**Type Systems:**

One of the most important distinguishing features of a programming language is the kind of type system it offers.

Type systems categorise every expression in a program as to what kind of “thing” it is.

E.g., is the variable x a number or a string? Once we know that, we can forbid the use of numbers where strings are expected and so on.

Some languages, such as Java, require us to specify the types in our programs, at least when declarations are made.

Some are smart enough to work out what the type must be using type inference, such as Haskell.

Most modern type systems contain much richer categories than just numbers, strings, booleans and so on. We’ll be looking in some detail at the type systems of Java and Haskell.

Type systems can be categorised as follows:

* dynamic types (e.g. PHP, Python, Ruby) type-checking performed at runtime. If you try to add two Booleans together, the mistake won’t be spotted until it’s too late.
* static types (e.g. Java, Haskell) type-checking performed at compile-time (most of it, anyway).

The debate about which is best will never end, and it really depends on the context. Dynamically typed languages have benefits for “exploratory programming”.

But static types have proven their value in large systems. Bob makes a change to an interface and the compiler will point out the resulting problems to Alice, who works on the other side of the world and has never met Bob but needs to use this interface...

A language is weakly-typed if it allows coercion (e.g. Java and its use of casting).

Just as static types catch “category errors” (trying to use an int where a float is expected, for example), languages with more expressive type systems can relegate more programming errors to the same bin.

They also give programmers more tools to express themselves, a wider variety of abstractions at their disposal.

Advanced type systems in use right now: Haskell, Scala, OCaml...

The avant garde: Agda, Idris, Coq.

**Syntax vs Semantic:**

Virtually every language consists of:

* a particular syntax,
* some behaviour associated with every instruction,
* some useful libraries, and
* a particular style or idiom that programmers of this language use.

If you want to learn to use a language, you need to care about all of these.

In this module, we don’t care that much about syntax, as important as that is to the language’s users.

We don’t care much about the ecosystem (libraries, build tools etc). That’s mostly a social issue.

We are interested to some extent in the idioms available (paradigms – do you use classes or functions?), but most of all we care about **semantics**. This tells us what a program will **do** when it runs.

**semantics:**

There are a number of ways that we can define the semantics of our languages, so we know exactly what a [25] will do.

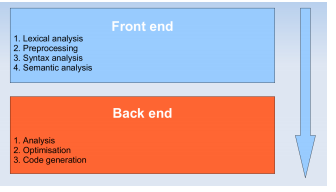
The most widely used of these are **operational, denotational** and **axiomatic** **semantics**. While these are definitely worth knowing about, they can be very formal.

We will normally talk informally about semantics although we will use the **Lambda Calculus** (λC) as a notation for semantics.

See the required reading to find out more about different notations for semantics.

**Syntax:**

We need notations to describe the legal expressions in a language if we want to implement it. The phases of a compiler:



**Lexical analysis:**

Lexical analysis means taking a source file and converting its contents into a **stream of tokens**.

A token is a sequence of characters that can be treated as a unit in the syntax of that language.

|  |  |
| --- | --- |
| **Type:** | **Example:** |
| ID | x |
| NUM | 256 |
| REAL | 3.14 |
| LPAREN | ( |

Depending on the language, whitespace and comments may be ignored.

**Defining Syntax:**

Any general-purpose language can be used to implement a lexer, but there are standard tools that save effort and produce fast, reliable, easily maintainable lexers:

* **Regular expressions** and **Backus-Naur Form** to define the lexer, and
* **Deterministic finite automata** to implement it.

**Regular expressions:**

A **language** is a set of strings. A string is a finite sequence of symbols. The symbols are taken from an **alphabet**.

E.g. the set of all legal Java programs is a language (and is infinite).

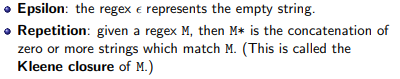
The symbols used in the language are drawn from the alphabet of UNICODE characters.

To specify a language, which may be infinite, we can use **regular expressions** (regexes).

Regular expressions consist of:

* **Symbols**: each symbol, a, matches the language containing just the string a.
* **Alternation**: given two regexes M and N, we can make a new regex M | N.
* **Concatenation**: given two regexes M and N, we can make a new regex M • N (M followed by N). The • is usually left out, so we just write MN.

Regular expressions consist of:



Regexes are available in every modern programming language. Each implementation adds some syntactic sugar.

For example, the ? (zero-or-one) operator: a? is equivalent to a | .

The most common regex syntax is PCRE – Perl Compatible Regular Expressions – which is available in lots of languages other than Perl.

**Context Free Grammars:**

Context-free grammars are an elegant way of describing the structure of a language. They are particularly useful because they do so declaratively.

A grammar has a set of productions of the form

symbol → symbol symbol . . . symbol

where there are zero or more symbols on the right-hand side.

Each symbol is either terminal, meaning that it is a token from the alphabet of the language, or non-terminal, meaning that it appears on the left-hand side of a production (i.e. we have to expand that symbol recursively).

No token can appear on the left-hand side of a production.

One of the non-terminals is designated as the start production.

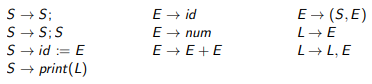
Straight-line programs include assignment of variables, numbers, addition and printing. We can also bracket together a statement, s, and expression, e, where s is executed for its side-effects and the return of the whole expression is the value of e.

1 a := 7;

2 b := c + ( d := 5 + 6 , d ) ;

3 print ( a ) ;

A CFG for straight-line programs:



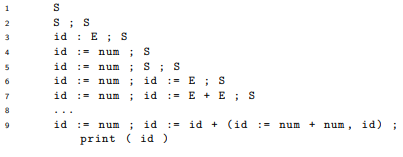
The start symbol is S. The terminals are id print num , + ( ) := ;

S is for statement (something that can have side-effects). E is for expression. L is for list.

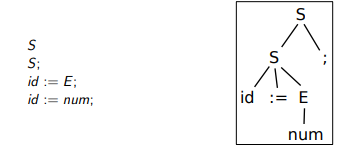
To see if our earlier program really a straight-line program is, we need to check that it is in the language of the grammar. To do this we perform a derivation.

Beginning with the start symbol, S, repeatedly replace any non-terminal symbol with one of its right-hand sides. We can choose whether to expand the leftmost or rightmost symbols, giving leftmost and rightmost derivations.

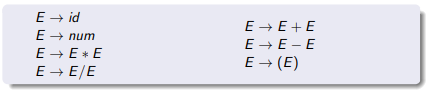
Beginning of a leftmost-derivation for our straight-line program:



A parse tree is made by connecting each symbol in a derivation to the symbol from which it was derived. For example, deriving x := 5;

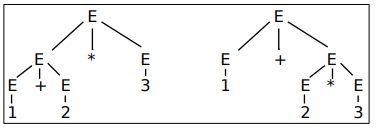


A grammar is ambiguous if a sentence in the language can have two derivations, or parse trees. Consider a language of arithmetic expressions:

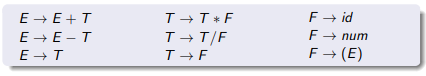


and the sentence 1 + 2 \* 3

Two possible derivations of 1 + 2 \* 3:



We can usually eliminate ambiguity by adding new non-terminals to rewrite the grammar. In this case, we can subdivide expressions into terms (things which can be added and subtracted) and factors (things which are multiplied and divided).



Backus-Naur Form (BNF) is very similar to CFG, but more convenient and thus likely to be seen “in the wild”. The official documentation for many programming languages may include a BNF specification.

A BNF specification is written as:

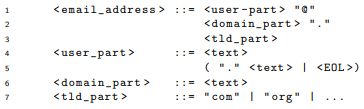


where is a non-terminal and expression may contain recursive references to other symbols (in which case it is non-terminal) or may be a terminal symbol.

A simple example:



A single derivation rule can contain several optional parts, separated by the character |.



**Extended BNF** (EBNF) allows the use of the regex operators for repetition, \* and +

**λ-calculus:**

So, in this model for functional application,

* Everything is an anonymous (higher-order, first class) function,
* We can create new functions, and supply arguments to functions to evaluate them, and that’s it.

Each function takes a single argument.

We are talking about the **untyped** λ-calculus, its simplest form.

λC + types + equality forms the basis of languages such as ML.

Note that there is no equality in the untyped λC. We can’t say in general whether two lambda-terms are the same.

**Untyped λ-calculus:**

Grammar for λ terms, e.



**Scope:**

Abstractions contain variables which are free or bound. x is bound in M iff it is in the scope of λx in M. Otherwise, it is free in M.

* x is bound: (λx.(λy.M x))
* x is free: (λy.M x)

**Application:**

Application reduces or simplifies an expression and is carried out by substitution.

Applying (λx.M)N means replacing x with N in M (and stripping off the binding).

We write this M[N/x]. Doing so might mean making changes to bound variables by renaming them.

**α-conversion:**

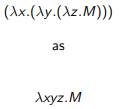
In (λx.(λy.M x))y we need to rename the inner y during application to something “fresh”.

Otherwise we would end up with (λy.M y), which isn’t what we wanted.

Renaming variables is called α-conversion.

**Notation:**

We can get rid of some λs and parentheses by writing:



But don’t forget this is not a single function taking three arguments, functions only ever take one.

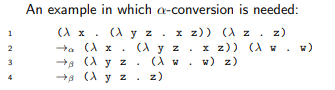
**β and η reduction, normal forms:**

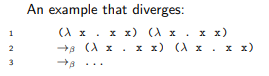
There are two ways to evaluate or reduce λ terms:

1. function application, called β-reduction, and
2. throwing away redundant terms, called η-reduction. I.e. in the term (λx.M x), if x is not in the free variables of M, then we can reduce the term to M.

**β-reduction examples:**





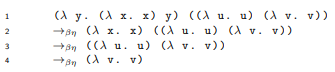


**Evaluation strategies:**

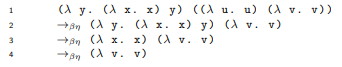
In call-by-name reduction, arguments are supplied to functions as they are (i.e. without being evaluated). This is also called being non-strict or lazy.

In call-by-value reduction, arguments to functions are evaluated before being supplied. This is also called a strict or eager evaluation strategy.

call by name:



call by value:



**Exploring the λ-calculus:**

There will be a lab session in a coming week that asks you to experiment with using λC to define a programming language. You will be creating something called a **Church encoding**.

You will see that Church encodings are verbose but simple, and thus easy to reason about precisely what the program will do when it runs.

**Programming Paradigms:**

A programming paradigm is a conceptual approach to problem-solving.

Each programming language can be said to fall into one or another paradigm or, as is often the case, into several paradigms at the same time.

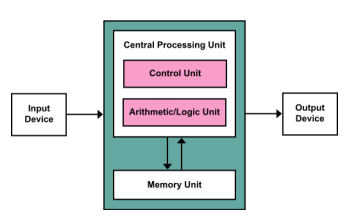
That is, each language embodies some approaches to problem solving that are available to the user through the conceptual and practical tools the language provides.

Each paradigm deploys its own jargon and most have passionate adherents.

Most widely used languages lend themselves to several paradigms. Paradigms are determined as much by what they forbid as what they encourage.

The earliest programming paradigm was procedural programming, which arises directly from the von Neumann architecture.

**von Neumann Architecture:**



The von Neumann architecture is a model for the FETCH/EXECUTE cycle, Input and Output, and provides for storing/retrieving data in registers.

Early high-level languages such as FORTRAN and COBOL reflected this architecture in that they focus on presenting instructions to the CPU one at a time.

The same is true of modern languages such as Java, C, Javascript etc.

Procedural languages have the procedure as the basic compilation unit, or way of organising the code. Any procedure can call any other, or itself.

Each high-level procedural language adds some specialised language or concepts relating to the problem domain – in the case of FORTRAN, mathematics, or for COBOL, business.

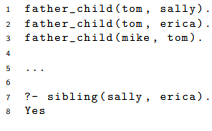
**Models of computing:**

It may seem odd to think about the concept of “executing instructions one at a time” being part of a particular paradigm. After all, that’s how CPUs work, right? They have to fetch the next instruction and execute it, so surely all languages will reflect that.

**declarative programming:**

But there is an alternative model, called declarative programming. In this paradigm, we use a PL to describe the problem, in expectation that the language implementation will solve it for us. In declarative style, the computer can solve the problem any way it likes. Declarative style is, for this reason, higher level than procedural style.

Prolog is an archetypal declarative language. It is specialised for the problem domain of logic. A Prolog program consists of a number of logical assertions, which builds up to a model of the world. We can then query this model.



**Lambda: The Ultimate Declarative:**

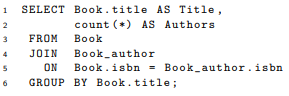
Think back to the lecture on the λ Calculus. A lambda expression doesn’t insist that it is evaluated lazily or eagerly, or with a particular evaluation strategy (e.g rightmost or leftmost derivation).



The lambda expression denotes the problem without saying how it should be solved.

**SQL:**

The most widely used declarative language is probably Structured Query Language (SQL). Using an SQL SELECT query, we specify a result set that we want to receive, and how the RDBMS does that is its own business.



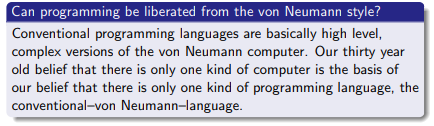
**Imperative paradigm:**

Probably the most widespread and influential paradigm is the imperative paradigm. This is an extension of the procedural paradigm, in that instructions are sequenced one-by-one but adds an emphasis on mutating state. That is, storing and retrieving values in memory, changing the values and storing them again, and so on.

C is the archetypal imperative language and is a low-level example – it allows you to manipulate memory directly with few restraints. Imperative languages can be relatively high level too though, such as Java. It’s probably true to say that most of the code ever written uses the imperative style.

**Limits of Imperative paradigm:**

When accepting the Turing Award in 1977, John Backus (of BNF fame) pointed out some of the limitations of imperative style. The reliance on mutating state, especially if that includes global variables, means that large programs become inherently complex.



Backus was pointing out that increasingly bloated and complex languages (Hi, C++!) lead to programs that are increasingly hard to get right and to maintain. His proposal was for smaller languages that avoid side effects and that lend themselves to algebraic reasoning. That is, languages that let us write programs that can be treated like mathematics, that let us work out what they do before they run, that let us write compilers that can reliably transform the program into a different one with the same semantics.

**Functional Paradigm:**

Backus was proposing that we use the functional paradigm. This is an approach to problem-solving that emphasises the use of functions and the avoidance of side-effects:

* The basic unit of code is the function, and always has a return value.
* Referential transparency. No side effects and functions always give the same value for the same input.
* No destructive assignment, no shared state.
* Functions can be passed as arguments and returned as values – higher order.

**Functional Programming:**

Functional languages tend to be:

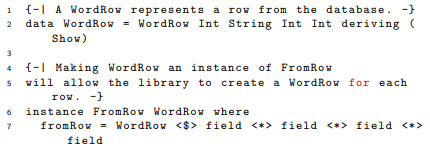
* Very concise with minimal syntax.
* Focused on recursion rather than iteration.
* As we’ve already said, functional languages are declarative. More amenable than imperative languages to algebraic reasoning.

A function with no side effects has the property of referential transparency. This means that wherever we see two calls using the same arguments to a pure function, foo, the result is guaranteed to be the same. We know that the result value of foo doesn’t depend on the time of day, the state of the file system, or anything else. Thus, we model the problem as a series of stateless computations. This isolation of side effects makes IO related bugs easier to find and pure functions easier to test and reason about.

Another feature of many functional languages is that they tend to have expressive type systems. We can express more of the constraints specific to our problem using types, and so the compiler can check more of the program for correctness. This correctness could be something simple, like noting that our code is trying to add together incompatible units, like inches and centimetres (this doesn’t require particularly expressive types in itself). Or it could be something more complex, like type-safe database access.

**Expressive types:**

Here is a fragment of Haskell code that is used to select data from a database in a type-safe way – that is, we can’t accidentally treat a DB column with the SQL type VARCHAR (string) as if it were a number.



These type systems enable parametric polymorphism, also known as generics. This means writing code which is very general and will work for any number of types.



Forty years after giving his talk, Backus seems to have been onto something! Functional languages like Haskell, Java, Scala, F# are increasingly widely used, and powerful features like generics have entered non-functional languages like Java. It’s also possible to get the benefits of functional style when using non-functional languages or multi-paradigm languages like Javascript.

**Problems with Haskell & FP:**

If we want to maintain purity (and FP adherents certainly do), IO is more complicated than it ought to be for beginners. The way Haskell uses the type system to quarantine IO-bound code from pure code means that it is quite a bit harder to get started writing, say, a game than using Python, for instance.

Haskell uses lazy evaluation, so that no expression is evaluated until it is actually needed. As well as having benefits, this can lead to memory leaks which are hard to pin down. The principled approach of FP languages can lead them to be arguably less flexible than imperative languages like C, which is quite happy to give you enough rope to shoot your foot off.

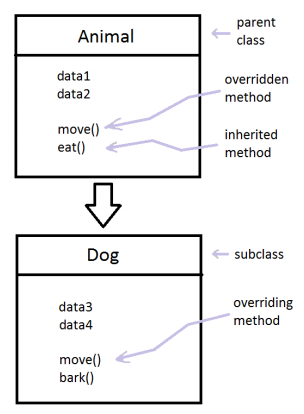
**Object Oriented Paradigm:**

In the 1980s and 90s the Object-Oriented paradigm became incredibly popular. It had been around for a decade or two by then, first appearing in languages such as Simula (1967) and Smalltalk (1980). The basic idea is that we solve problems by creating a bunch of objects that interact with each other by sending messages.

Each object has data and behaviour. The data is stored in fields whereas the behaviour is accessed via methods. Most OO languages are also imperative languages. That is, underneath the concepts of objects and message-passing, the main problem-solving strategies relate to mutating state. One of the main conceptual tools of OO languages is inheritance.

**Inheritance:**

Inheritance allows us to base one set of objects (or class) on another. We can create an entire taxonomy (family tree) to represent the problem.

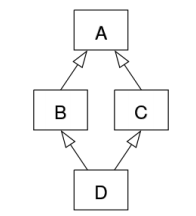


Inheritance provides for ad hoc polymorphism. That is, we can write methods that work with the Animal class, and they will work for all subclasses of that class. Wherever an Animal is required, we can supply a Dog. This is different to parametric polymorphism (generics) which will work for any class, including ones we don’t know about yet.

**Multiple Inheritance:**

Inheritance is conceptually straightforward so long as we only allow single inheritance. That is, each class can have a single parent class. This restriction may stop us from expressing what a class is. Some OO languages, however allow for multiple inheritance.

The problem with multiple inheritance arise when there could be name clashes in the parent classes.



This is known as the diamond inheritance problem. Say A implements a method, foo, and both of its child classes override foo. D doesn’t override foo, so which version of the method should D.foo refer to?

C++ allows multiple inheritance and forces the programmer to manually resolve any ambiguity by doing things such as fully qualifying the method name. Common Lisp provides a default ordering on the parent classes.

Scala and, since version 8, Java get round this by using interfaces which can contain concrete methods. In each of these languages a class can only have a single parent class but can mix in any number of interfaces (called traits in Scala). These interfaces can contain abstract specifications of methods that concrete classes must provide an implementation for, but can also provide default implementations, so this is essentially multiple inheritance.

**OO Design Principles & Patterns:**

In addition to the fundamental ways of structuring OO code (classes, inheritance etc), the OO paradigm encompasses a number of design principles and patterns. OO design principles represent provide guidelines on how to structure your code. An example:



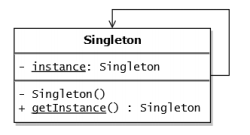
What does this mean? Essentially, it means that we should make as many components of the system (such as classes) final, so that they are closed for modification and other code in the system can rely on them not changing. But we should be able to extend these components using inheritance and by mixing in interfaces.

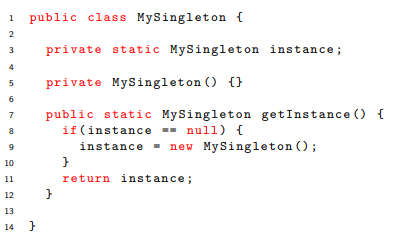
**Design Patterns:**

Design patterns are tried and tested solutions to recurring problems. It’s now a widespread idea, but it began in 90s in the OO world with the “Gang of Four” book. A simple example is the Singleton pattern which can be used whenever we want a class to only ever have a single instance. This instance can be shared by many objects.

**Singleton Pattern:**

The constructor is private, so no one can create a Singleton object directly. To get hold of one, users have to call the getInstance() method, which gives out a reference to the (private) instance object.





The Singleton pattern is criticised because it introduces global state into the system. It has also tended to be overused, cropping up where it isn’t really needed.

**In which domains is OO better than FP:**

Functional languages are great for writing parsers, compilers and such like – symbolic manipulation of data in a pure setting. OO languages are useful where the problem does belong to the kingdom of nouns, e.g. writing a game that includes Sprites, Obstacles and so on that each know how to move around, collide etc., and perhaps complex webapps like online banking.

**Conclusion:**

We could say that the earliest conceptual models of computing, those of Alonzo Church and Alan Turing, led to two broad and largely incompatible paradigms: declarative and imperative. Other paradigms, such as OO, are less fundamental, in a language could allow OO style whilst being a functional language (e.g. O’Caml), an imperative language (C++) or multi-paradigm (Lisp).

Functional programming is a powerful paradigm and its most useful features are rapidly being adopted by more mainstream languages. Each paradigm has its own limitations and killer apps, and a decent programmer should be familiar with several. Since a paradigm is a set of conceptual tools, learning a new one can expand your mind!

**Object Orientation:**

OO has been the dominant paradigm in industry for at least 20 years. It consists of the use of objects, message-passing and concepts like inheritance to decompose problems. As well as providing these concepts, OO comes with a loose set of design principles that are intended to make programs easier to construct, manage and reuse. The two most important principles are loose coupling and high cohesion.

**OO Principles:**

Loose coupling means creating a system in which the parts don’t depend on each other any more than they have to. High cohesion means gathering together logically related data and behaviour in the same place. Encapsulation helps us to achieve loose coupling and high cohesion. This means keeping the data and behaviour that should logically “belong” to a certain class hidden within that class, using access modifiers like private and protected. This is also called information hiding.

Encapsulation is a good thing because it means that other objects can’t reach in and change the state embodied in an object. If all fields are kept private and the only way to access them is via the public methods, then external objects don’t need to know anything about the implementation. This addresses some of the problems Backus was concerned about – big, sprawling programs that share global variables are too difficult to write, mainly because any part of the program could change the global variable at any time.

**Problems with OO:**

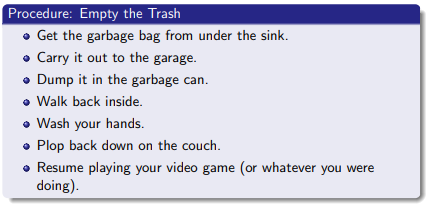
One of the main problems with OO is the overuse of inheritance. This introduces unnecessary coupling in the form of the fragile base class problem: Modifications to the base class implementation can cause inadvertent behavioural changes in subclasses.

Using traditional interfaces (ones without any concrete code in them) avoids this problem because no implementation is shared, only the API. Another way of stating this is that inheritance breaks encapsulation.

**Problems with inheritance:**

Steve Yegge (an influential programmer/blogger who worked at Amazon and Google) wrote a critique of OO called Execution in the Kingdom of Nouns1 . The concepts that OO provides encourages (or forces) the programmer to think about the problem in terms objects (nouns) first and behaviour (verbs, implemented as methods), second. This has the benefit of imposing a very clear architectural structure onto the problem, but of obscuring the actual solution.

Yegge pointed out that the world doesn’t work this way. His example:



Everything we need to do in order to achieve our goal is a verb. But nouns are the only modelling tool that Java provides (unlike other OO languages, like C++). Java forces us to create a system in which nouns like Sink, GarbageBag, GarbageCan etc are the most prominent concepts. In the functional paradigm, on the other hand, verbs (functions) are the most important concept. The main thing is that we want the flexibility to model the problem in the most appropriate way.

**Soild Principles:**

SOLID is an acronym coined by OO expert Robert Martin, representing good OO design principles. It stands for:

S : Single responsibility principle.

O : Open-closed principle.

L : Liskov substitution principle.

I : Interface segregation principle.

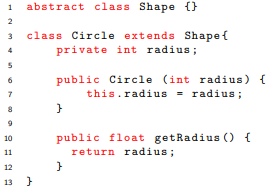
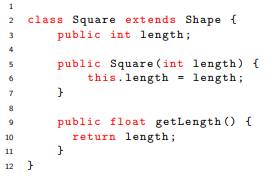
D : Dependency Inversion principle.

**Single responsibility principle:**

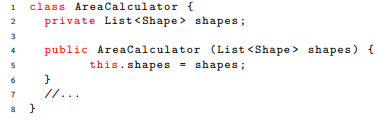
A class should have one and only one reason to change, meaning that a class should have only one job.

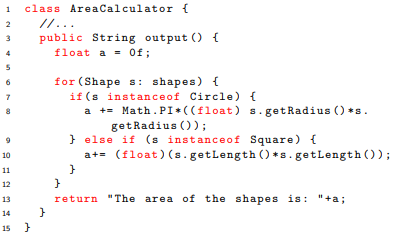
A class should gather together logically coherent data and behaviour. Following this principle will help us achieve high cohesion.

Consider a class hierarchy for shapes:

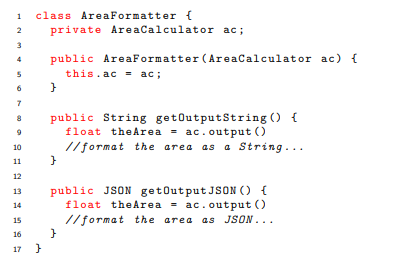
Now we want to construct a class that takes a collection of Shapes and sums their area:





Now what if we want the output in a different format, such as JSON? We need to add methods to the AreaCalculator class, such as outputString, outputJSON, and so on. But the AreaCalculator class is really just for calculating areas. The problem is that it is now responsible for two things, calculating the area and formatting it.

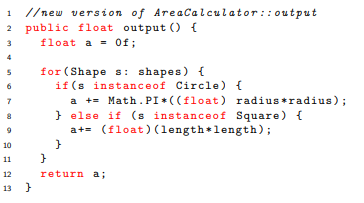
The solution is to put the formatting functionality in a new, completely separate class, say, AreaFormatter. This class can contain all the details of formatting area outputs as JSON, a regular String, HTML or whatever we need. Now we can modify AreaCalculator so that all it can do is work out the area of shapes.



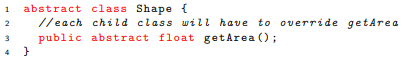
**Open / closed principle:**

Objects or entities should be open for extension, but closed for modification. In other words, it should be easy to add behaviour to a class without modifying the class itself. This principle helps us achieve loose coupling.

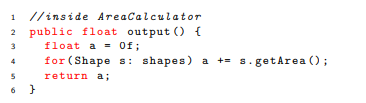
Going back to the AreaCalculator example, what happens to this code if we add a Triangle shape?



We need to modify the AreaCalculator class to deal with new shapes, violating the Open/Closed Principle. A better approach would be to give each class the responsibility for calculating its own area:



Then we can write a method in AreaCalculator that can deal with any sort of shape, since we know they will all have a getArea method:

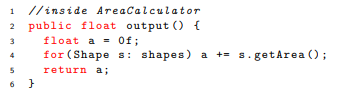


The next time we add a new shape, AreaCalculator will continue to work. We could also achieve this using an interface.

**Liskov Substitution Principle:**

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T. In other words, it should be possible to supply any subclass where its parent class is expected. This principle supports loose coupling.

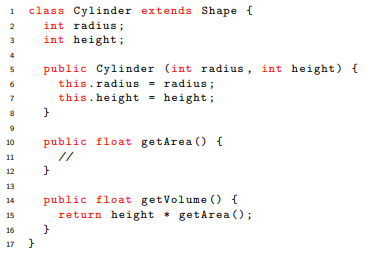
We’ve already seen an example of this in AreaCalculator:



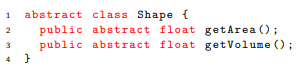
This code works for all subclasses of Shape, even those that haven’t been defined yet.

**Interface Segregation principle:**

A client should never be forced to implement an interface that it doesnt use or clients shouldnt be forced to depend on methods they do not use. Like classes, interfaces should have a single responsibility, supporting loose coupling and high cohesion. Sticking with our example, say we needed to add the Cylinder shape. This is a solid (3D) shape, so we can calculate its volume.

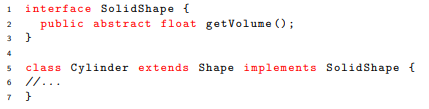


However, if we want to construct a VolumeCalculator class that can calculate the volume of any Shape (sticking to the LSP), we have to force all shapes to have a getVolume method.



Shapes like Circle and Square don’t have any volume. We could create getVolume methods for these classes that always return 0, but this principle tells us we should not do that!

Instead, we’ll create a new interface SolidShape:



Now we can create a VolumeCalculator class that works with any class that implements SolidShape. There is no need to make any changes to the non-solid shapes like Circle, and we can add more solid shapes in future, like a Cube class, by making them implement this interface.

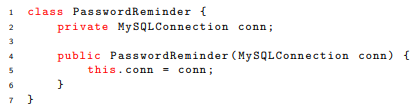
**Dependency Inversion Principle:**

The final principle is concerned with loose coupling. It is also known as Inversion of Control.

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions

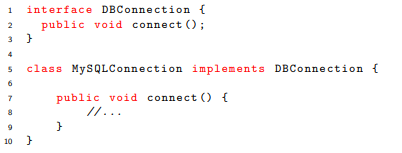
This principle states that all objects should be as loosely coupled as possible, making it easy to extend the system in the future. Dependencies should be abstract, not concrete.

Consider this class, which is part of the backend of a web application:

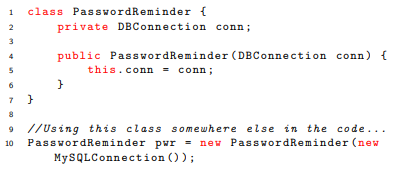


What happens if we decide to start using PostgreSQL instead of MySQL? We need to change this class, and every other that makes use of a connection, violating the Open/Closed Principle.

We can make the system more loosely coupled using an Interface:



Now PasswordReminder doesn’t care what kind of database it uses, so long as it knows how to make a connection:



**Conclusion:**

The OO paradigm offers a set of techniques for structuring code in systems that are easy to debug, maintain and extend, even when they become very large. Used naively, OO is certainly not a guarantee of success. It is all too easy to create brittle systems that virtually need to be rewritten from scratch in order to extend them. The SOLID principles are intended to avoid this situation, but putting them into practice is easier to talk about than to do. It takes hands-on experience. The best way to develop a good sense of design is by working on good code put together by others.

**Design Patterns:**

**Pattern Language:**

The idea of design patterns is due to Christopher Alexander, an architect who wrote about solutions to recurring problems in the built world. An example of a pattern from Alexander’s work: Shelves and other surfaces at waist-height around the perimeter of a room accommodate the transit of objects.

**design patterns in software:**

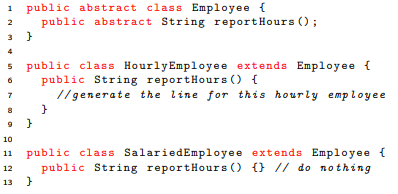
In the mid 90s the idea transferred by analogy to software in the classic book Design patterns : elements of reusable object-oriented software by Erich Gamma et al. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems.

A design pattern describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

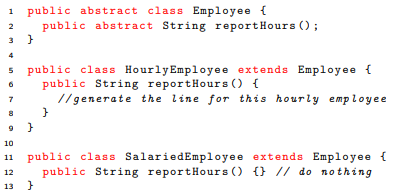
Visitor pattern:

The Visitor design pattern is a way of separating an algorithm from an object structure on which it operates. Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. Visitor creates a separation between objects and the algorithms that work on them. This allows us to add new operations to existing object structures without modifying those structures. It is one way to follow the Open/Closed Principle.

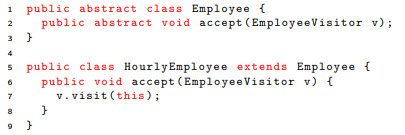
As an example, consider a Payroll system in which some employees are paid a monthly salary, while some are paid by the hour:



We need to ensure that all Employees have the reportHours method so that our code is polymorphic (LSP). But the current situation breaks other SOLID principles. Which ones?

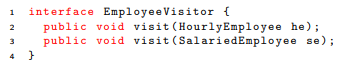


We can fix this by making Employees visitable:

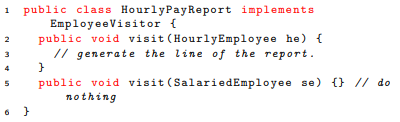


All subclasses of Employee should implement the accept method in the same way.

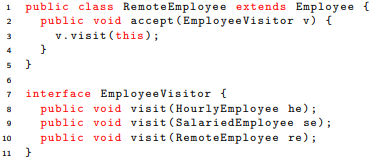
Now we create a Visitor interface with a visit method for each type of Employee:



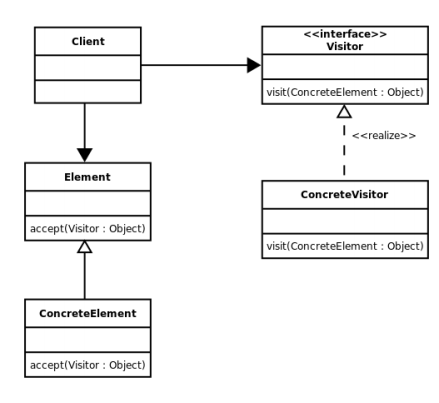
Our concrete Visitor is specialised to a particular task:



If we add more subclasses of Employee we just need to add a visit method for this subclass to the Visitors. If we want a different report, or do something else altogether, like find the average salary, we can produce more Visitors.



visitor pattern:



**Decorator Pattern:**

Another way of overcoming the fragile base class problem is to use the Decorator pattern. The Decorator pattern allows us to add behaviour to an object without affecting any other objects of the same class. We decorate, or add functionality to, objects statically or at runtime by “wrapping” other objects around them.

Imagine we are asked to create stock-keeping software for a Deli. The system will include classes to represent sandwiches, which can be on brown or white bread, and have a variety of fillings, including extra cheese, which determines the cost.

We create a base class, Sandwich. We can then create lots of subclasses, like WhiteBreadCheeseAndHamSandwich and BrownBreadCheeseAndPickleSandwich. That would be a really messy design! We could give up on using inheritance and add fields to Sandwich, like breadType : Enum and fillings: List, but then we aren’t using inheritance mechanisms to help us organise the code.

This is a perfect fit for the Decorator pattern. We will decorate the Sandwich class with extra cheese and fillings. We have an abstract class, Sandwich, with abstract method price() and a concrete implementation class WhiteBreadSandwich, with a starting price of $3.00.

The base class: