavior that's executed during the transition. It may be any behavioral expression.
- A missing activity indicates that you don't do anything during the transition. A missing guard indicates that you always take the transition if the event occurs. A missing trigger-signature is rare but does occur. It indicates that you take the transition immediately, which you see mostly with activity states.
- When an event occurs in a state, you can take only one transition out of it. So if you use multiple transitions with the same event, the **guards** must be mutually exclusive.
- Remember that state machines can show only what the object directly observes or activates.

10.1. Internal Activities

- States can react to events without transition, using **internal activities:** putting the event, guard, and activity inside the state box itself.

10.2. Superstates

- Often, you will find several states share common transitions and the internal activities. In these cases, you can make them substates and move the shared behavior into a superstate.

10.3. Concurrent state

- States can broken into several orthogonal state diagrams that run concurrently.

10.4. Implementing State diagrams

- A state diagram can be implemented in three main ways: nested switch, the state pattern, and state tables. The most direct approach to handling a state diagram is a nested switch statement.
- The **State pattern [Gang of Four]** creates a hierarchy of state classes to handle behavior of the states. Each state in the diagram has one state subclass. The controller has methods for

each event, which simply forwards to the state class. The top of the hierarchy is an abstract class that implements all the event handling methods to do nothing. For each concrete state, you simply override the specific event methods for which that state has transitions.

- The **state table** approach captures the state diagram information as data. We then build either an interpreter that uses the state table at runtime or a code generator that generates classes based on the state table.
- These implementations are pretty minimal, but they should give you an idea of how to go about implementing state diagrams. In each case, implementing state models leads to very boilerplate code, so it's usually best to use some form of code generation to do it.

10.5. When to use state diagrams?

- State diagrams are good at describing the behavior of an object across several use cases. State diagrams are not very good at describing behavior that involves a number of objects collaborating. As such, it is useful to combine state diagrams with other techniques. You should remember to use the mix of techniques that works for you.

11. Activity Diagrams

- Activity diagrams are a technique to describe procedural logic, business process, and work flow. In many ways, they play a role similar to flowcharts, but the principal difference between them and flowchart notation is that they support parallel behavior.
- The activity diagram allows whoever is doing the process to choose the order in which to do things. In other words, the diagram merely states the essential sequencing rules I have to follow. This is important for business modeling because process often occur in parallel. It's also useful for concurrent algorithm, in which independent threads can do things in parallel.
- When you do have parallelism, you need to synchronize. And you can do this using **join** before closing the processes.
- The nodes on activity diagrams are called **actions** not activities. Strictly, an activity refers to a sequence of actions, so the diagram shows an activity that is made up of actions.
- Conditional behavior is delineated by decisions and merges. A **decision** has a single incoming flow and several guarded outbound flows. Each outbound flow has a guard: a Boolean condition placed inside square brackets. Each time you reach a decision, you can take only one of the outbound flows, so the guards should be mutually exclusive.
- A **merge** has multiple input flows and a single output. A merge marks the end of conditional behavior started by a decision.

11.1. Decomposing actions

- Actions can be decomposed into sub activities. Actions can be implemented either as subactivities or as methods on classes. You can show a subactivity by using the rake symbol. You can show a call on a method with syntax **class-name::method-name**

11.2. Partition

- Activity diagrams tell you what happens, but they do not tell you who does what. In programming, this means that the diagram does not convey which class is responsible for each action. In business process modeling, this does not convey which part of an organization carries out which action. This isn't necessarily a problem; often, it makes sense to concentrate an what gets done rather than an who does what parts of the behavior.

- If you want to show who does what, you can divide an activity diagram into **partitions**, which shows which actions one class or organization unit carries out.

11.3. Signals

- Actions can respond to signals. A **time signal** occurs because of the passage of time. Such signals may indicate the end of a month in financial period or each micro second in a real-time controller.
- A **signal** indicates that the activity receives an event from an outside process. This indicates that the activity constantly listens for those signals, and the diagram defines how the activity reacts.
- As well as accepting signals, we can send them. This useful when we have to send a message and then wait for a reply before we can continue.

11.4. Tokens

- If you're sufficiently brave to venture into the demonic depths of the UML specification, you'll find that the activity section of the specification talks a lot about tokens and their production and consumption. The initial node creates a token, which then passes to the next action, which executes and then passes the token to the next. At a fork, one token comes in, and the fork produces a token on each of its outward flows. Conversely, on a join, as each inbound token arrives, nothing happens until all the tokens appear at the join; then a token is produced on the outward flow.

11.5. Flows and Edges

- UML 2 uses the term **flow** and **edge** synonymously to describe the connections between the two actions. The simplest kind of edge is the simple arrow between two actions. You can give a name to an edge if you like, but most of the time, a simple arrow will suffice.

11.6. Pins and Transformations

- Actions can have parameters, just as methods do. You don't need to show information about parameters on activity diagram, but if you wish you can show them with **pins**. If you're decomposing an action, pins correspond to the parameter boxes on the decomposed diagram.
- When you are drawing an activity diagram strictly, you have to ensure that the output parameters of an outbound action match the input parameter of another. If they don't match, you can indicate a **transformation** to get from one to another. The transformation must be an expression that's free of side effects: essentially, a query an the output pin quary that supplies an object of the right type for the input pin.
- You don't have to show pins an an activity diagram. Pins are best when you want to look at the data needed and produced by the various actions. In business process modeling, you can use pins to show the resources produced and consumed by actions.
- If you use pins, it's safe to show multiple flows coming into the same action. The pin notation reinforces the implicit join.

11.7. Expansion regions

- With activity diagrams, you often run into situations in which one action's output triggers multiple invocations of another action. There are several ways to show this, but the best way is to use an expansion region. An **expansion region** marks an activity diagram area where actions occur once for each item in a collection.

11.8. Flow Final

- Once you get multiple tokens, as in an expansion region, you often get flows that stop even when the activity as a whole doesn't end. A **flow final** indicates the end of one particular flow, without terminating the whole activity. This approach allows expansion regions to act as filters, whereby the output collection is smaller than the input collection.

11.9. Join Specifications

- By default, a join lets execution pass on its outward flow when all its input flows have arrived at the join.(Or in a more formal speak, it emits a token on its output flow when a token has arrived on each input flow). In some cases, particularly when you have a flow with multiple tokens, it's useful to have a more involved rule.
- A **join specification** is a Boolean expression attached to a join. Each time a token arrives at the join, the join specification is evaluated and if true, an output token is emitted.

11.10. When to use Activity diagram?

- The great strength of activity diagrams lies in the fact that they support and encourage parallel behavior. This makes them a great tool for work flow and process modeling.
- The main strength of doing this may come with people using UML as a programming language. In this case, activity diagrams represent an important technique to represent behavioral logic.

12. Communication Diagrams

- Communication diagrams, a kind of interaction diagram, emphasize the data links between the various participants in the interaction. Instead of drawing each participant as a lifeline and showing the sequence of messages by vertical direction as the sequence diagrams does, the communication diagram allows free placement of participants, allows you to draw links to show how the participants connect, and use numbering to show the sequence of messages.
- With communication diagram, we can show how the participants are linked together.
- We can symbolize them using ball-and-socket notation.

12.1. When to use communication diagrams?

- A more rational approach says that sequence diagrams are better when you want to emphasize the sequence of calls and that communication diagram are better when you want to emphasize the links.

13. Composite Structures

- One of the most significant new features in UML 2 is the ability to hierarchically decompose a class into an internal structure. This allows you to take a complex object and break it down into parts.

13.1. When to use composite structures?

- A good way of thinking about the difference between packages and composite structures is that packages are compile-time grouping, while composite structure show runtime grouping. As such, there are natural fit for showing components and how they are broken into parts; hence, much of this notation is used in component diagrams.

14. Component Diagrams

15. Timing Diagrams

- Timing diagrams are another form of interaction diagrams, where the focus is on timing constraints: either for a single object or, more usefully a bunch of objects. Timing diagrams are useful for showing timing constraints between state changes an different objects.