220

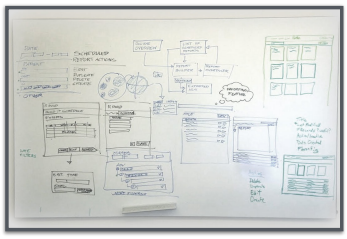
Software Engineering Design

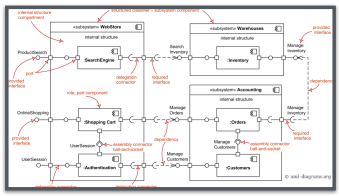
Dr Robert Chatley - rbc@doc.ic.ac.uk

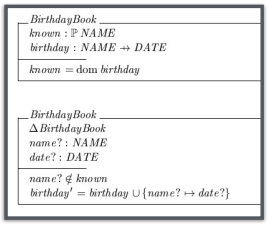
@rchatley #doc220

Welcome to the course on Software Engineering Design (CO220).

The course webpage is: http://www.doc.ic.ac.uk/~rbc/220
We will use Piazza for questions and discussion:
http://piazza.com/imperial.ac.uk/fall2015/220
Course materials, tutorial exercises etc will be on CATe.

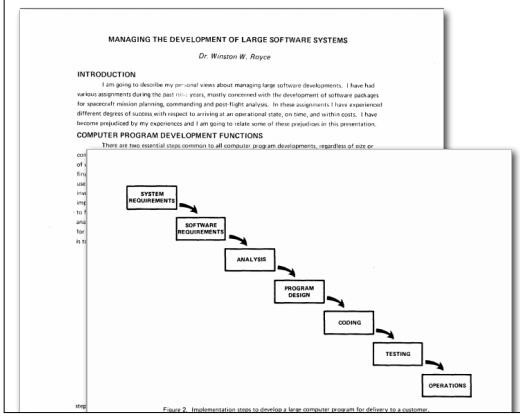






Traditionally, software design courses have often concentrated on notations and formal methods for drawing out and specifying how a piece of software should be structured, and how it should behave. This course aims to take a more modern approach…

Waterfall Development


MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS

Dr. Winston W. Royce

INTRODUCTION

COMPUTER PROGRAM DEVELOPMENT FUNCTIONS

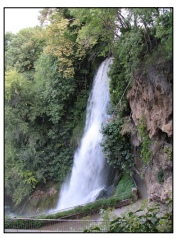


photo: stefg74

#doc220

Winston W. Royce wrote a paper in 1970 called "Managing the Development of Large Software Systems". Unfortunately this paper was somewhat misinterpreted by the industry at large, and what emerged was the Waterfall Model of software development. It set out a number of different phases of software design and implementation, which in the Waterfall model flow from one to the other sequentially. Design was done as an early phase, before coding and testing.

One of the problems with this is that we only get one shot at getting things right - there isn't much scope for going back and reworking things after a phase ends. As it happens, this wasn't actually what Royce meant when he wrote the original paper, but it was what people took from it and it stuck for quite a few years.

photo: Rachel Davies @ UnrulyMedia

#doc220

The waterfall model of upfront design is not very representative of how most software development teams working today do design. Teams tend to work using agile, iterative methods, and to revise and refine the design of their software constantly as they add new features, fix bugs, etc. They still talk about, think about, and draw software designs, but this is not so often a formal stage of the development process. Design is a matter of organising and re-organising code in order to solve problems, and avoid future problems. The focus of this course is on how to do this well - it is much more focussed on code than on notation.

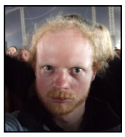
Lab / Coursework

- Weekly lab/tutorial (202/206)
- No separate coursework
- http://piazza.com/imperial.ac.uk/fall2015/220


Patrick


Oana


Akara


Tim

#doc220

photo: Rachel Davies @ UnrulyMedia

Exercises to be done in pairs

http://www.extremeprogramming.org/rules/pair.html
http://collaboration.csc.ncsu.edu/laurie/Papers/PP%20in%20Introductory_CSED.pdf

3 Languages

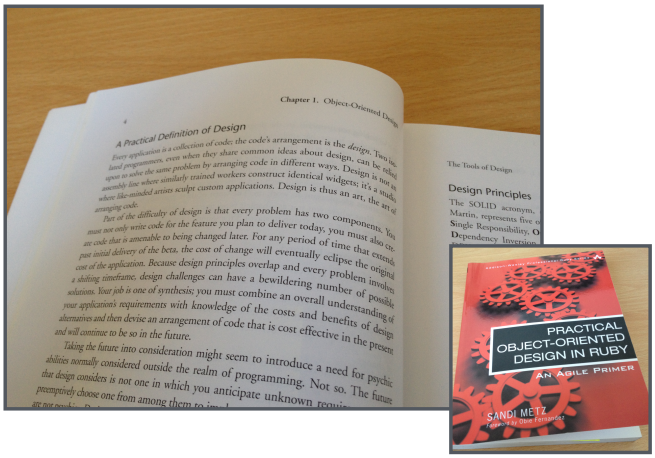
Ruby

Typing	static	dynamic	dynamic
OO model	class-based	class-based	object-based
Inheritance	single	single + mixins	prototype-based
Execution	compiled	interpreted	interpreted

#doc220

In the course we will use code examples in three different languages, which have various different properties.

The aim is not to learn these languages thoroughly (although you might like to take that opportunity), and you won't be assessed on how well you know them. The idea is to show that there are elements of design that are common regardless of the language we choose to implement our software.

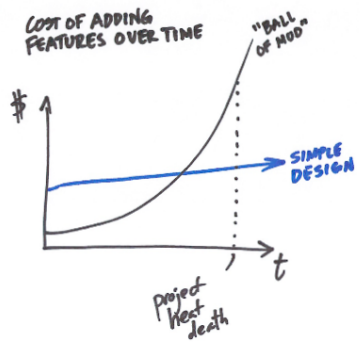


#doc220

What do we mean by design anyway? In her book on design, Sandi Metz writes "Every application is a collection of code; the code's arrangement is the design."

She goes on, "Part of the difficulty of design is that every problem has two components. You must not only write code for the feature you plan to deliver today, you must also create code that is amenable to being changed later."

Cost of Change



http://www.jbrains.ca/permalink/the-three-values-of-software

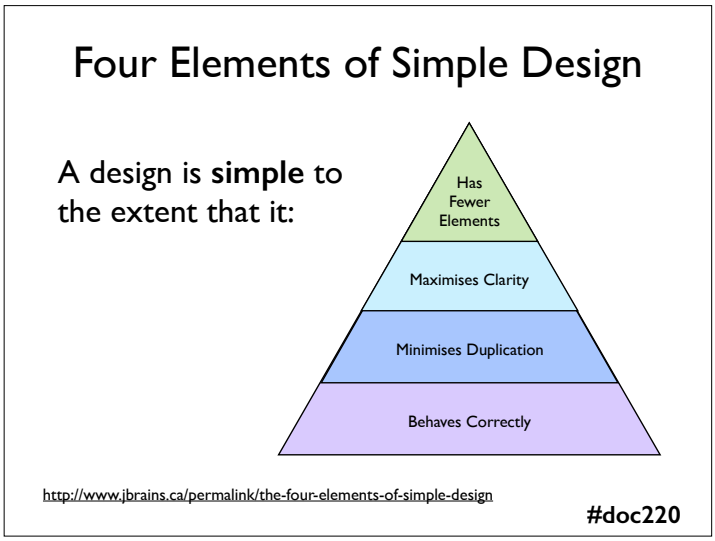
#doc220

In the article linked on the slide, J.B. Rainsberger refers to design as being one of the three values of software. Without design, the marginal cost of adding a new feature will gradually increase until it becomes too hard to add anything new. Paying attention to improving the design over time will allow us to evolve and maintain the software effectively.

This is one of the most important aspects of software design. As Sandi Metz writes in her book "Practical Object Oriented Design in Ruby": "Changing requirements are the programming equivalent of friction and gravity. They introduce forces that apply sudden and unexpected pressures that work against the best-laid plans. It is the need for change that makes design matter".

In order to produce better designs, we can learn and follow a set of design principles for software development. One example of this is the DRY principle, described by Hunt and Thomas in their book *The Pragmatic Programmer*. This principle is about making it the case that every piece of knowledge, or every decision, should be located in only one place within a system. If we do not do this, then when it comes to making a change of behaviour, we should only need to change the system in a minimal number of places. The more places we need to change, the longer the change will take, and the more risk there is that we will break something.

In this course we will try to cover a set of useful design principles to help us improve the design of our software.



In another article, J.B. Rainsberger sets out his four elements of simple design as follows. A design is simple to the extent that it 1) passes its tests 2) minimises duplication 3) maximises clarity 4) has fewer elements.

Minimising duplication helps to make the design more maintainable and hence more robust. If code is duplicated, then making a change to it may mean making the same change in multiple places, which is more work than we would like. By aiming to maximise clarity, we work to improve comprehension of the design by humans. Rainsberger's fourth principle - to reduce the number of elements - aims to make the overall size of the system smaller, making it simpler and easier to comprehend.

The way a piece of software is divided into components, and the way that those various components fit together and interact is known as the software's architecture. Some structures are commonly used and so there are some patterns that are familiar to experienced engineers. Later in the course we will look at some of these in detail.

> Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently [...] Habitability makes a place livable, like home. And this is what we want in software - that developers feel at home, can place their hands on any item without having to think deeply about where it is.
>
> *--Richard Gabriel, Patterns of Software*

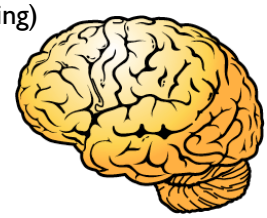
#doc220

Software architecture is not only about structure; it has many concerns. This quote from Dick Gabriel suggests that an important aspect of software design is making the code habitable for the people who maintain it. It should follow familiar patterns and idioms and be easy to find your way around in and change.

Sandi Metz makes a similar point, saying "Applications that are easy to change are a pleasure to write and joy to extend. They're flexible and adaptable. Applications that resist change are just the opposite; every change is expensive and makes the next cost more".

Designing for the Human Brain

We can only hold **4±1** items in our **short-term** (working) memory



The brain tries to cope with this by **chunking** information together

Thomas Mullen: Cognitive Psychology and the Fundamentals of Good Software Design

#doc220

When we learn new things, we want most likely want to commit them to our long-term memory. In order to get into long-term memory, information has to be held and repeated in the short-term memory. The short-term memory has been shown to only be capable of holding 4±1 things at the same time. To try to cope with this, the brain tries to group information together if it can find a higher-level concept or analogy to group things under.

Design Patterns

Original "Gang of Four" (GoF) book published 1994 - still very relevant

Presents 23 patterns, common object structures that solve particular problems

Examples in C++ and Smalltalk

#doc220

Following on from Alexander's work, the book Design Patterns, commonly known as the Gang of Four book, contains a number of patterns that are possible solutions to frequently occurring problems in object-oriented design. The Gang of Four book mostly has examples in C++ and in SmallTalk, but the patterns apply to any object-oriented design.

A later book, Russ Olsen's *Design Patterns in Ruby*, presents a lot of the same patterns, but shows how they may be applied in Ruby instead. There are also other books available that illustrate patterns in other languages like Java, C#, etc.

The key point here is that patterns are not dependent on specific languages and may be helpful in solving many OO design problems. They talk about the interactions between collaborating objects, and the structures that they form. They do not dictate the implementation.

Pattern Descriptions

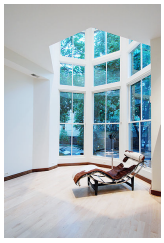
Name: handle for design problem/solution
Problem: intent, motivation, applicability
Solution: elements, relationships, collaborations
Consequences: results and trade-offs

GoF has a very detailed pattern description format with many sections

#doc220

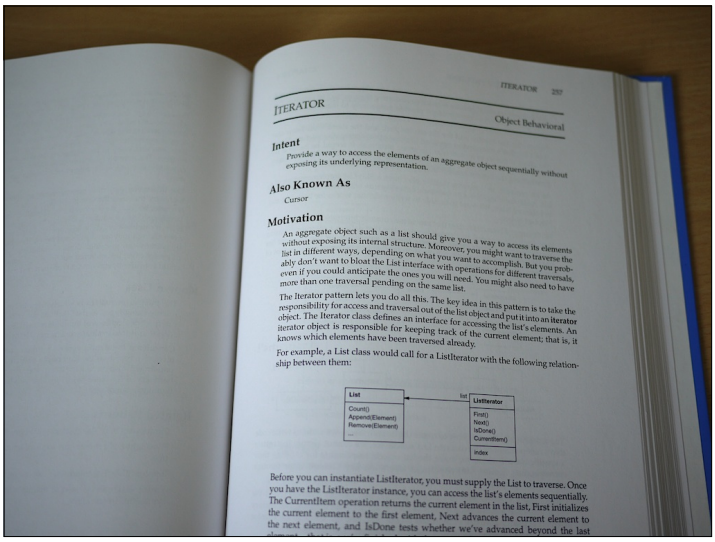
There are now a lot of patterns books, but all follow a similar form for describing patterns. They give the pattern a name, so that people can refer to it and talk about it with other engineers, and show a problem that the particular pattern might solve, an example of how the solution might be implemented, and any consequences of using this pattern.

An Example from "Real" Architecture



#doc220

The idea of architectural patterns stems back to "real" architecture, and the construction of real buildings. The above pictures show a "bay window", all three follow the pattern, but the way they are built and fit in to the rest of the building in each case is different. Christopher Alexander built a collection of architectural design patterns in his writings.



ITERATOR 257

ITERATOR — Object Behavioral

Intent
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also Known As
Cursor

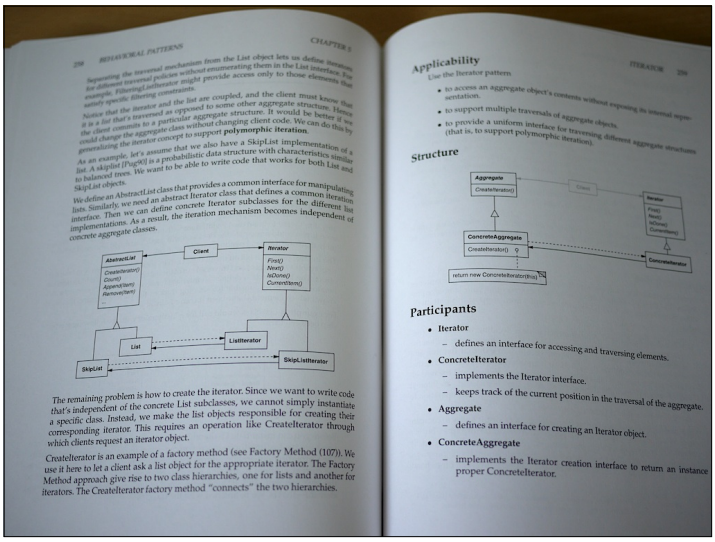
Motivation
An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an *iterator* object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

For example, a List class would call for a ListIterator with the following relationship between them:

Before you can instantiate ListIterator, you must supply the List to traverse. Once you have the ListIterator instance, you can access the list's elements sequentially. The CurrentItem operation returns the current element in the list. First initializes the current element to the first element. Next advances the current element to the next element, and IsDone tests whether we've advanced beyond the last

Here we see an extract from the Gang of Four book, showing how one of the most commonly used patterns - the Iterator - is set out in the pattern format.



Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

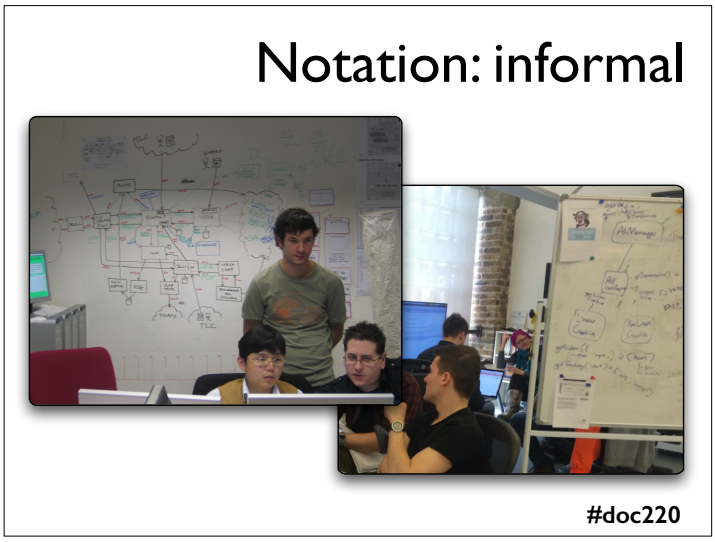
Structure

Participants

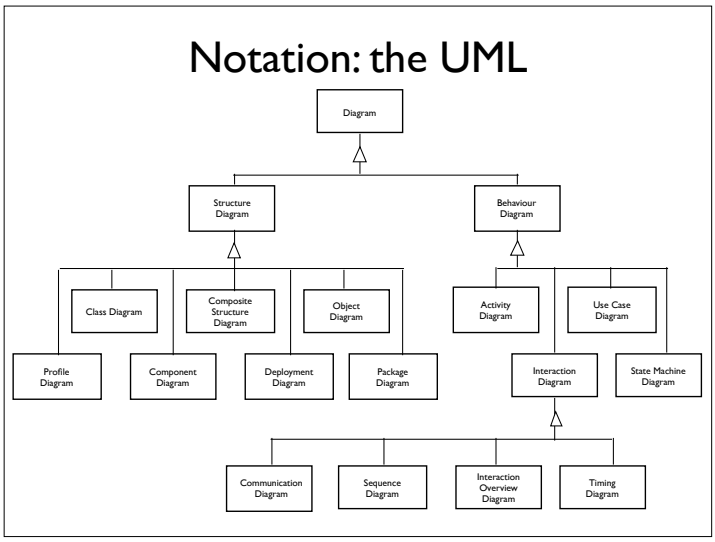
- Iterator
  - defines an interface for accessing and traversing elements.
- ConcreteIterator
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- Aggregate
  - defines an interface for creating an Iterator object.
- ConcreteAggregate
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

We see an example implementation, and the overall structure of the pattern, identifying the participants and how they interact. We often call objects that work together like this “collaborators”.

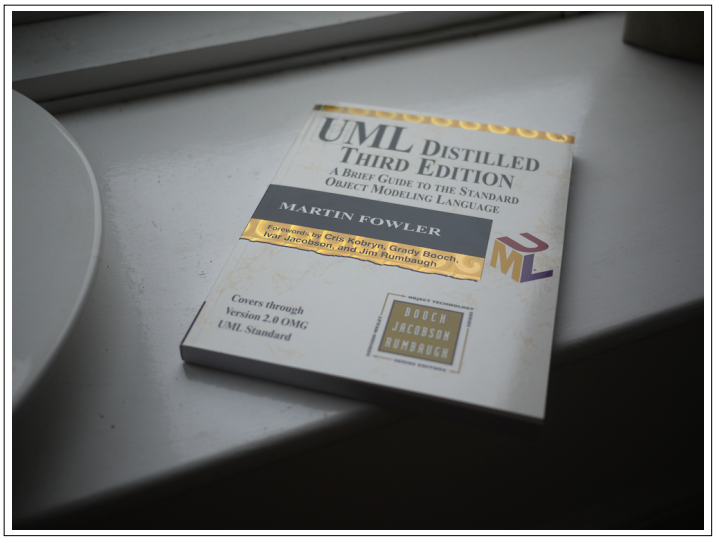
Knowing a core set of patterns allows you to recognise them in existing code, and to follow them when extending it. Importantly, it also gives you a vocabulary of design terminology that allows you to discuss software design with other engineers.



We often need some sort of notation to describe a design. Quite often the best notation is informal. Diagrams are frequently drawn as collections of boxes and lines, hastily sketched in a notebook or drawn up on a whiteboard. Drawing can help with thinking, and a diagram can act as a useful point of focus during a design discussion. It does not need to follow a strict notation to be helpful, but following a few common idioms can make things easier to understand.

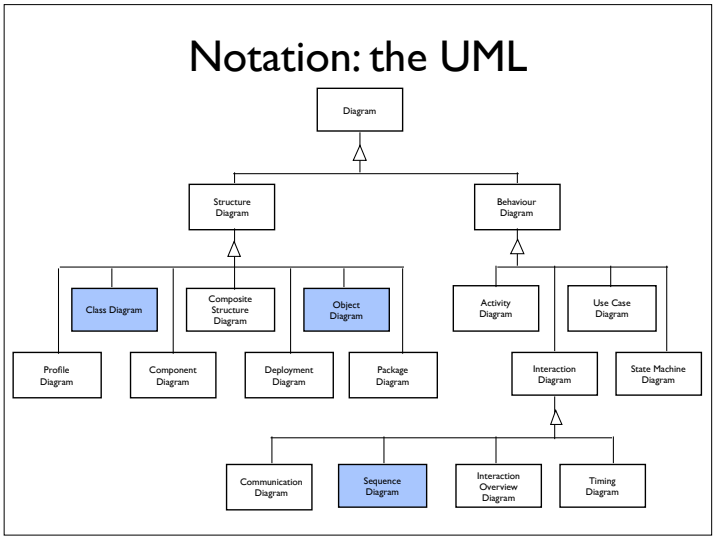


A notation which is widely used, though often interpreted informally, is the Unified Modelling Language (the UML). The UML is a complicated thing which can be used to describe many different types of situation, using many different types of diagrams. It is a formal notation, and there is a “correct” way of doing it, but some of the most useful UML diagrams are sketched quickly on the back of an envelope during a coffee break.

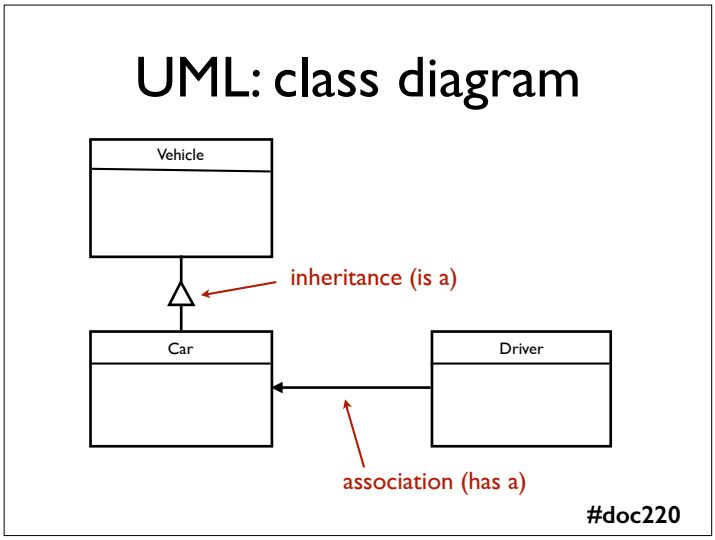


Martin Fowler's UML Distilled is generally accepted to be the best book on the UML if you need a reference. It is quite a slim volume, but covers the main types of diagrams in detail.

Fowler describes three ways of using the UML: as a sketch, as a blueprint, or as a programming language. We mostly only concern ourselves with sketching, using the UML to communicate software designs among humans.

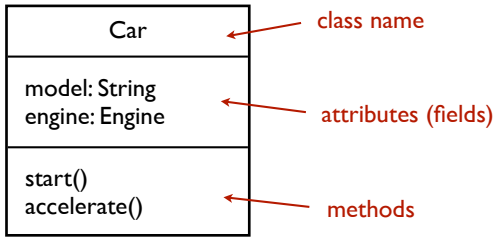


We will look at a small subset of the UML, looking at three different types of diagrams, which are probably the most commonly used, and the most useful to understand: class diagrams, object diagrams, and sequence diagrams.



The UML class diagram shows the relationships between different classes in an object-oriented system. It uses the triangle to indicate inheritance, and lines between peers to indicate associations, which can be directional - the dependency indicated by an arrowhead.

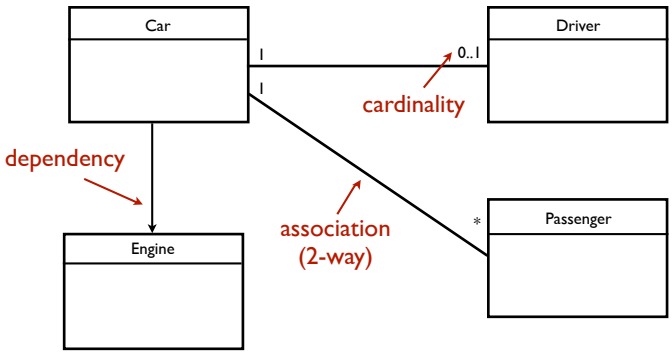
UML: class diagram



#doc220

Each box in the class diagram represents a class. We can divide this box into sections showing at the top the class name, in the middle any attributes or fields, and in the lower section, any methods. You can divide methods and fields into public and private by marking them with + and - if you wish.

UML: relationships

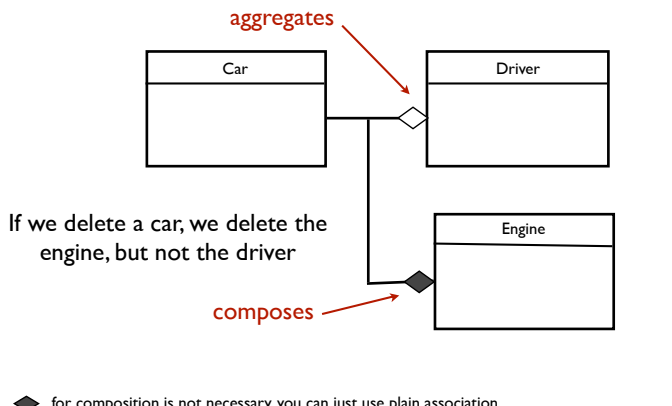


#doc220

The lines between classes show dependencies, what refers to what. An arrowhead indicates a unidirectional dependency: A uses B, but B knows nothing of A. A line with no arrowheads indicates a two-way association: A refers to B, and B refers to A. We may also annotate the associations with cardinality constraints, so a car may have many passengers, but a passenger may (at any one time) only ride in one car.

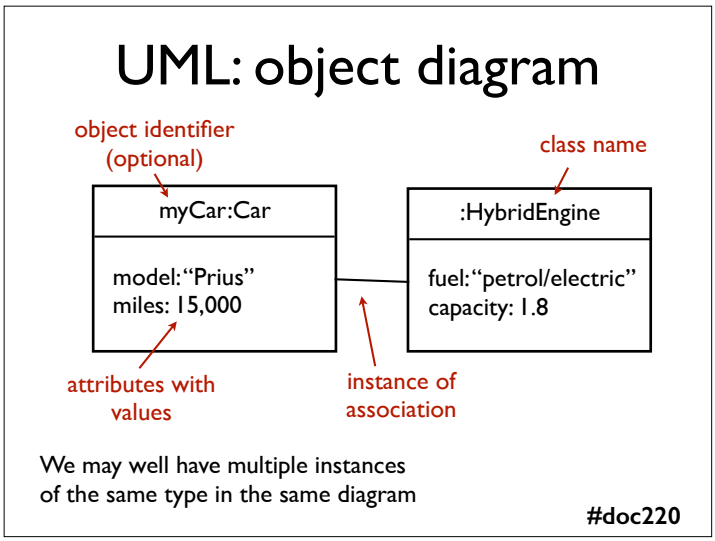
---

composite vs aggregate

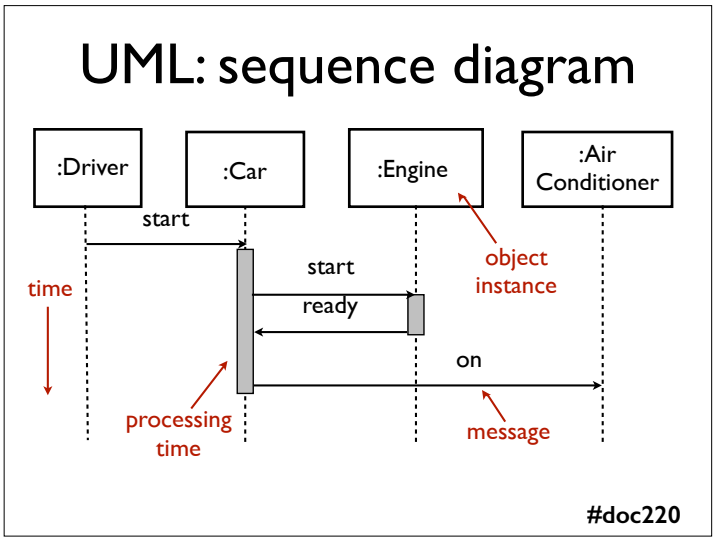


#doc220

Two similar, but different concepts: A composition is something that is made up of, and wholly includes, its constituent parts. An aggregate is a grouping of independent entities. A university course aggregates a number of students, but if we remove that course from the curriculum the students still exist and may take part in other classes. However, a course might compose some lectures and coursework, which would no longer be needed if the course no longer existed.



An alternative, or supporting, type of UML diagram is the object or instance diagram. Rather than showing relationships between classes, this shows the relationships between, and the values held by, particular instances of classes. An object diagram may show how several instances of the same type interact, and give examples of particular values in their fields.



Both class and object diagrams show a snapshot at a particular moment in time. The relationships between objects (if not classes) may vary often during the execution of a program, but these diagrams do not capture any of this dynamism. An alternative style of diagram is the sequence diagram, which again shows object instances, but also shows the interactions between them, but showing the messages that they send and receive between one another over time.
