Chapter 2

Views of Classification

Classification lies at the heart of object-oriented programming.

By way of exposition, the object-oriented paradigm is introduced as one of the five major programming paradigms. The object-oriented approach is that which emphasises component-centred software construction. Historically, many different areas of computer science and software engineering have been touched by object-oriented programming; the newly-appearing languages are reviewed according to their changing emphases. A definition of what makes a language 'object-oriented' is given, cataloguing the essential distinctive language features: objects, classes, inheritance and polymorphism. Despite this consensus on language features, the primary notion of classification is deeply misunderstood. Some unhelpful ways in which classification has been analysed motivate the need to develop an integrated formal treatment of types, classes, inheritance and polymorphism.

2.1 The Object-Oriented Paradigm

What started out as an interesting set of new programming techniques [BDMN73] that subsequently revolutionised user-interface design [GR83, Gold85] has now invaded the software mainstream [Stro86, Stro91, Cox86, CN91, Meye88, Meye92, Keen89]. Object-oriented programming is a maturing technology, increasingly supported by dedicated analysis and design methods [RBPE91, Booc94, WWW90, JCJÖ92]. Useful overviews of languages and techniques may be found in [Budd91, Hend92].

Initially, object-oriented programming was acclaimed chiefly for its conceptual modelling power, offering a more natural framework for decomposing software systems using human-centred metaphors [GR83, Gold85]. More recent work has identified improved system modularisation effects obtained by imposing an

object-oriented perspective during the design phase [Cox86, Meye88, RBPE91, Booc94]. The current popular focus seeks to exploit the potential of this technology for reducing development costs in the software industry. Cox sees the migration from conventional to object-oriented methods as having the same impact on software development as the industrial revolution had on manufacturing. The transition is seen as equivalent to elevating a cottage craft designing expensive bespoke products to a mass manufacturing industry producing cheap interchangeable components [Cox86, CN91].

2.1.1 The Language Spectrum

There is a wide spectrum of programming languages available to software developers today. From a purely technical standpoint, it is possible to classify them into five groups according to the main structuring concept behind their design:

- command-oriented - a program is a sequence of imperative statements to update the global state of the system; exemplars include the imperative languages: *Fortran*, *Cobol*, *Algol*, *Pascal*, *C*;
- expression-oriented - a program is a nested set of mathematical expressions to be evaluated; exemplars include the functional languages: *Scheme*, *ML*, *Hope*, *Miranda*, *Haskell* and the functional subset of *Lisp*;
- constraint-oriented - a program is a collection of logical equations to be solved; exemplars include the predicate language *Prolog*, some rule-based expert sytem languages and the database query language *SQL*;
- process-oriented - a program is a choreographed dance between communicating processes executing in parallel; exemplars include the MIMD-architecture languages, particularly *Occam*;
- object-oriented - a program is an assembly of interacting components, each hiding local state and offering a set of external services; exemplars include the languages: *Simula*, *Smalltalk*, *Self*, *C++*, *Objective C*, *CLOS*, *Eiffel*, *Beta*, *Sather*, *Trellis*, *Oberon*, *Ada-95* and *Modula-3*.

Each of these groups represents a distinct *programming paradigm*, or conceptual mind-set, which emphasises certain aspects of computation while de-emphasising others. For example, the pure functional languages promote expression evaluation and suppress the notion of state modification. Some languages fall naturally into one or other category; others are hybrids which straddle more than one category. For example, whereas *Smalltalk* [GR83] and *Eiffel* [Meye88] are thoroughbred object-oriented languages, *C++* [Stro91] may be considered an imperative language with object-oriented extensions and *CLOS* [Keen89] supports a dual function/object perspective. Hybrid languages tend to take a weaker theoretical stance, offering a less disciplined collection of heterogenous language features. Pure languages support a smaller set of programming idioms in a clear and consistent way.

2.1.2 Software Components

Object-oriented languages are those which emphasise a component-centred approach to software construction [CH86, Cox86, CN91]. The objects themselves are the software components, each providing a well-specified set of services to client programs or to other objects. While it is possible to argue that all the above approaches support program decomposition of one kind or another, the object-oriented approach has the peculiar advantage that it draws boundaries around useful, reusable parts of a system. One could not easily imagine removing individual commands, predicates or processes from one program and using them directly in another. The effect of a single imperative statement depends too much on the current global system state. Adding a single predicate may affect the inferencing behaviour of a logic program in an unexpected way, since the execution model is implicit and apparently non-deterministic. Adding an extra process to a concurrent system may require changes to the synchronisation strategy to avoid deadlock. Only objects and functions succeed in isolating local system behaviour and therefore may serve as system components. They are contrasted below.

Functions have the property of *referential transparency*, that is, the relationship between a function's result and the values supplied as its arguments is constant. Repeated invocation on the same values always yields the same result. This guarantees predictable behaviour when a function is reused in a new context. However, expression-oriented languages exclude the notion of state and there are many computational tasks such as input, output and long-term storage which are difficult to model without some notion of state.

An object encapsulates a set of related functions and state variables in a single cohesive package. Objects partition the global state of a system. Each object controls a few state variables, to which it restricts access through its functions. The behaviour of an object is locally predictable, but not referentially transparent, since the result of requesting a service from an object is partly dependent on the object's local state, which may change over time. However, from a practical point of view, objects are more useful units of modularity than single functions. Since an object's functions may act upon its state and interact with each other, it makes sense to include or exclude whole objects. While it is possible to import a single function, or an entire subroutine library into a program, these units of modularity are either too fine-grained or too coarse-grained by comparison [Meye88].

2.1.3 Software Architectures

The traditional function-centred development approach [YC79, Myer78, Page88] known as top-down design with step-wise refinement [DDH72, Wirt83] is best suited to one-off systems. This is because decomposing a system *ab initio* around its main functions commits you to major system boundaries which are difficult to move subsequently. Meyer points out [Meye88] that subroutine-sized units of modularity are vulnerable to changes in data formats and difficult to excise from their original context for adaptation in new systems. This is because procedures are tightly-coupled through their parameter lists; the

inclusion of an extra parameter usually initiates a global edit session to update each procedure in a call-graph. His slogan: "Real systems have no top" emphasises the client's perpetually changing top-level requirements during system development, which may have severe knock-on effects to the calling sequence of procedures and the number and types of arguments passed down.

Object-oriented programming promotes design techniques for adapting objects to new requirements. *Classification* is a means of grouping object designs according to their common functions, allowing rapid selection of an appropriate design. *Inheritance* is a way of extending and modifying a design. *Polymorphism* is a means of generalising over several compatible designs. These techniques support the ready adaptation and reuse of plug-in software components [Cox86]. Object designs are the product of analysing three or more separate systems [Booc94], ensuring a degree of generalisation. Objects carry their state with them, so changes to data formats have localised effects, especially since object interfaces are standardised, allowing easier substitution. Programs may be event-driven, allowing easier modification to the collection of top-level services. Objects are compact, but multi-functional, allowing systems to be adapted to more than one use. The only vulnerability of object-oriented systems is to a major restructuring of the communication pattern among objects [JCJÖ92, p135-141].

In contrast to functional decomposition with step-wise refinement, there is no simple handle-turning strategy for object-oriented development. The focus of object-oriented design is a dual one: the developer should ideally keep one eye on the requirements of the system at hand and the other eye on the available components in his library. The iterative refinement of a design is neither wholly top-down, nor bottom-up. Booch has called this approach "round-trip Gestalt design" [Booc94]. Object-oriented software architectures are a compromise between specific and general designs. Several hundred common design clichés, known as *patterns* [GHJV95], have been identified - an example is the concept of an iterator that visits every element of a collection to perform an operation on it. On a larger scale, *frameworks* [JF88, WJ90, Wirf91] provide a reuseable domain-specific harness, or control structure, from which application-specific details have been removed - an example is an event-driven user interface. Whereas a pattern is merely a reusable design, a framework is the dual of a library component - a reusable implementation.

2.1.4 Software Development

Object-oriented programming (OOP) is increasingly supported by commercial object-oriented analysis and design methods. Object-oriented analysis (OOA) elicits required system behaviours and allocates these to collaborating groups of objects; object-oriented design (OOD) directs system decomposition towards minimally-coupled software components with encapsulated state and supports integration with existing reusable library components. While some advocate hybrid development strategies [WPM90] and contend that one or more stages in the OOA/OOD/OOP cycle may be replaced by conventional development techniques, others emphasise the seamless transition between the different phases of object-oriented development [WN94, CY91a, CY91b]. It seems clear

from the above discussion that functional analysis may prove inappropriate for object-oriented development, insofar as this draws system module boundaries in the wrong places. On the other hand, adopting a conceptual approach with OOA/OOD may provide a useful discipline for programmers implementing in a conventional language [Meye88]. The full benefits of reuse are only achieved with the complete OOA/OOD/OOP development path: whereas a design pattern may be portable to a conventional language, a framework depends on specific object-oriented language features such as polymorphism and dynamic binding to couple component parts with the application harness.

Despite fierce commercial competition over the last five years, no single object-oriented development methodology has yet reached a level of maturity comparable with the older functional decomposition methods [DeMa79, Your89, GS79]. All published methods have something to offer. The state of the art is strong on notations and heuristics but generally weak on process.

Premature attempts have merely reworked static entity-relationship modelling techniques [SM88, SM91, CY91a, CY91b, RBPE91], suppressing the important behavioural aspects of objects until too late; or they have reworked dataflow techniques [RBPE91, WPM90] without acknowledging the different directions in which function- and object-based decomposition directs analysis. Synthetic approaches [WPM90, CABD94] do not always apply their borrowed techniques appropriately. The well-established market leaders create a popular appeal [CY91a, CY91b, RBPE91, Booc91, Booc94] by providing diagrammatic notations in which to record detailed designs and program implementation information. These tend to capture and fix the developer's first intuitions about his design and so fail to promote an effective process of discovery, evolution and refinement. Those with greatest industrial experience [RBPE91, Booc94] compensate by offering sets of heuristics and desiderata for well-designed systems. A more imaginative strand [BC89, WW89, WWW90, Gibs90, JCJÖ92, RG92] emphasises the behavioural aspects of objects, striving to retain the plasticity of early analysis and to provide refinement strategies for reworking designs. These have yet to reach critical mass in a commercial context. Other methods are starting to incorporate metrics [HE94] and the wider management perspectives [Lore93, Jaco87, JCJÖ92]. The culture of reuse is still new in software houses. New roles and styles of reward need to be established, to encourage investment in, and reuse of, library components [Booc94, Booc95].

At the time of writing, a horizontal rationalisation is taking place, effectively signalling an end to the "methodology wars" of the early 1990s. For economic reasons, Booch and Rumbaugh agreed from October 1994 to collaborate on synthesising their two popular notations [Booc94, RBPE91]. From October 1995, Booch's company, *Rational*, bought out Jacobson, effectively bringing *Objectory* [JCJÖ92] under the same umbrella. Henderson-Sellers is building a competing consortium to include the *Moses* and *BON* methods [HE94, WN94] and which may attract the behaviour-centred school.

2.2 An Object-Oriented Perspective

Object-oriented products have achieved market penetration in user interfaces, CAD/CAM modelling, databases, office information systems, heterogenous cooperating systems and communications. There are three major world conferences dedicated to object technology (OOPSLA, ECOOP, TOOLS) of which the largest (OOPSLA) attracts some 3000 delegates annually. We expect the fascination with object-oriented programming to continue, not least because the object-concept has proved a powerful concrete metaphor, both in grounding and packaging abstract programming concepts. Different useful aspects of object-orientation have been highlighted as it has migrated from field to field. This is reflected in the diversity and subtly changing emphasis of representative languages as they have appeared over the last 30 years.

2.2.1 Simulation and Distributed Control

When *Simula* [BDMN67, ND81] was designed by the Norwegian Computing Centre, it was originally intended to support discrete-event simulation. With little revision, it actually proved to be a very effective general-purpose language. *Simula* introduced all the important original features of what eventually became known as the object-oriented languages, such as encapsulation, classification and inheritance. Syntactically similar to the *Algol*-family, *Simula* exercised a strong influence on the later design of *Ada* [IBHK79] and *CLU* [Lisk87, LABM81] which adopted its data abstraction, a now recognised technique for reducing software complexity, but not its class inheritance. *Simula* also introduced objects encompassing their own thread of control and supported synchronised coroutines, inspiring later research into parallel and distributed computing. The *Actor* family [Agha86, Agha88, AH87] and *POOL* family [Amer87, Amer90] of languages later exploited asynchronous concurrency and distribution more fully. *Simula* therefore established trends both in data abstraction and parallel computing.

2.2.2 Conceptual Modelling and Interfaces

Developed during the late 1970s at XEROX PARC, *Smalltalk* [Gold81, GR83, Gold85] is chiefly famous for grounding object-oriented concepts in the world's first graphical programming environment. Striving to lessen the abstract conceptual load on the programmer, *Smalltalk* generated a new vocabulary of half-familiar sounding software components, such as *windows, icons, menus, pointers (collectively, WIMP), streams* and *controllers*, many of which had a direct visual representation in the environment. *Smalltalk* expressions, styled as messages sent to objects, are executed immediately in a type-free, dynamically interpreted way. The language cheerfully suffers abuse from the novice programmer and the environment goes to any lengths to offer feedback on the consequences of his actions.

Smalltalk was responsible for generating widespread interest in object-oriented concepts [Gold81]; indeed it first coined the phrase *object-oriented*. Its other impact is felt in the industry-wide adoption of graphical user interfaces as a standard. Smalltalk's WIMP-style interface was copied by Apple Computer for

the *Lisa* workstation and subsequently distributed more widely on the hugely popular *Mac*; which in turn influenced Microsoft's *Windows* and the *OpenLook* and *Motif* variants of *X-Windows*. Development environments for other languages have taken more than ten years to reach the standard set by *Smalltalk*; witness the current fascination with *Visual Basic* and *Visual C++*. To this day, *Smalltalk* remains the language of choice for exploratory programming, especially in the commercial sector in the USA. OOPSLA 1994 reported a large increase in the number of consultancy companies dealing exclusively in *Smalltalk*.

2.2.3 Artificial Intelligence

Classification has been an enduring topic of interest in the field of Artificial Intelligence, where researchers have investigated property inheritance in taxonomic hierarchies [Quil68, Mins75, Brac83, BL85, Tour86]. In the early 1980s, the AI language of choice was *Lisp* and soon object-oriented extensions were appearing. The MIT language *Flavors* [Cann80, Moon86] provided the first models for multiple inheritance and the combination of inherited functions, whereas XEROX's *LOOPS* [SBMC83, BS83, SB86] explored the mixing of functional, object, rule-based and data-driven approaches.

A rationalisation of these experimental languages appears in the *Common Lisp Object System (CLOS)* [BDGK88, Keen89, GWB91], which was the first language to balance the dual function/object perspective. *CLOS* also has the most sophisticated reflective facility, whereby the language's execution model is handled by a self-describing kernel of meta-objects [KRB91].

2.2.4 Systems Programming

In the mid 1980s, the focus moved to mainstream programming. *Objective C* [Cox86, CN91] and *C++* [Stro84, Stro86, Stro87, ES90, Stro91] were both extensions to the systems programming language *C*, but had different emphases.

Objective C provides a *Smalltalk*-like interpreter on top of *C*. With its commitment to open-ended interfaces, it was the first language to support the widespread exchange and sale of pre-compiled software components on standard hardware platforms; this potential has been dramatically realised with the adoption of *Objective C* at the heart of the *NeXTStep* operating system [NeXT93] and the exponential growth in applications generated and exchanged within the *NeXT* user community. Objective C was responsible for the creation of the first market in compiled objects; the name *applets* (mini applications) was coined for products which could be sold or traded over the Internet and executed immediately on the local host machine.

C++ was designed to be "a better *C*" and incorporated independent extensions for data abstraction and object-orientation at a much lower level [Stro84, Stro88]. *C++* is a statically compiled language, offering greater efficiency but less open-ended compatibility than *Objective C*. *C++* affords a more finely-tuned control over program behaviour and the use of resources than its rivals.

The features of *C++* all act independently - most combinations have a legal meaning. The language is so complex and adaptable, that strict house style guidelines become necessary [Lipp91, DS89]. The widespread commercial migration to *C++* has mostly to do with existing heavy investments in *C* as the systems programming language of choice. For good or ill, the use of *C++* has reached critical mass and the language is likely to dominate systems programming for the forseeable future.

2.2.5 Software Engineering

In the late 1980s, the emphasis moved to software quality issues, such as correctness and robustness. More streamlined languages appeared, having fewer features that were used in more disciplined ways. These were intended to enforce regular programming practices and so reduce the scope for introducing software errors.

Influenced equally by *Simula* and *Ada*, *Eiffel* [Meye88, Meye92] is the best exemplar of this approach. Statically compiled, it retains a *Smalltalk*-like open-ended flexibility and has an *Ada*-like parametric polymorphism. It is chiefly famous for incorporating executable specifications, the pre- and post-conditions and data type invariants [Hoar72] found in formal methods such as *VDM* [Jone86] and *Z* [Spiv89]. Objects enter into contracts with each other, guaranteed by the terms of specifications. These have a dual role: firstly, they document the intended semantics of functions and secondly they determine responsibility for exception handling if a contract is broken. *Eiffel* is slightly less efficient than *C++* but much more secure.

Trellis [SCBK86] adopted an even more conservative strong typing scheme in the wake of the first formal analysis of classification as a kind of subtyping. *Sather* [Omoh94] is an *Eiffel*-like language with a *Trellis*-like type system. *C++* intends to incorporate an *Eiffel*-like exception-handling mechanism from version 4.0. *Ada 95* [ABBD95], which is properly object-oriented by virtue of having added inheritance to the original *Ada* definition, may also be considered a member of this group of languages.

2.2.6 Database Technology and Hybrids

One significant hybrid group are the object-oriented database languages [Fish87, MSOP86, AHS89]. The main impetus behind this group is the desire to extend the modelling power of the long-held relational model. The normalising force of the latter has the advantage of reducing data-dependency, but the disadvantage of splitting logical entities over many files. A functional query model [Ship81] is used as an alternative to *SQL* to address whole objects. Furthermore, the query language dovetails more closely with the programming language in which the database is embedded.

For reasons of space, we do not discuss here in detail a wide variety of other hybrid approaches, such as the hybrid concurrent and distributed languages, hybrid logical languages and object-oriented extensions to conventional imperative languages. The concurrent and distributed languages focus on the

local encapsulation of objects that communicate over wide area networks via message passing. Typically, a coarse-grained parallelism is adopted, with proxy objects acting as local shadows for remote objects. Research concerns include object access and migration, locking strategies and authorisation. The hybrid logical languages mix object access styles: query-by-value is mixed with navigation-based access. The extensions to conventional languages, such as *Object-Oriented COBOL*, can be regarded mostly as attempts to keep up with the latest fashion, rather than any serious attempt at language redesign.

2.2.7 Experimentation and Minimalism

Finally, three languages worthy of particular mention for their experimentation and originality are *Self* [US87] *Beta* [Mads93] and *Java* [Sun95].

Self was intended as a stripped-down *Smalltalk*. It diverges from all the languages above in that it has no classification and no fixed notion of inheritance. Objects are cloned from each other and may delegate requests to any object of their choice. The dynamic nature of *Self* forced the development of most known strategies for the automatic optimisation of object-oriented programs [CUL89, CU90].

Beta is a successor to *Simula*. It is remarkable in that it generalises over every programming language abstraction. It has only one construct: the *pattern*. A pattern may declare data (a record) or functions (an encapsulated type); it may define a body (a procedure) or have all of these things (an object encompassing its own thread of control).

Java [Sun95] was developed by Sun Microsystems as a minimal object-oriented language in the style of C++ (but not restricted by the requirement to be compatible with C). *Java* is unusual in that it is an interpreted language, designed to interact with multimedia components over the World Wide Web on the Internet. A specialised Web browsing tool, *Hot Java*, allows *Java* programs, known as *applets* to be pulled from any remote source and interpreted on the local host machine.

2.3 Defining 'Object-Oriented'

Later chapters will discuss the technical merits of object-oriented language features, especially their amenability to formal analysis. The question arises as to *what* features constitute an *object-oriented* language [Stro88]. In his seminal analysis [Wegn87], Wegner initially classified languages into groups such as: *object-based, class-based, prototype-based, inheritance-based, delegation-based* and *object-oriented*; particularly relevant to us is one contrast:

- object-based - having the notion of objects, but not inheritance; exemplars include *Modula-2* [Wirt82], *Ada* [IBHK79] and *CLU* [LABM81];
- object-oriented - having the notion of objects and class-based inheritance; exemplars include *Smalltalk* [GR83], *C++* [Stro91] and *Eiffel* [Meye88].

This more or less established a *de facto* standard for canonical object-oriented languages, which should support the classification paradigm. This breaks down into the features: *objects, classes, inheritance* and, as a consequence, *polymorphism*.

2.3.1 Objects

An object is a perceptual entity with crisp boundaries [Booc94]. In the real world, we call things objects if they are tangible or physical things which we can manipulate. Although object-oriented languages may vaunt their ability to model real-world entities, or to provide physical metaphors for software concepts, having a tangible counterpart is not especially what constitutes a software object.

In computer science, the term *variable*, denoting a static, compile-time program text concept, is sometimes contrasted with *object*, a dynamic run-time software concept. As a result of assignment or binding, the relationship between variables and the objects they contain may change over time during program execution. Although we may refer variously to things such as *integer objects* or *functional objects* in languages which have such concepts, object-oriented languages denote something more specific by the term.

An object is a software entity possessing the properties: *identity, state* and *behaviour* [Booc94]. The *identity* of an object is the unique handle on that particular object. It is often implemented as a memory address, but this is not inevitable. *Smalltalk*, for example, supports a virtual memory system. All object references are unique indices into a table, which may translate to a memory address, or to an offset in a file - the identity of an object is therefore location-independent. The *state* of an object is its private memory store, holding the data which it controls. This is realised as a block of memory (or disk space) accessed by its identity. The *behaviour* of an object is the set of external services it offers, or tasks it is capable of performing. These services are implemented as functions owned by the object, which may access or modify its state. The association of functions to objects is something which may exist only at compile-time; or else an object may store an additional pointer to a table of its functions. Objects are analogous to machines [Meye88] responding to requests to inspect, and commands to update, their internal state.

The formal significance of programming with objects is often missed. The properties of *identity* and *state* distinguish objects from values (in expression-oriented languages) and the property of *identity* distinguishes objects from tuples (in constraint-oriented languages). McLennan suggested that the key distinguishing property of an object is its mutable state [MacL82]. Mathematical values do not change over time, in the sense that 3 may never become 4. Functional expressions also disregard identity - it is immaterial whether a value is passed as 3, or as the different but equivalent (1 + 2). By contrast, it is immensely significant which of two objects are being manipulated, even if they temporarily have the same state. The loss of identity has no effect in expression-oriented languages, but may have one in constraint-oriented languages. The tuples of a relation are accessed by content only, some state

values constituting a key index. As a result, two tuples having the same state may accidentally be merged, since they are considered identical. By contrast, an object's identity is unique and independent of its state.

2.3.2 Classes

A class is a collection of things grouped according to some external criterion. Classification carries with it the idea of a hierarchy of subdivisions, for example the familiar biological taxonomy of animal genera and species.

Objects are grouped by their external behaviour into classes. A class describes a family of objects, up to a certain level of generalisation, which have the same observable behaviour. For example, the class COMPARABLE describes all objects that have the inequality functions {<, >, <=, >=}. This may include objects with strictly more functions, such as integers or reals. The objects whose behaviour is wholly described by a given class are called *instances of* that class. Objects may otherwise be considered members of a series of increasingly more general classes, each one subsuming the last. For example, 3 may be an instance of the class SMALL_INTEGER, but may also be considered a member of the more general classes INTEGER, COMPARABLE and OBJECT.

Nearly all languages overlay the term *class* with additional meanings. The class construct is chiefly a program text concept [Meye88] for defining objects. In this respect, a class functions variously as:

- a record template, defining the data storage required by its instances;
- a type specification, defining the functional interface respected by its instances;
- a table of values, defining data and functions shared by all its instances;
- a module, packaging data and function definitions and controlling their visibility.

Figure 2.1 illustrates an example primitive CIRCLE class, using *Eiffel* syntax for clarity. A class definition usually encompasses a set of variable declarations (defining state) and a set of function definitions (defining behaviour). Some data declarations may be initialised with values to be shared by all instances. *Smalltalk* calls these *class variables*. Most data declarations simply reserve storage space in each individual object. *Smalltalk* calls these *instance variables*. All instances of a class typically share the same functions (*pace* [Keen89]). Any shared material is hived off into a precomputed table by the compiler. Access to data may be solely through functions [GR83, CN91, Keen89] or the language may supply a separate export mechanism [Meye92, Stro91]. The interface of a class is the set of signatures corresponding to its externally-available components. Even type-free languages have the notion of class *protocols* [GR83, NeXT93]. The name of a class may be used as a type identifier in strongly-typed languages. For all intents and purposes, *type* and

class are considered identical notions - this view comes under critical examination later.

```
class CIRCLE
creation make                          -- identify name of creation procedure(s)
feature { ANY }                        -- export to all with read/execute access
        -- data declarations
   radius : REAL;                      -- allocated to each circle
   pi : REAL is 3.1415926;             -- shared by all circles
        -- function declarations
   make (r : REAL) is                  -- procedure to initialise a circle
        do radius := r end;
   diameter : REAL is                  -- function to compute diameter
        do Result := 2*radius end;
   circumference : REAL is             -- function to compute circumference
        do Result := 2*pi*radius end
   area : REAL is                      -- function to compute area
        do Result := pi*radius*radius end
end -- CIRCLE
```

Figure 2.1: Class Definition

Encapsulating data with access-functions is known as *data abstraction*. Languages with data abstraction allow user-defined (concrete) data types to have the same status as primitive in-built types [LZ74, LZ75, LZ77, Lisk80, Stro88]. Wegner called *CLU class-based* because it offered data abstraction [Wegn87]. Clearly, data abstraction derives from *Simula*'s classes, however classification and data abstraction should really be considered distinct notions - *CLU* has no class hierarchy [LABM81, Lisk87]. The notion of *encapsulation* dates from Parnas' dictum on modularity [Parn72], in which highly cohesive components are assembled behind an abstraction barrier. The main purpose of this is to reduce external dependency and thereby foster a loose coupling between modules. Encapsulation has two faces:

- information-hiding - it relieves the module user of the need to know the details of a module's implementation; this measure is intended to counter complexity;

- protection - it prevents the module user from tinkering with the internal details of a module's implementation; this is a security measure.

Meyer identifies *class* with *module* in this sense [Meye88], since an object's state and functions are highly cohesive. Furthermore, if a class defines a data type and a module controls the visibility of its operations, then *type* and *module* must always be coincident (unlike *Ada* and *Modula-2*). The notion of *abstract data type* was developed in a more mathematical direction by Guttag, Reynolds and others [Gutt75, Gutt77, GH78, Reyn74, Reyn75, Reyn83]. An *abstract data type* is an algebraic specification, in terms of function signatures and axioms. In this context, it is misleading to refer to user-defined (concrete) types as *abstract data types*, where the phrase *encapsulated data type* is more appropriate.

A further overlaid meaning arises from certain languages' admitting classes as first-class elements [GR83, Keen89]. Here, classes are themselves considered objects. They exist at run-time and may respond to requests, particularly to generate new instance-objects on the fly. *Objective C* distinguishes the notions of *class* (external descriptor) and *factory object* (run-time instance-object generator) [CN91]. The behaviour of class-objects is mediated through *metaclasses*, which recursively describe classes and themselves [KRB91].

2.3.3 Inheritance

Biological inheritance is the enjoyment of genetic qualities you did not strive to develop. In probate law, inheritance is the disposition of acquired wealth you did not have to earn. In both cases, it involves obtaining something from your forbears for free (but outside of your control). By metaphorical transfer, the term is linked to classification schemes in AI, which propagate descriptive properties in inheritance graphs.

Taxonomic classification is based on the idea of subdividing groups according to a few properties that become salient at a given level of specialisation. Whereas the class of *mammals* is distinguished from *birds* or *reptiles* by having warm blood, giving birth to live young and suckling, the mammalian subclass of *carnivores* is further distinguished from *herbivores* by having forward-pointing ears and eyes and sharp teeth. Early models of human memory organisation [Quil68] and knowledge representation [Mins75] adopted such a hierarchical network model for clustering information. According to the *principle of cognitive economy*, only salient distinguishing properties were represented at each node and the remaining class properties were *inherited* implicitly from more general classes.

In object-oriented programming, a class may be defined to inherit from another, in which case it immediately obtains all the data and function declarations of its parent. Inheritance is a short-hand mechanism for defining classes incrementally, based on existing parent classes. The child class need only specify what makes it different from its parent. A hallmark of a well-designed object-oriented library is the factoring of classes into an *inheritance hierarchy*, in which minimal additions are made at each level [GR83]. The process of abstracting out classes at intermediate levels of generalisation requires greater design effort, but provides more anchor-points from which to derive subsequent class variants. Inheritance is regarded as crucial to software extensibility and flexible reuse. A module that is already in use may be subsequently adapted and extended, without loss of security to existing clients, through inheritance; this is known as the *open-closed principle* [Meye88].

The mechanisms of inheritance are highly complex. Inherited instance variables are implicitly added to those defined locally. Together, they form a larger record template for generating instances of the child class. Inherited class variables are still unique but their scope extends to instances of the child class. Inherited functions are typically not copied, but may be applied directly to instances of the child class. Here, recompilation is avoided, either by preserving the linear order in which instance variables are added - with *single*

inheritance, inherited functions access the same offsets in child class instances [GR83, CN91]; or alternatively by providing an indirect indexing scheme [Stro87, Stro91, Meye92] or dynamic function access [Keen89] - this is usually necessary in languages having *multiple inheritance*, where a child class has more than one parent and therefore inherits the second and subsequent parents' instance template out of linear order. Multiple inheritance gives rise to *feature clashes*, where a class may inherit the same named declaration from more than one parent. Languages either force the disambiguation of clashes through name qualification [Stro91, Meye92] or determine that one declaration has priority [Moon86, BDGK88, Keen89] using automatic conflict-resolution schemes, or else they merge the declarations.

```
class CYLINDER
inherit CIRCLE
   redefine area                      -- to mean total surface area
creation make
feature { ANY }
         -- additional data declarations
   height : REAL;                     -- allocated to each cylinder
         -- additional and modified functions
   make (r, h : REAL) is              -- procedure to initialise a cylinder
         do radius := r; height := h end;
   area : REAL is                     -- modified to compute total surface area
         do Result := 2*pi*radius*(radius + height) end;
   volume : REAL is                   -- additional function to compute volume
         do Result := pi*radius*radius*height end
end -- CYLINDER
```

Figure 2.2: Class Definition with Inheritance

Inheritance is mostly used in an additive way, whereby a child class simply adds extra data and function declarations to those inherited from its parent. Inheritance may also be used to modify the behaviour of the parent class in the child. An inherited function may be replaced by an alternative definition in the child class, either for efficiency's sake [Meye88] or because the operation has slightly changed in meaning [GR83]. This is called *overriding* or *redefining*, since the inherited function is typically hidden and no longer available. An exception to this is where the replacement function incorporates, or invokes, the inherited version. This is known as *method combination* [Moon86, GR83, Kee89]. Usually, replacement functions must respect the type signature or protocol of the functions replaced [Stro91, Meye92, NeXT93] but this is not always enforced in the language definition [GR83]. *Smalltalk* judges function compatibility by name equivalence only, opening the way to type-unsound substitution. Programmers typically follow their own protocol disciplines. In strongly-typed languages, data declarations usually have constant types [Stro91] but they may be retyped with a more restricted type in some languages [Meye92]. This is intended to reflect the specialisation of component parts along with the enclosing whole.

Figure 2.2 illustrates a simple example of inheritance in Eiffel. In this example, CYLINDER inherits *radius*, *pi*, *diameter*, *circumference* from CIRCLE, replaces

area and adds *height* and *volume*. Note how *area* and *volume* may use inherited features *radius* and *pi* directly. Inheritance often has the effect of partitioning the parent class into one or more specialised subsets. Here, it merely has the effect of extending and modifying the parent's structure and behaviour - it would be strange to think of cylinder-instances as members of a more general CIRCLE class. We leave this example as a deliberate provocation!

2.3.4 Polymorphism

Coming from the Greek *poly* (many) and *morphe* (form), *polymorphism* is a term from type theory denoting a generalisation over types. Traditionally, the strongly-typed programming languages are *monomorphic*, that is, variables are given exactly one type and may only be bound to values having this type. A *polymorphic* language is one in which type constraints are systematically generalised. Variables may be bound to values having more than one type. This opens the way to generic styles of programming, in which algorithms which behave systematically over families of types may be encoded in an economical way.

Strachey and others first identified families of types that were sufficiently similar in structure that one could write polymorphic functions acting uniformly over them [Strac67, Strac73, MS76, Tenn81, Reyn83]. These were typically the container types, such as lists and stacks, for which functions like *cons*, *append*, *push* and *pop* could be written irrespective of the type of element they contained. Tennent [Tenn81] recommended the use of type parameters to abstract over the unknown parts of each type. Elsewhere, Strachey noted a tendency in existing programming languages to provide functions with multiple definitions. The operator + might be used to add integers and reals, but then also to concatenate strings and append lists. He distinguished:

- *parametric polymorphism* - parameterised functions acting in a systematic way over a variety of types; this is usually known today as *genericity* [Meye88];
- *ad hoc polymorphism* - the undisciplined adding of new meanings to old function names; this is usually known today as *overloading* [Stro91].

Strachey rejected *ad hoc* polymorphism on the grounds that it was not amenable to formal analysis. No semantic correspondence need exist between the different definitions overloaded on a single function name. Explicit parametric-polymorphic mechanisms later entered into the designs of *ML* [Miln78, MTH90] *Hope* [BMS80] and *Ada* [IBHK79]. In *ML*, sophisticated type inference is used to propagate actual type information into type parameters at function call sites. In *Ada*, generic packages must be instantiated with actual types before use - the compiler generates a separate image for each instantiation.

A further distinct kind of polymorphism results from inheritance. This is sometimes referred to as *inclusion polymorphism* [CW85]. An inherited function is naturally polymorphic, since it is by definition applied to instances of

many classes. However, a restriction is placed on these classes: they must always inherit (directly, or transitively) from the class which defined the function. The inheritance hierarchy therefore defines a bound on polymorphic function application. Object-oriented languages may combine inclusion with parametric [Meye88, Stro91] and *ad hoc* [Stro91] varieties. *Eiffel* has a form of *constrained* parametric polymorphism and a form based on *type anchors* [Meye92]. The model of type polymorphism described in later chapters seeks to harmonise these approaches insofar as they act in a systematic way.

Inclusion polymorphism defines a type-compatibility relationship in the strongly-typed languages [Stro91, SCBK86, Meye92]. A variable with a class-type is considered inherently polymorphic. It may be bound to an object of its declared type, or of some other inheriting type. The declared type of the variable is the upper bound on the type of object it may receive. This unidirectional relaxation of the type system is regarded as secure, since any descendent class is expected to have *at least* the functions declared in the parent's interface, either because it inherits the parent's functions, or provides its own replacement versions. These security claims are reviewed in later chapters.

```
c : CIRCLE;
circ : CIRCLE;
cyli : CYLINDER;
...
!! circ.make(3.0);              -- create instance of default type CIRCLE
!! cyli.make (4.0, 7.5);        -- create instance of default type CYLINDER
c := circ;
...  c.area;                    -- dynamically select CIRCLE's area
c := cyli;
...  c.area;                    -- dynamically select CYLINDER's area
```

Figure 2.3: Polymorphism with Dynamic Binding

Related to polymorphism is the issue of binding. The goal of polymorphism is to permit the writing of generalised algorithms. In object-oriented programming, this is frequently realised by defining a basic function for a general class and reimplementing this function in more specific descendent classes. As a result of function replacement during inheritance, there may exist many variants of the same basic function in descendent classes. Programs are characterised by the use of *dynamic binding*, whereby a general algorithm is interpreted locally by objects in different ways. Figure 2.3 illustrates both polymorphic assignment and the dynamic selection of different object responses, using the example *Eiffel* classes from figures 2.1 and 2.2. Here, even though the polymorphic variable *c* has the *static type* CIRCLE, the language's binding mechanism ensures that an appropriate function is called for the *dynamic type* of each successive object stored there.

Smalltalk [GR83] was influential in establishing this style of programming. Adopting the message-passing model [Hewi77], Smalltalk distinguishes *messages*, the requests sent to objects, from *methods*, the functions implementing the objects' responses, defined in their classes. The jargon uses the term *method* to emphasise the fact that a function is owned by a particular

class; it is not free-standing and may only be invoked by selecting it from its class. This idea is further reinforced by the polymorphism of messages - each class has its *own method* for dealing with the request.

Message passing with dynamic binding is so characteristic of object-oriented programs that some treatments incorrectly use the term *polymorphism* to mean dynamic binding, especially where the language has no notion of types [GR83] or contrasts *polymorphism* with *genericity* [Meye88], the statically-bound variety. Hereafter, the typing and binding issues are properly distinguished. Polymorphism describes first and foremost a generalisation over types; how this affects binding is a secondary matter. Message-based dynamic binding has the effect of confusing systematic and *ad hoc* varieties of polymorphism in object-oriented programming. The introduction of functions with the same names [GR83, CN91] and same protocols [NeXT93] in disjoint parts of the hierarchy is open to the same abuses as arbitrary overloading. Against this, good practice will impose a common type family [SCBK86, Meye92] on functions that are redefined below a given class, and advise against multiple reintroduction.

A particular use for this is found in the creation of *deferred* [Meye92] or *abstract* [GR83] classes. A deferred class has one or more *deferred functions*, whose signatures [Stro91] and specifications [Meye92] only are given. The bodies of such functions are not supplied at this level of generalisation, but are defined in different forms in descendent classes. The purpose of this mechanism is to provide a strong typing and semantics for a family of poymorphic functions. Program expressions containing variables typed in the deferred class may receive any descendent instance which implements these functions. Deferred classes support the writing of generalised algorithms and the combination of polymorphic typing with dynamic binding supports plug-in software components. Properly speaking, an *abstract* class is one having *all* of its functions deferred [Meye88]. Such a construct may rightly be called an *abstract data type*, since it provides no concrete implementation.

Languages may use dynamic binding universally [GR83], frequently [CN91, Meye92], or rarely [Stro91]. An analysis of many programs has revealed that about 20% of object-oriented code needs dynamic binding [Booc91, Simo92]. Strongly-typed languages can perform at least a partial static analysis of bindings. Dynamic by default, *Objective C* will revert to static binding if static types are supplied [CN91]. *C++* will only use dynamic binding if functions are marked *virtual* [Stro91]. Dynamic by default, *Eiffel* will optimise static bindings as a compiler post-process [Meye88]. Central to the efficient handling of dynamic binding is the creation of a *dispatch table* by the compiler. This grows in the product of classes and methods that use dynamic binding. In languages with a high commitment to dynamic binding, optimisation strategies must be found to reduce the size of the table. This may be in the form of selector index colouring [DMSV89], or other table packing mechanisms [Dries93, AGS94, DH95].

2.4 The Challenge of Classification

What is classification? Object-oriented languages are confused in their formal appreciation of *class*. This is somewhat alarming, since classification is arguably what makes a language distinctively object-oriented. Rather than rise to the challenge, some past analyses have pushed this issue aside. The issues are presented below, together with an argument for seeking a more satisfactory interpretation of *class*.

2.4.1 Class Ambiguity

Perhaps the biggest obstacle to formalising the notion of *class* comes from the multivalent use of the concept in existing languages. By default, one is left to assume that *class* replaces the usual notion of *type*, especially in languages without strong typing [GR83]. Class names are used everywhere as type identifiers [CN91, Stro91, Meye92]. On the other hand, a class is different from a type in that it is open to extension - what are open-ended types? Then again, a class is like a package or module [IBHK79, Wirt82], the concrete implementation of a type. But what kind of construct is an open-ended implementation? There are two main dimensions of ambiguity:

Concrete - Abstract Dimension

- class as specification - providing the visible type, the signature interface (protocols) for a family of objects, its instances;
- class as implementation - providing the instance template, a class table of shared data, a collection of function definitions and modular packaging.

Monomorphic - Polymorphic Dimension

- class as monotype - providing the exact concrete type of its own instances, used when creating instances;
- class as polytype - providing the bounded polymorphic type for members of its larger family, used when attached to inherited functions.

The overloading of the term *class* has for long obstructed clear thinking. To facilitate a better understanding, these meanings must first be unravelled. The concrete-abstract dimension has been considered previously [Sakk89, PW89]. It is examined in some detail below.

2.4.2 Convenience View

The *convenience view* [Simo93] treats the class purely as an implementation construct, a kind of extensible record. It concentrates on the internal structure, in terms of storage for data attributes and methods. Inheritance is a shorthand for defining larger records incrementally by extension; and is often used in preference to composition when creating structured classes.

Relegating the class to a mere unit of implementation, while formally lax, does bring certain efficiency gains:

- maximum reuse of implementations;
- economy in levels of indirection in structures;
- economy in levels of nesting in call-graphs.

To illustrate and elaborate on these advantages, consider the derivation of RECTANGLE as a subclass of POINT, in figure 2.4 below:

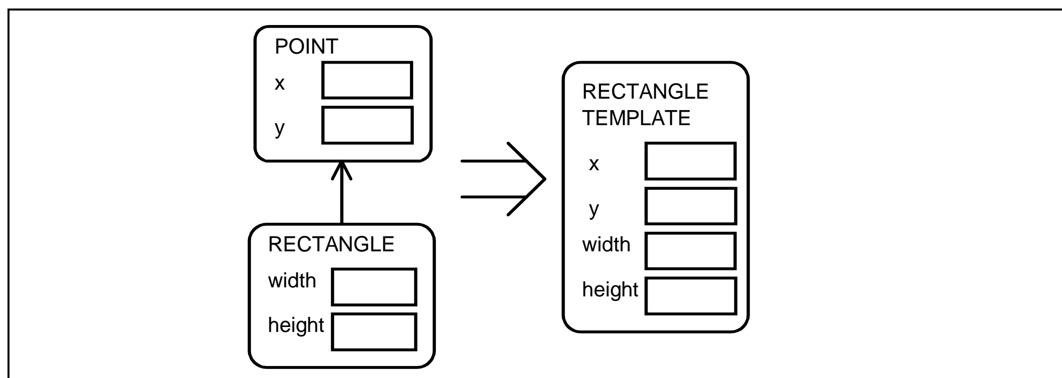

Figure 2.4 Implementation Inheritance

By inheriting from POINT, the RECTANGLE class obtains all of POINT's data and functions in an unencapsulated way. It may access the *x* and *y* coordinates of its location directly, without having to go through POINT's interface. Where object references are implemented through pointers, one level of indirection is removed by expanding all of POINT's contents at the top level.

However, such a design begs the question whether a RECTANGLE is truly a kind of POINT in any sensible taxonomy of geometric shapes. A RECTANGLE could equally be composed of POINTs; or perhaps a POINT might be thought of as a degenerate kind of RECTANGLE. Maximising reuse of implementation can lead to strange, sometimes counter-intuitive abstractions. Figure 2.2 above also adopted this convenient strategy when deriving a CYLINDER from a CIRCLE.

2.4.3 Ambitious View

The *ambitious view* [Simo93] treats the class as a kind of incremental specification. It concentrates on the external, behavioural aspects, in terms of function signatures and, in the case of *Eiffel* [Meye92], the semantics of these functions. Inheritance describes the evolution of a specification which is gradually made more concrete.

Elevating the class to a unit of specification brings with it a flavour of:

- type development (adding new function signatures, extending behaviour);
- type reification (implementing deferred functions; refining the definitions of existing implemented functions);
- type restriction (subclassing seen as forming disjoint subsets; subtypes having alternative specialised behaviours).

Meyer's promotion of classes as specifications with deferred implementations [Meye88, Meye92] contrasts most strongly with the convenience view. This is illustrated in figure 2.5 with abstract STACK and QUEUE classes which have multiple alternative concrete implementations:

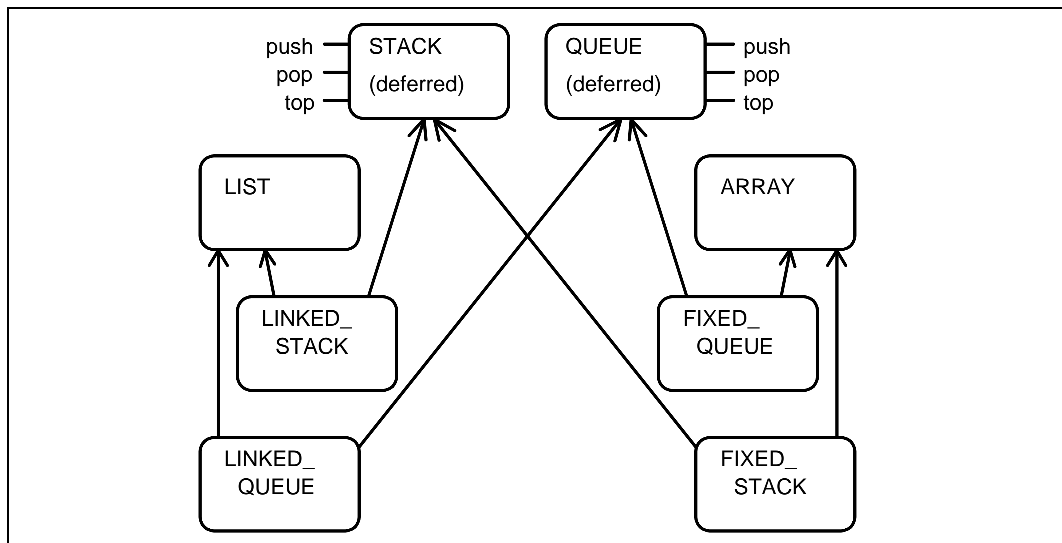

Figure 2.5: Specification Inheritance

Here, inheritance corresponds almost exactly to a process of reification. The STACK class provides deferred routines (≈ code stubs) for operations *push()*, *pop()* and *top()*, whose semantics are verified using axiomatic assertions. The concrete LIST and ARRAY classes provide alternative implementations. By inheriting jointly from STACK and LIST, the LINKED_STACK class identifies *push()* with LIST's *cons()* operation, *pop()* with *tail()*, and *top()* with *head()*. By contrast, FIXED_STACK inherits from STACK and ARRAY, adds another *count* attribute and uses this as the index into ARRAY addresses in the alternative implementations of *push()*, *pop()* and *top()*. Both implementations have to conform to the LIFO semantics of STACK. A similar treatment applies to the reification of QUEUEs.

Classes occupy a whole spectrum, from abstract to concrete. Many are partially concrete specifications, subject to increasing refinement. Inheritance is seen as expressing a type compatibility relationship between more refined classes and their ancestors. Classes are considered to be types [Stro91, Meye92] with attached implementations.

2.4.4 Tension Between Views

The tension between implementation and specification concerns was documented as early as 1987 [Lisk87]. This aspect was recognised as a contrast between *essential* and *incidental* inheritance [Sakk89] or *strict* and *non-strict* inheritance [PW89]. Whatever the nomenclature, the strong variety of inheritance implies at least a sharing of class specification by which all descendent classes must be bound. The weak variety implies only implementation sharing, in terms of the opportunistic reuse of code and declarations for storage allocation. The strong kind may occur with some implementation sharing, since most languages strive to map in a fairly straightforward way from abstract types onto their concrete counterparts.

However, opposing ambitious and convenience pressures can lead to divergent designs. Implementation concerns creep into abstract design in undesirable ways. In figure 2.6 we illustrate a case from [Meye88], in which POLYGON is intended as the abstract super type of all geometric figures. However it is also used to model concrete n-vertex polygons and, as such, defines a routine *addVertex()* to adjust its geometry to suit. Now, the descendent class RECTANGLE should not be allowed to add to its vertices. *Eiffel* has orthogonal inheritance and export mechanisms, so the inherited routine *addVertex()* is simply not exported in RECTANGLE.

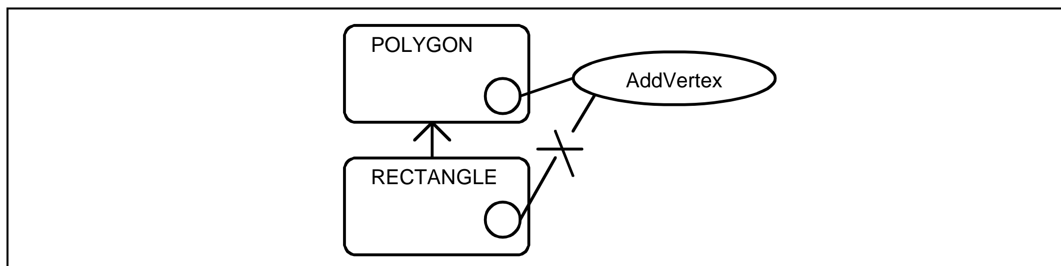

Figure 2.6: Selective Inheritance

This kind of selective inheritance was criticised in [SC92] since, in our view, it leads to type violation: RECTANGLE does not respond to all the public functions of POLYGON, therefore it cannot be a kind of POLYGON. *Eiffel* version 3 [Meye92] now includes selective export and feature undefinition mechanisms which exacerbate this case. The fault lies not so much with the incorrect export rules, but in expecting the POLYGON class to fulfil both abstract and concrete goals. Enforcing proper type abstraction would lead to a separation of concerns into a deferred GEOMETRIC_FIGURE class and a concrete (n-vertex) POLYGON class.

In *C++* [Stro91] there is the first recognition of strong and weak inheritance fulfilling different purposes. Classes are considered types, but sometimes inheritance is not treated as subtyping. The language supports both *private* inheritance, in which a derived class inherits only the implementation of its parent, and *public* inheritance, in which a derived class also inherits the (entire) specification of its parent.

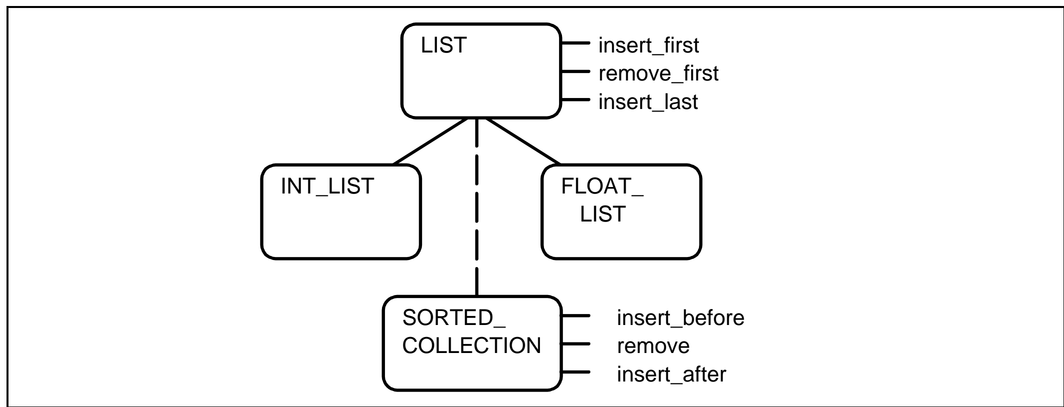

Figure 2.7: Private and Public Inheritance

In figure 2.7, INT_LIST and FLOAT_LIST inherit publicly from LIST; as such, they are type compatible with LIST and respect all of LIST's functions. SORTED_COLLECTION inherits privately from LIST, to use its linked cell representation, but does not export any of LIST's functions; rather it defines its own interface. An object of type SORTED_COLLECTION is not type compatible with a LIST variable.

Rather than contend with these apparently conflicting semantic demands, some have sought to drive a wedge between the notions of *class* and *type* [Snyd86a, Snyd86b]. In languages like *CommonObjects* [Snyd87] and *POOL-I* [Amer87, Amer90] you can reason about class and type independently. It is possible to design orthogonal class and type hierarchies, which maximise implementation reuse and type abstraction, respectively. Figure 2.8 illustrates this. On the left of the figure is a *type* hierarchy, in which a CIRCLE is considered a special case of ELLIPSE where the two foci are coincident; and a SQUARE is treated as a special case of RECTANGLE where adjacent sides are of the same length. On the right of the figure is a *class* (*ie* implementation) hierarchy, in which an ELLIPSE is constructed as an extension to the CIRCLE record with an extra focus and radius; likewise a RECTANGLE is an extension to the SQUARE record with an extra side. What is remarkable here is that the two hierarchies can be shown to link nodes in exactly the opposite order.

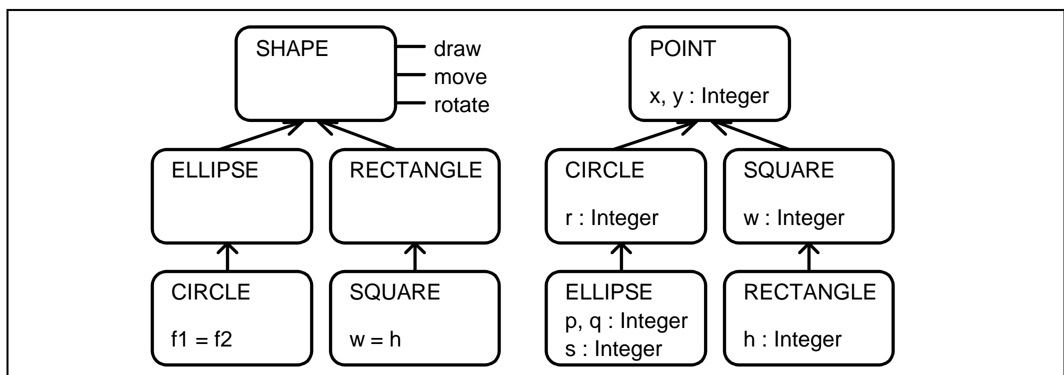

Figure 2.8: Orthogonal Class and Type Hierarchies

In a slightly different vein, the language *Emerald* [RL89, BHJL86] abandons the use of inheritance as a way of expressing implementation sharing, but retains a type hierarchy to capture the similarity and compatibility of software components. In *Emerald*, all implementation reuse is achieved through fine-grained composition, in which even the operations are treated as components with formal interfaces (requiring operands of the right type).

2.4.5 In Defence of Classification

Clearly, this segregation of *class* and *type* is a pragmatic success. But in focussing too closely on the implementation strategies adopted by a generation of *Smalltalk* programmers, have we somehow lost sight of the notion of *class* and *classification*? Perhaps it would be better to use a different term, such as *record* or *template* to describe the implementation units used above. It is a major contention of this thesis that the relegation of the *class* concept to such a mundane level constitutes a gross failure of nerve. The false division of *class* and *type* along concrete/abstract lines is a red herring. Types always were considered from both the abstract and the concrete points of view, so why not classes? Later chapters will develop a different view, which attempts to relate formally the notions of *class* and *type*, while preserving the possibility of attaching evolving implementations.

Consider in the first instance the philosophical argument, that classes and types aspire to the same goal, namely abstraction. Classification is a natural activity in psychology, which underpins our more mathematical notion of types. There is manifestly a strong desire to capture abstraction even in the type-free languages: *Smalltalk* [GR83] would appear to have advocated the "implementable abstraction" to a generation of programmers more accustomed to arrays, variable counters and (just emerging) record types. *Flavors'* [Moon86] introduction of multiple inheritance provided much stronger support for the factorisation of the common behaviour of objects; indeed one could argue that concept differentiation in AI is precisely the same task as that faced by designers of coerceable typing systems, a proposition recognised by the designers of *CLOS* [BDGK88] but not fully harmonised in its current specification.

A second philosophical argument is that type systems, as they are implemented in conventional strongly typed languages today, are far from complete in that they seldom venture to express systematic relationships between general, abstract types. To draw an ethical analogy, these languages are not so much pure, as fortunate that they have not been tempted. The *Algol-68* experiment [VMPK75] in implicit type coercion was a notable exception, but proved to be so theoretically difficult that subsequent languages ceded this hard-won territory. They cannot claim to hold the high moral ground; they have scarcely entered the battle. Any language that seeks to implement a truly general system of polymorphic and abstract types will face exactly the same difficulties as those encountered in class-based type factorisation in object-oriented programming.

It has been said that *inheritance* is the property that chiefly distinguishes the object-oriented languages [Wegn87]. This thesis disagrees with that emphasis.

The key concept is *classification*, something not attempted in other languages. Classification must carry with it the notion of hierarchy and an ordering on families of objects having different types. The kind of inheritance with polymorphism that occurs in object-oriented languages follows naturally from programming with classes, rather than with types. In later chapters, it will become clearer how this notion of *class* is more sublime than *type*.

In any case, chapters 3-5 will soon demonstrate that simple type hierarchies of the kind illustrated in figure 2.8 are insufficient to explain the type behaviour of object-oriented languages. Chapter 3 investigates object types and subtyping [Card84, CW85] as a first step in this direction; but later the focus shall return to polymorphic theories of classification.