outlines the domains of classes that you're likely to find in an application, and what the classes are in each domain. The chapter introduces a quantitative metric, the encumbrance of a class, and shows how it relates to class domains. The chapter uses the idea of domains and encumbrance to evaluate three kinds of class cohesion. (Class cohesion indicates how closely knit the attributes and operations of a given class are and whether they "belong together" in that class.)

Chapters 10 and 11, which explore the design fundamentals of object orientation, are at the heart of Part III. Chapter 10 introduces the ideas of state-space and the behavior of classes, along with the concepts of class invariants and operation preconditions/postconditions, which form the foundations of design by contract. Chapter 11 looks at the properties of subclasses in terms of state-spaces, together with the design principles of type conformance and closed behavior, which guide the development of robust class hierarchies.

In Chapter 12, I use concepts from the earlier chapters of Part III to identify and remove the dangers that lurk in some object-oriented constructs, including the prized mechanisms of inheritance and polymorphism. Chapter 13 dissects two actual designs to illustrate techniques for operation organization that can improve the resilience of class design. Chapter 14 further addresses the question of what constitutes good object-oriented design by inspecting the quality of a class's interface and by showing the criteria for a class to properly implement an abstract data-type.

8 Encapsulation and Connascence

This chapter covers the two fundamental properties of object-oriented system structure: encapsulation and connascence. Although both of these properties of software structure were present in traditional systems, object orientation, with its new complexities, elevates their significance considerably. The understandability and maintainability of object-oriented software—even the value of object orientation itself—rest fundamentally on encapsulation and connascence.

In the first section of this chapter, I discuss encapsulation; in the second, I discuss connascence and then explore how good object-oriented software depends on a combination of good encapsulation and good connascence.

8.1 Encapsulation Structure

As I mentioned in Chapter 1, software emerged in the 1940s from the primeval swamp as a collection of unicellular creatures known as machine instructions. Later, these evolved into other unicellular creatures known as lines of assembler code. But a grander structure soon appeared, in which many lines of code were gathered into a procedural unit with a single name. This was the subroutine (or procedure), with examples such as **computeLoanRepayment** and the all-time star of The Subroutine Hall of Fame, **computeSquareRoot**.

The subroutine introduced encapsulation to software. It was the encapsulation of lines of code into a structure one level higher than that of the code itself. The subroutine was a marvel of its time. As I mentioned in Chapter 1, it saved precious machine memory by hiving off dozens of instructions, and it saved human memory, too, by giving programmers a single term (such as **computeLoan-Repayment**) to refer to dozens of lines of code. No wonder the subroutine's advent induced boundless rapture among ecstatic programmers and gave rise to all-night coding parties.

8.1.1 Levels of encapsulation

I term the subroutine's level of encapsulation *level-1* encapsulation. (Raw code, with no encapsulation, has *level-0* encapsulation.) Object orientation introduces a further level of encapsulation. The class (or object) is a gathering together of subroutines (known as operations) into a yet higher-level structure. Since operations, being procedural units, are already at level-1 encapsulation, the class is at *level-2* encapsulation. See Fig. 8.1.

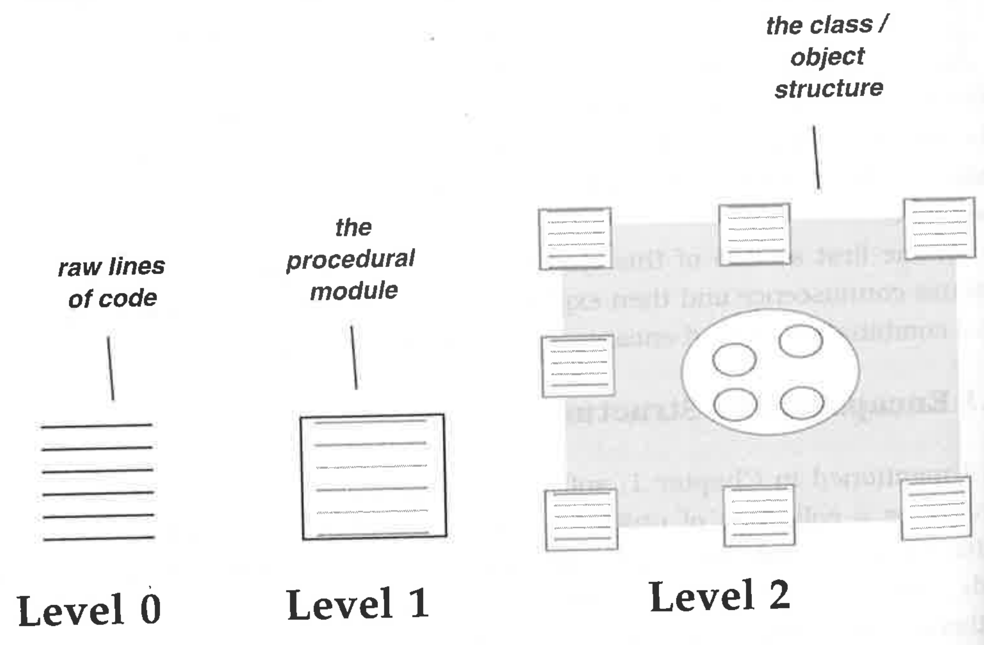

Fig. 8.1: Three levels of encapsulation exhibited by software constructs.

My analogy between organisms and software structures, though far from profound, isn't entirely gratuitous. Procedural modules, such as those found in structured design, didn't realize the reusability people expected.[1] Classes stand alone far better than procedures do. Classes come closer to biological organs in their ability to be transplanted from application to application. And components, which I discuss in Chapter 15, do even better than classes!

"But why stop at level-2 encapsulation?" you ask. A good question! We are already seeing level-3 and level-4 encapsulation, in which classes are grouped into even higher-level structures, such as the packages and components I discuss in Chapters 7 and 15. Outside level-3 and higher structures, only *some* classes (or parts of their interfaces) are made visible.

Business classes in a large object-oriented organization are often grouped "horizontally" into level-3 structures by respective subject area, much as we did in Chapter 7. For example, an airline might have a subject area related to the passenger, another for the airport, another for the personnel, and another for the aircraft. Since a class related to a passenger's frequent-flyer program wouldn't have much to do with a class for aircraft inventory, the two classes could be encapsulated in two different level-3 packages.

Level-3 structures can also be grouped "vertically."[2] For example, classes that would work together to implement the policy for **UpgradeFrequentFlyer** might include: **Passenger** (especially its **Preferences** and **FrequentFlyer** aspects), **Reservation**, **Leg** (of a **Flight**), **AirplaneSeatingConfiguration**, **Seat**, and lowlier classes such as **PrioritizedQueue**, **Date**, **Time**, and so on.

(By the way, a typical policy to move frequent fliers who've requested upgrades from economy to first class is triggered, say, 72 hours before departure. The system assigns each passenger wishing to upgrade a priority according to a combination of **FrequentFlyer.status**, **Reservation.dateMade**, **Reservation.fareBasis**, and so on. The highest-priority passengers are given upgrades, followed in descending order by the next-highest-priority passengers until the available seats are filled.)

[1] The Module of the Month Club offered an illustration of this. After joining this club, one could order the latest in tremendously reusable procedural modules. Unfortunately, the club soon went out of business.

[2] A *horizontal group* of classes comprises classes from the same domain that do not necessarily interact. (Packages typically exhibit this structure.) A *vertical group* of classes interact to implement a piece of business activity; they typically come from several domains. (Components normally exhibit this structure.) For more on which classes belong to which domains, see Section 9.1.

8.1.2 Design criteria governing interacting levels of encapsulation

Table 8.1 summarizes some traditional structured-design criteria in terms of the encapsulation levels of Section 8.1.1. It shows which criterion applies to each pair of encapsulation levels. For example, cohesion is a classic measurement of the quality of the relationship between a procedure (a level-1 construct) and the lines of code (level-0 constructs) within the procedure. I briefly describe each of the table's boxes in the paragraph and definitions that follow it.

Table 8.1.
The Structured-Design (or Level-1) Criteria Governing Interrelationships Among Elements at Each Pair of Encapsulation Levels.

TO: / FROM:	level-0 construct (line of code)	level-1 construct (procedure)
level-0 construct (line of code)	Structured programming	Fan-out
level-1 construct (procedure)	Cohesion	Coupling

The principles of structured programming govern the relationship between a line of code and other lines of code within the same procedure. Fan-out, cohesion, and coupling are terms from structured design.[3]

- *Fan-out* is a measure of the number of references to other procedures by lines of code within a given procedure.
- *Cohesion* is a measure of the "single-mindedness" of the lines of code within a given procedure in meeting the purpose of that procedure.
- *Coupling* is a measure of the number and strength of connections between procedures.

[3] See, for example, [Page-Jones, 1988] and [Yourdon and Constantine, 1979] for detailed discussions of fan-out, cohesion, and coupling.

Table 8.2 is an extension of Table 8.1 to include level-2 encapsulation. Notice that although the original (level-0 and level-1) section remains basically the same, level-2 encapsulation gives us five more boxes to name.

Class cohesion is an obvious analogue to the cohesion of a procedure, but it's at one level of encapsulation higher. It refers to the single-mindedness of a set of operations (and attributes) in meeting the purpose of the class. Class coupling is a measure of the number and strength of connections between classes.

Although the other three boxes don't have names, we could give them names (or we could dig out names from the mystic depths of the object-oriented literature). Then we'd have nine names. And, if we included level-3 encapsulation, we'd have sixteen names.

But enough names already! When the number of fundamental particles of physics exploded, physicists began to wonder whether their particles were quite so fundamental after all. When the number of fundamental design criteria explodes like this, perhaps we should look for a deeper criterion behind them all. If we can find such a criterion, then it should apply to software elements at all levels of encapsulation—even level-5, if we're ever blessed with such a level.

In the next section, I propose such a criterion: *connascence*.

Table 8.2.
An Extension of Table 8.1 to Include Level-2 Encapsulation (Classes).

TO: / FROM:	level-0 construct (line of code)	level-1 construct (operation)	level-2 construct (class)
level-0 construct (line of code)	Structured programming	Message fan-out	—
level-1 construct (operation)	Cohesion	Coupling	—
level-2 construct (class)	—	Class cohesion	Class coupling

8.2 Connascence

Connascence, which is derived from Latin, means "having been born together." An undertone to this meaning is "having intertwined destinies in life." Two software elements that are *connascent* (or that *connate)* are born from some related need—perhaps during requirements analysis, design, or programming—and share the same fate for at least one reason. Following is the definition of *connascence* as it applies to software:

> *Connascence* between two software elements **A** and **B** means either
>
> 1. that you can postulate some change to **A** that would require **B** to be changed (or at least carefully checked) in order to preserve overall correctness, or
>
> 2. that you can postulate some change that would require both **A** and **B** to be changed together in order to preserve overall correctness.

In this section, I explore the varieties of connascence. My chief purpose is to present the concept as a general way to evaluate design decisions in an object-oriented design, even if that design includes level-3 or level-4 encapsulation structures.

8.2.1 Varieties of connascence

I'll begin with a simple, non-O.O. example of connascence. Let's take software element **A** to be the single line of traditional code declaring

```
int i;          // line A
```

and element **B** to be the assignment:

```
i := 7;         // line B
```

There are at least two examples of connascence between **A** and **B**. For instance, in the (unlikely) situation that **A** were changed to **char i;** then **B** would certainly have to be changed, too. This is *connascence of type.* Also, if **A** were changed to **int j;** then **B** should be changed to **j := 7;**. This is *connascence of name.*

Now, some variations on the theme of connascence: The above example of **i** on lines **A** and **B** showed *explicit connascence,* which is in effect connascence that's detectable by a good text editor. In other words, it's connascence that leaps off the page and says, "This element is connascent with that one."

Some connascence, however, is *implicit.* For example, in an assembler routine I once saw

```
X: JUMP Y+38
...
Y: CLEAR R1
...                 // 38 bytes of code here
CLEAR R2            // This is the instruction being jumped to from X—call it Z
...
```

There are 38 bytes between **CLEAR R1** and **CLEAR R2**. Exactly 38 bytes! This *connascence of position* between these two innocent instructions at **Y** and **Z** is forced upon them by the nasty jump at line **X**. Although the need for this offset of precisely 38 bytes isn't apparent in the code after line **Y**, woe betide anyone who inserts another instruction somewhere in those 38 bytes.[4]

Clearly, explicitness and implicitness are neither binary nor absolute. Instead, connascence has a spectrum of explicitness. The more implicit connascence is, the more time-consuming and costly it is to detect (unless it's well documented in an obvious place). Connascence that spans huge textual distances in a class-library specification or other documentation is also likely to be time-consuming and difficult to discover.

Note that

1. Two software elements needn't communicate with each other in order to be connascent. (We saw an example of this in the connascence of position between lines **Y** and **Z** in the above assembler routine.)

[4] That, by the way, is exactly what a maintenance programmer did. The next time the system ran, it crashed shortly after line **X** was executed.

2. Some forms of connascence are *directional*. If element **A** refers to an element **B** explicitly, then **A** and **B** would be *unidirectionally connascent* (the direction being *from* **A** *to* **B**). Many examples of connascence of name are directional—for example, the connascence of name introduced when one class inherits from another. If element **B** also referred to element **A** explicitly, then **A** and **B** would be *bidirectionally connascent*.

3. Some forms of connascence are *nondirectional*. Elements **A** and **B** would be nondirectionally connascent if neither one referred explicitly to the other. For example, **A** and **B** are connascent if they use the same algorithm, although neither one refers to the other at all.

Most of the connascence I've described above is *static connascence*. That's connascence that applies to the code of the classes that you write, compile, and link. It's connascence that you can assess from the lexical structure of the code listing.

The following list (which isn't exhaustive) gives some further varieties of static connascence.

Connascence of name

We saw this in the first example (lines **A** and **B**) above, in which two programming variables needed to have the same name in order to refer to the same thing. Another example: A subclass that uses an inherited variable of its superclass must obviously use the same name for the variable that the superclass uses. If the name is changed in the implementation of the superclass, then it must also be changed in the subclass if correctness is to be preserved.

Connascence of type or class

We also saw connascence of type in the example with the data-type **int**. If **i** is assigned the value **7** on line **B**, then **i** should be declared to be of type **int** on line **A**.

Connascence of convention

Let's say that the class **AccountNumber** has instances in which positive account numbers, such as 12345, belong to people; negative ones, like –23456, belong to corporations; and 00000 belongs to all internal departments. The code will be sprinkled with statements like

```
if order.accountNumber > 0
then ...
```

There's a connascence of convention among all the software elements touching an account number. Unless this convention of **AccountNumber** meaning is encapsulated away, these elements may be widespread across the system.[5]

The hominoid of Chapter 1 furnishes us with another example. Let's say that **Hominoid** had the attribute **direction**. **direction** could be represented in many ways, for instance

0 = north; 1 = east; 2 = south; 3 = west
N = north; E = east; S = south; W = west
0 = north; 90 = east; 180 = south; 270 = west

Every client of **Hominoid** who uses the attribute will be exposed to the chosen convention for representing **direction**; this will create great connascence of convention. Thus, it is important to choose a decent convention (say, the third one) to represent **direction**.

Connascence of algorithm

Connascence of algorithm is similar to connascence of convention. Example: A software element inserts symbols into a hash-table. Another element searches for symbols in the table. Clearly, for this to work, they must both use the same hashing algorithm. Another example: the encoding and checking algorithms for check-digits in a customer's account number.

A bizarre example of connascence of algorithm that I saw recently was caused by a bug in the implementation of one of a class's operations. This operation was supposed to return an array of values, sorted into ascending

[5] This is actually a connascence of "value/meaning convention." It's similar to hybrid coupling in structured design and has caused many a maintenance problem in systems.

order. But, because of the bug, the last two array values were always switched. Owing to the absence of the source code, no one was able to correct the bug.

So the "fix" was this: Every class that invoked this operation had code added to it that switched back the two offending array values. This yielded horrid connascence of algorithm, which hurt everyone badly when the original defect was finally corrected. Then, all the code patches (in more than forty places) had to be found and removed.

Connascence of position

Most code in a procedural unit has connascence of position: For two lines of code to be carried out in the right execution sequence, they must appear in the right lexical sequence in the listing. There are several kinds of connascence of position, including *sequential* ("must appear in the correct order") and *adjacent* ("must appear next to each other"). Another example of connascence of position is the connascence in a message between the formal arguments in the sender and the actual arguments in the target. In most languages, you must set out actual arguments in the same sequence as the formal ones.

Dynamic connascence is connascence that's based on the execution pattern of the running code—the objects, rather than the classes, if you like. It, too, has several varieties:

Connascence of execution

Connascence of execution is the dynamic equivalent of connascence of position. It comes in several kinds, including *sequential* ("must be carried out in a given order") and *adjacent* ("must be carried out with no intervening execution"). There are many examples of connascence of execution, including initializing a variable before using it, changing and reading the values of global variables in the correct sequence, and setting and testing semaphore values.

Connascence of timing

Temporal connascence crops up most often in real-time systems. For example, an instruction to turn off an X-ray machine must be executed within *n* milliseconds of the instruction to turn it on. This timing constraint remains true no matter how much the operating system needs to preempt the **xRayController** task in order to carry out the workload of its other tasks.

Connascence of value

Connascence of value usually involves some arithmetic constraint. For example, during the execution of a system, the **lowPointer** to a circular buffer can never be higher than the **highPointer** (under the rules of modulo arithmetic). Also, the four corners of a rectangle must preserve a certain geometric relationship in their values: You can't move just one corner and retain a correct rectangle.

Connascence of value is notorious for occurring when two databases hold the same information redundantly, often in different formats. In that situation, procedural software has to maintain a bridge of consistency between the databases to ensure that any duplicated data have identical values in each database. The maintenance of this software, which is probably performing awkward format translations, may be cumbersome.

Connascence of identity

An example of connascence of identity is provided by a typical constraint in an object-oriented system: Two objects, **obj1** and **obj2**, each of which has a variable pointing to another object, must always point to the same object. That is, if **obj1** points to **obj3**, then **obj2** must point to **obj3**. (For example, if the sales report points to the March spreadsheet, then the operations report must also point to the March spreadsheet.) In this situation, **obj1** and **obj2** have connascence of identity; they must both point to the same (that is, identical) object.

8.2.2 Contranascence

So far, I've tacitly equated connascence with "sameness" or "relatedness." For example, two lines of code have connascence of name when the variable in each of them must bear the same name. However, connascence also exists in cases where *difference* is important.

First, I'll offer a trivial example. Let's say that we have two declarations:

```
int i;
int j;
```

For correctness—indeed, merely for the code to compile!—the variable names, **i** and **j**, must differ from each other. There's a connascence at work here: If, for some reason, we wanted to change the first variable name to **j**, then we'd also have to change the second name *from* **j** to something else. Thus, the two declarations are not independent.

I've heard this kind of connascence called "connascence of difference" or "negative connascence." I use the shorter term *contranascence.* Although it sounds like the opposite of connascence, contranascence is actually a form of connascence in which difference, rather than equality, must be preserved.[6]

A familiar case of contranascence crops up in object-oriented environments with multiple inheritance (the ability of a subclass to inherit from multiple superclasses). If class **C** inherits from both classes **A** and **B**, then the features of **A** and **B** should not have the same names: There's a contranascence of name between **A**'s features and **B**'s features.

As a more concrete example of contranascence, consider an application in a video-rental store.

The class **ProgramRentalItem** may inherit from both **PhysicalInventoryItem** and **RecordingMedium**. These two classes (or their superclasses) may each have an attribute named **length**. But **PhysicalInventoryItem**'s **length** may mean the physical length of an item in inches, and **RecordingMedium**'s **length** may mean the playing time of the program (the movie or whatever is on the video tape).

Although you could argue that **duration** would be a more appropriate name for the second attribute, you may have to take what you get from your class library. In our case, the class **ProgramRentalItem** needs to inherit both of the attributes named **length**, and this clash in names brings a serious problem.

6 The highfalutin term for absence of connascence is *disnascence:* Two software elements are disnascent if one has absolutely nothing to do with the other. Of course, *independence* works pretty well, too!

Across an entire library of classes, of which any pair may share a subclass, there's a contranascence of name across *all* classes because of this risk of name clashes under multiple inheritance. No wonder multiple inheritance has acquired a bad reputation, and no wonder good object-oriented languages contain mechanisms that remove this rampant contranascence.[7]

8.2.3 Connascence and encapsulation boundaries

I'll go so far as to say that connascence and contranascence are at the heart of modern software-engineering constructs.

To explain this, I'd like to return to Section 8.1's topic, encapsulation. Although I defined encapsulation and the levels to which it may aspire, I didn't say much about why encapsulation is important.

Encapsulation is a check on connascence, especially contranascence. Imagine a system comprising 100,000 lines of code. Imagine further that the entire system resides in a single module, say a main procedure. Imagine next that you have to develop and maintain this system. The contranascence among the hundreds of variable names would of itself be a nightmare: Simply to pick a name for a new variable, you'd first have to check dozens of other names to be sure of avoiding a clash.

Another problem would be the deceptive nature of the code. Consider two lines of code that are adjacent on a source listing. You might wonder *why* the two lines are adjacent: Is it because they *must be* adjacent (owing to connascence of position) and that inserting a line would wreck the system? Or did they just *happen to* wind up adjacent when the code was all shoveled into the same module?

So, a system that's not broken into encapsulated units has two problems: rampant connascence (chiefly through contranascence), and the confusion over what is *true* connascence and what is accidental similarity or adjacency. (An example of accidental similarity would be two variables named **i**, in two entirely separate and unconnected classes. There would be no connascence of name here; either variable could be renamed **j** with no sad consequences.)

Connascence is also why object orientation "works." Object orientation eliminates—or at least tames—some of the connascence that runs wild in traditional modular systems with only level-1 encapsulation. Again, I'll explain with an example: the hash-table example of Section 8.2.1.

7 One of the best language mechanisms is Eiffel's **rename** keyword. See [Meyer, 1992], for example. See also Exercise 5 in Chapter 12 for another look at this example.

I'll assume a system that maintains a single hash-table and is designed with only level-1 encapsulation (which is the level of encapsulation of, say, structured design). The system must access the hash-table from several (at least two) places in the code: location(s) that update the table and location(s) that look up symbols in the table. The code in these locations will have connascence of algorithm; if you think up a better hashing algorithm, then you'll have to find all the code locations that use the current algorithm and make the necessary changes.

Level-1 encapsulation will neither guide you to where these places are nor tell you how many places there are. Even if there are only two places with the hashing algorithm, they may be close together or far apart in the system listing. You'll be on your own (unless you can find some friendly and accurate documentation).

An object-oriented system, with at least level-2 encapsulation, has a natural home for the hashing algorithm in the single class **SymbolTable**. Although there will still be a connascence of algorithm between the operation **insertSymbol** and the operation **lookupSymbol**, the connascence will be under control. It will be encapsulated within the boundary of a single element (the class **SymbolTable**). If good object-oriented design has been used, then there will be no connascence due to the hashing algorithm anywhere else in the system (that is, anywhere outside **SymbolTable**).

8.2.4 Connascence and maintainability

Connascence offers three guidelines for improving system maintainability:

1. Minimize overall connascence—this includes contranascence, of course—by breaking the system into encapsulated elements.

2. Minimize any remaining connascence that crosses encapsulation boundaries. (Guideline 3, below, will help here.)

3. Maximize the connascence within encapsulation boundaries.

The above guidelines transcend object orientation. They apply to any software-construction approach with level-2 encapsulation or level-3 encapsulation, or an even higher level. Also, as you may have noticed, the guidelines express a very old principle in design: Keep like things together and unlike things apart. However, this old principle never told us what "like things" were; in fact, they're software elements with mutual connascence.

Figure 8.2 shows two classes with connascence between them. (The connascence is directional, with the direction indicated by arrowheads.) Some of this connascence (shown by the broad lines) violates object-oriented principles by connecting the internal design of one class to the internal design of another: "Like things" have been placed in different software structures.

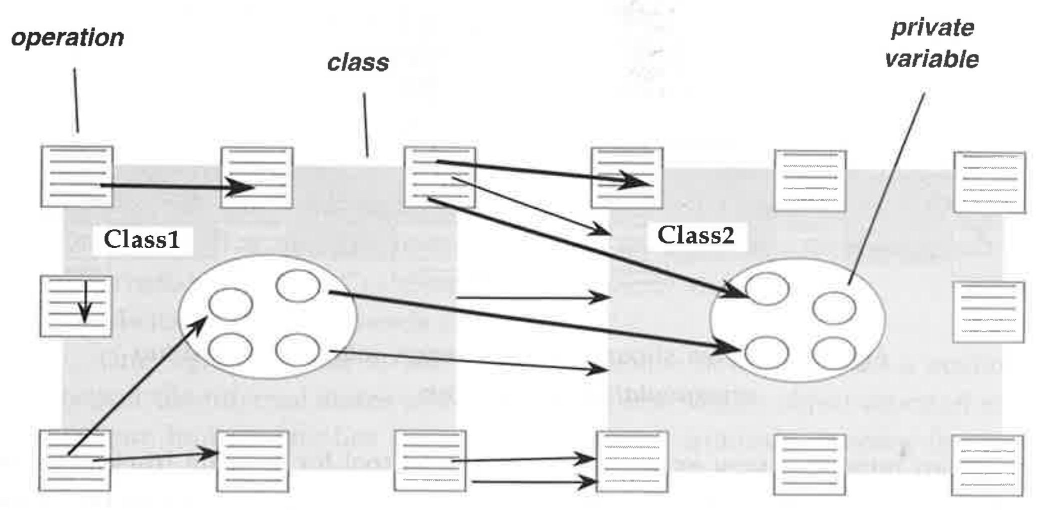

Fig. 8.2: Lines showing connascence (for example, connascence of name), with broad lines violating encapsulation boundaries.

For example, a line from a method of **Class1** that refers to a variable within **Class2** violates object-oriented encapsulation. (Note that directional connascence violates encapsulation only when it crosses *into* an encapsulated unit.)

Figure 8.3 shows two other classes that have no offending encapsulation-busting connascence.

Connascence represents a set of interdependencies in software. Explicit connascence is apparent in the source code and can often be discovered with little more than a text editor's search or a cross-reference listing. Implicit connascence (connascence not readily apparent from the code itself) may be detectable only by human ingenuity, aided by whatever documentation exists for the system under examination.

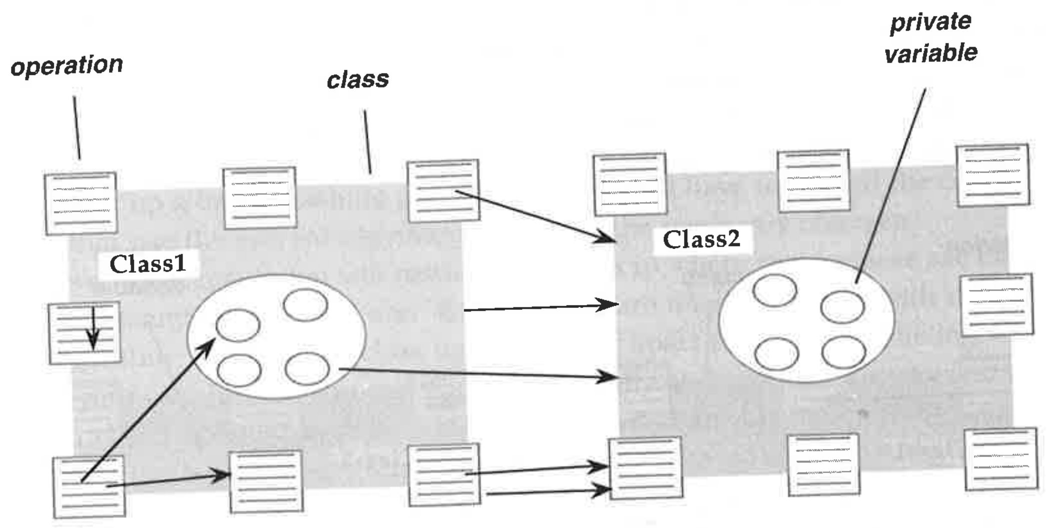

Fig. 8.3: Lines showing connascence, with none violating encapsulation boundaries.

The human mind is a very expensive (and fallible) tool for keeping track of widespread connascence. Implicit connascence, especially when it straddles encapsulation boundaries, presents an extra challenge to system maintainers. Implicit connascence becomes particularly difficult to track several months after the code has been designed and written. Perhaps future CASE tools will assist people in monitoring connascence and in revealing implicit connascence. This would be especially helpful in large systems that exhibit level-2 (or higher) encapsulation.

8.2.5 Connascence abuses in object-oriented systems

As we saw in Section 8.2.3, the level-2 encapsulation capabilities of object orientation are a tremendous boon for a designer trying to tame connascence. However, in this final section on connascence, I give three examples of how object-oriented designers sometimes violate the principle of "keeping connascence at home"—that is, within class boundaries. The first concerns the friend function of C++, the second concerns a misuse of inheritance, and the third concerns a gratuitous introduction of connascence in breach of object-oriented principles.

1. The friend function of C++

The friend function of C++ was created expressly to violate encapsulation boundaries. It's an element outside the boundaries of a class that has access to the private elements of objects of that class. So, if a friend function **ff** gets the handle of an object of class **C1**, it can mess about in the internals of that object willy-nilly. The connascence between **ff** and **C1** is therefore high; it includes connascence of name, type, convention, and so on. Every change that a designer makes to the internal design of **C1** will require **ff** to be thoroughly checked and possibly changed.

If **ff** is the friend of only one class, then you could argue that **ff** is really part of that class and to all intents lies within its boundary. Fair enough. However, if **ff** is also the friend of **C2**, **C3**, and **C4**, then that argument fails. Unfortunately, many C++ designs have exactly that structure . . . and with friends like those, who needs enemies!

One legitimate use of the friend construct, however, is as a scaffold to inspect the internal states of objects under test. Since object-oriented encapsulation limits white-box scrutiny of objects, it ironically hinders the testing of object-oriented systems. The friend function lifts the veil of secrecy from the implementation of the class under test.

2. Unconstrained inheritance

The construct of inheritance—although wildly popular in most object-oriented shops—may sometimes introduce raging connascence. If you allow a class to make use of both externally visible and internally visible elements of a superclass, then you will introduce a great deal of connascence across major (class) encapsulation boundaries. This will include connascence of name, connascence of class, and other varieties of connascence.

In a shop where I once consulted, an analysis/design team had begun to build its class library with some very well-thought-out hierarchies of Smalltalk classes. For example, **DomesticShipment** and **ExportShipment** were both (reasonably enough) subclasses of **Shipment**. However, the team's language and design strategy allowed subclasses unbridled access to programming variables within superclasses. This meant that the maintainers of subclasses **C1**, **C2**, and so forth, had to stay aware—almost on a minute-to-minute basis—of any changes to the internal design of the superclass **C**, because **C**'s maintainer could create disastrous results in descendant classes just by making an innocuous change to an (ostensibly private) variable name.

The team worked around this problem by having Jim be responsible for, say, **DomesticShipment**, **ExportShipment**, **Shipment**, and other related classes. Jim would make sure that any changes he made to **Shipment** were propagated to classes connascent with **Shipment**—and there were actually many such classes. Unfortunately, however, the rampant connascence among the classes couldn't be parceled out to individual team members like the classes themselves could. The result was that every class maintainer had the tedious chore of keeping abreast of internal design changes to all that class's superclasses. Thus every change to the class library caused high anxiety all round, since time pressures denied the team any chance to keep the library documentation up to date.

If Jim and his colleagues had taken into account the guideline that connascence should not cross encapsulation boundaries, it would have told them that inheritance by a subclass should be restricted to only those features of the superclass that are already externally visible. (Another way to say this is: The notion of inheriting abstract behavior should be divorced from the notion of inheriting the internal implementation of such behavior.[8]) Had their library been designed according to this principle, their lives would have been less fraught with trouble, and maybe they wouldn't have called their shop The Land of the Midnight Fix.

3. Relying on accidents of implementation

In another shop of yore, a programmer named The Weasel—don't ask me why!—had created a class **Set** that furnished the behavior of a mathematical set. (Its operations included **add**, **remove**, **size**, and so on.) The class appeared to be well designed and written and it always performed fine.

The Weasel used **Set** in several places in his own applications. For example, he used the **retrieve** operation, which retrieved elements of the set one by one in random order until every element had been provided. However, The Weasel knew that **retrieve** happened to retrieve elements in exactly the *same* order in which they'd been added to the set, even though that wasn't documented or supposed to be a property of the operation. He made use of this accidental, undocumented fact several times in his application. He had thus

[8] See [Porter, 1992] for a further discussion of this point.

created a connascence of algorithm across the encapsulation boundary between his application and the internals of the **retrieve** operation.

When **Set** was later replaced by a different implementation—one that happened not to preserve order in the same way—many of The Weasel's applications blew up. Many of the users also blew up, but The Weasel was nowhere to be found. It was rumored that he had taken up another identity and had gone underground in the Middle East. So, let's all hope that The Day of The Weasel has passed forever.

8.2.6 The term connascence

Although I didn't make up the word "connascence"—it's in *Chambers' Twentieth Century Dictionary*, for example, and in *Webster's Third New International Dictionary* (as "connate")—I have taken a lot of flak over the past few years for using it. I carried out some market research on *coupling* as an alternative term, but at one shop, the developers huddled together upon hearing the word and a muttering arose in their midst. Then, turning to me, they spake as with one voice, saying: "Hey, don't waste our time with that ancient structured crap!"

I tried *interdependence* at another shop, but apparently that term was too bland to register a single blip on the team's cognitive radar. That's why I like *connascence:* It's a term that gets people's attention and doesn't need to skirt around any prior usage.

Once, however, I got burned. A developer at one shop said, "Hey, I've used connascence for years with a different meaning. I say that two objects are connascent when they get instantiated together at run-time." Since I could hardly argue with his use of the term as meaning "born together," I told him he could substitute the humbler term *interdependence* for my more general notion.[9] You can do that, too, if you like—I'm by no means wedded to *connascence*.

[9] If two objects *had to* be instantiated together, then I'd say that the two objects had *connascence of instantiation*. If two objects *had to* have the same lifetime, I'd say they had *connascence of lifetime*—or if I wanted to impress people, *connascence of duration*.

8.3 Summary

Encapsulation is a venerable concept in software. The subroutine, invented in the 1940s, introduced encapsulation of code into procedural modules; this is level-1 encapsulation. However, object-oriented structures are more sophisticated than traditional procedural structures like the subroutine. Object orientation involves at least level-2 encapsulation. In level-2 encapsulation, operations (implemented by methods) are themselves encapsulated, together with attributes (implemented by variables), into classes.

The complexities of level-2 encapsulation introduce many novel interdependencies among design elements. Rather than give each kind of interdependency its own term, I introduce the general term *connascence*. Connascence exists when two software elements must be changed together in some circumstance in order to preserve software correctness. Contranascence is a form of connascence in which difference, rather than similarity, must be preserved. Disnascence is the absence of connascence.

Connascence comes in several forms. Static connascence derives from the lexical structure of a code listing. Examples include connascence of class and connascence of convention. Dynamic connascence depends on the execution pattern of code at run-time. Examples include connascence of timing and connascence of value. Explicit connascence is immediately apparent from reading a code listing. An example is connascence of name. Implicit connascence is apparent only from a study of the code or its attendant documentation. Connascence of execution or algorithm is usually implicit. Copious implicit connascence raises software-maintenance costs.

The level-2 encapsulation of object orientation addresses the problem of potentially rampant connascence in large, modern systems. This encapsulation provides solid class structures within whose boundaries unbridled connascence can be corralled.

However, there are several ways that connascence may escape encapsulation boundaries—even in an object-oriented design. In this chapter, we saw three examples of poor design: The first was the use of C++'s friend function deliberately to nullify the benefits of object-oriented encapsulation. The second was the misguided use of inheritance to allow a subclass to inherit the implementation of a superclass. The third was allowing the internal (and probably volatile) details of a class's algorithm to be relied on by code in other classes.

The three examples above violate the central principle of object-oriented design: Minimize overall connascence—this includes contranascence, of course—by breaking the system into encapsulated elements. Then minimize any remaining connascence that crosses encapsulation boundaries by maximizing the connascence within encapsulation boundaries.

8.4 Exercises

1. A man I met in a pub told me that every idea in modern music can be found somewhere in Haydn's works. One could say something similar about Yourdon and Constantine's book on structured design [Yourdon and Constantine, 1979]: Its pages form a magnum opus of oft-overlooked design ideas, which are "rediscovered" time after time. Check this book to see whether Yourdon and Constantine have anything to say on the subject of connascence.

2. Use of the **goto** statement has become notorious over the past few decades as a cause of incomprehensible software. From the standpoint of connascence, can you justify the **goto**'s bad reputation?

3. Assume that you've become tired of object orientation. You want to create a new software paradigm that employs levels of encapsulation and has various forms of connascence. Explain (in a general way) how you would set forth the design criteria and guidelines for connascence and encapsulation in your paradigm.

4. This chapter discussed connascence mainly in terms of programming code. Are there any other examples of connascence that crop up in the wider context of the overall software-development project?

5. In Section 8.2.4, I suggested that implicit connascence that crosses encapsulation boundaries usually proves particularly troublesome to maintainers of an object-oriented system. Can you give examples of such connascence and suggest how to make it more explicit, and thus easier to track?

6. Further research is needed into the forms of connascence that apply to modern software-design paradigms, especially as object-oriented design becomes popular. Set up an experiment to elicit the degrees of ill caused by the different varieties of connascence (for example, connascence of name and position). The experiment could measure the effects of connascence (both within and across encapsulation boundaries) on presumed dependent factors, such as human comprehension time/cost, debugging time/cost, and modification time/cost. Perhaps you can recruit volunteers to review source code and measure their time to detect bugs deliberately seeded into the code.

8.5 Answers

1. Yes, Yourdon and Constantine begin an exploration of what is essentially connascence in Chapter 3 of their book. However, after introducing connascence (under the vague term *structure)* and touching on it briefly, the chapter goes off at an angle from the general concept. Later, especially in Chapter 6, the book flirts tantalizingly with the topic again.

2. For a long time, we've known that the undisciplined use of the **goto** statement causes the static and dynamic structures of code to diverge. In terms of connascence, it implies that static connascence of position (in the code listing) gives little clue to dynamic connascence of execution (at run-time). Since maintainers make modifications to the static code, **goto**s increase the risk that a static change will violate some connascence of execution. Furthermore, any **goto** induces an additional connascence of name between the **goto** itself and the label that's the target of the **goto**, together with contranascence among the label names themselves.

3. Here's one possible framework on which to base a definition of a future software paradigm:

   a. State the intended purpose and scope of applicability of your paradigm.

   b. State the paradigm's encapsulation structure. State what the paradigm's components are and which components are contained within which.

   c. In terms of the above encapsulation structure, state the default visibility rules of the paradigm. This will prescribe the allowed connections among components and will state the "boundaries of privacy" established by the encapsulation structure.

   d. List the possible forms of connascence inherent in the paradigm. There will be explicit connascence, which appears in the source code, and implicit connascence, which will be more difficult to perceive because it is "invisible." As we saw, implicit connascence becomes especially subtle when it transcends the official encapsulation structure of the paradigm.

e. Classify as much as possible the pernicious effects of each form of connascence in various contexts.

f. Suggest heuristics for deriving or modifying software designed under this paradigm in order to minimize the pernicious effects mentioned above.

4. Yes, there are many examples of connascence that spans project deliverables. For instance, you can find connascence between the model of user requirements and the design model of a software implementation of those requirements. Recently, I encountered a distressingly vast connascence of name when a business decided to change the word **Customer** to **Client**. The havoc this caused was immense! A good "full project life-cycle" modeling tool could keep track of these hundreds of lines of connascence across deliverables and thereby reduce the onus on the human mind to keep track of them.

5. I recently saw two examples of implicit connascence in two separate object-oriented systems.

   In the first example, a pair of classes in a business system each contained the number **5** (representing the number of office buildings the company owned). This created implicit connascence of value between the two classes, because to change one **5** (but not the other) to **6** would cause an error. If the literal constant **5** were instead denoted by **numOfOffices**, then the connascence would become explicit and less of a problem: Change the value once, and it would be changed everywhere. To do this, you would presumably store the value of **numOfOffices** in a database.

   In the second example, a real-time communications application contained two objects that would issue voluminous communications across a network at exactly the same time. This caused the network to clog unnecessarily, because there was no reason that the objects *had to* start communicating simultaneously. The problem was solved when the two objects were made to stagger their transmissions. In other words, this system had an implicit contranascence of timing, which was made explicit by setting up a scheduling table for the objects' communication. (Notice that in this example, preserving the contranascence of timing is necessary for performance, rather than for strict correctness of the software.)

9 Domains, Encumbrance, and Cohesion

Classes in a system are not all alike. For example, in a brokerage application you might find the classes **Equity**, **AccountPosition**, **Date**, **Time**, **List**, and **Set**. In an avionics system, you might find the classes **Flap**, **FuelTank**, **Date**, **Time**, **Set**, and **Tree**.

Notice that there's something about the classes **Equity** and **FuelTank** that sets them apart from the classes **Date** and **Set**. For example, **Equity** seems more complex, intricate, and specialized than the simple **Date** class. That's because **Equity** and **FuelTank** are from the business domain, whereas **Date** and **Set** are from the foundation domain.[1] Moreover, the classes **Equity** and **FuelTank** seem different from each other, as well, because they're from two different industries (brokerage and aviation).

I begin this chapter by defining domains of classes. In the chapter's second section, I introduce encumbrance as a quantitative measure of a class's "sophistication" and show how classes from higher domains normally have higher values

[1] Please note that here I mean *business* in the widest possible sense. I include businesses involving avionics, instrumentation, and microwave-oven control, along with the more traditional businesses of banking, insurance, and fishmongery.