Lecture 2: Decoupling I

A central issue in designing software is how to decompose a program into parts. In this lecture, we'll introduce some fundamental notions for talking about parts and how they relate to one another. Our focus will be on identifying the problem of *coupling* between parts, and showing how coupling can be reduced. In the next lecture, we'll see how Java explicitly supports techniques for decoupling.

A key idea that we'll introduce today is that of a *specification*. Don't think that specifications are just boring documentation. On the contrary, they are essential to decoupling and thus to high-quality design. And we'll see that in more advanced designs, specifications become design elements in their own right.

Our course text treats the terms *uses* and *depends* as synonyms. In this lecture, we'll distinguish the two, and explain how the notion of *depends* is a more useful one than the older notion of *uses*. You'll need to understand how to construct and analyze dependency diagrams; uses diagrams are explained just as a stepping stone along the way.

2.1 Decomposition

A program is built from a collection of parts. What parts should there be, and how should they be related? This is the problem of *decomposition*.

2.1.1 Why Decompose?

Dijkstra pointed out that if a program has N parts, and each has a probability of correctness of c – that is, there's a chance of 1-c that the developer gets it wrong – then the probability that the whole assemblage will work is c^N. If N is large, then unless c is very close to one, c^N will be near zero. Dijkstra made this argument to show how much getting it right matters – and the bigger the program gets, the more it matters. If you can't make each part almost perfect, you have no hope of getting the program to work.

(You can find the argument in the classic text *Structured Programming* by Dahl, Dijkstra and Hoare, Academic Press, 1972. It's a seductive and elegant argument, but perhaps a bit misleading. In practice, the probability of getting the whole program completely correct is zero anyway. And what matters is ensuring that certain limited, but crucial, properties hold, and these may not involve every part. We'll return to this later.)

But doesn't this suggest that we shouldn't break a program into parts? The smaller *N* is, the higher the probability that the program will work. Of course, I'm joking – it's easier to get a small part right than a big one (so the parameter *c* is not independent of *N*). But it's worth asking what benefits come from dividing a program into smaller parts. Here are some:

- *Division of labour.* A program doesn't just appear out of thin air: it has to be built gradually. If you divide it into parts, you can get it built more quickly by having different people work on different parts.
- *Reuse.* Sometimes it's possible to factor out parts that different programs have in common, so they can be produced once and used many times.
- *Modular Analysis.* Even if a program is built by only one person, there's an advantage to buidling it in small parts. Each time a part is complete, it can be analyzed for correctness (by reading the code, by testing it, or by more sophisticated methods that we'll talk about later). If it works, it can be used by another part without revisiting it. Aside from giving a satisfying sense of progress, this has a more subtle advantage. Analyzing a part that is twice is big is much more than twice as hard, so analyzing about a program in small parts dramatically reduces the overall cost of the analysis.
- *Localized Change.* Any useful program will need adaptation and extension over its lifetime. If a change can be localized to a few parts, a much smaller portion of the program as a whole needs to be considered when making and validating the change.

Herb Simon made an intriguing argument for why structures – whether man-made or natural – tend to be build in a hierarchy of parts. He imagines two watchmakers, one of whom builds watches in one go, in a single large assembly, and one of who builds composite subassemblies that he then puts together. Whenever the phone rings, a watchmaker must stop and put down what he as currently working on, spoiling that assembly. The watchmaker who builds in one go keeps spoiling whole watch assemblies, and must start again from scratch. But the watchmaker who builds hierarchically doesn't lose the work he did on the completed subassemblies that he was using. So he tends to lose less work each time, and produces watches more efficiently. How do you think this argument applies to software?

(You can find this argument in Simon's paper *The Architecture of Complexity*.)

2.1.2 What Are The Parts?

What are the parts that a program is divided into? We'll use the term 'part' rather than 'module' for now so we can keep away from programming-language specific notions. (In the next lecture, we'll look at how Java in particular supports decomposition into parts). For now, all we need to note is that the parts in a program are *descriptions*: in fact, software development is really all about producing, analyzing and executing descriptions. We'll soon see that the parts of a program aren't all executable code – it's useful to think of specifications as parts too.

2.1.3 Top Down Design

Suppose we need some part *A* and we want to decompose into parts. How do we make the right decomposition? This topic is a large part of what we'll be studying in this course. Suppose we decompose *A* into *B* and *C*. Then, at the very least, it should be possible to build *B* and *C*, and putting *B* and *C* together should give us *A*.

In the 1970's, there was a popular approach to software development called *Top-down Design*. The idea is simply to apply the following step recursively:

- If the part you need to build is already available (for example, as a machine instruction), then you're done;
- Otherwise, split it into subparts, develop them, and combine them together.

The splitting into subparts was done using 'functional decomposition'. You think about what function the part should have, and break that function into smaller steps. For example, a browser takes user commands, gets web pages, and displays them. So we might split *Browser* into *ReadCommand*, *GetPage*, *DisplayPage*.

The idea was appealing, and there are still people who talk about it with approval. But it fails miserably, and here's why. The very first decomposition is the most vital one, and yet you don't discover whether it was good until you reach the leaves of the decomposition tree. You can't do much evaluation along the way; you can't test a decomposition into two parts that haven't themselves been implemented. Once you get to the bottom, it's too late to do anything about the decompositions you made at the top. So from the point of view of risk – making decisions when you have the information you need, and minimizing the chance and cost of mistakes – it's a very bad strategy.

In practice, what usually happens is that the decomposition is a vague one, with the hope that the parts become more clearly defined as you go down. So you're actually figuring out what problem you're trying to solve as you're structuring the solution. As a result, when you get near the bottom, you find yourself adding all kinds of hacks to make the parts fit together and achieve the desired function. The parts become extensively *coupled* to one another, so that the slightest alteration to one isn't possible without changing all the others. If you're unlucky, the parts don't fit together at all. And, finally, there's nothing in top-down design that encourages reuse.

(For a discussion of the perils of top-down design, see the article with that title in: *Software Requirements and Specifications: A Lexicon of Software Principles, Practices and Prejudices*, Michael Jackson, Addison Wesley, 1995.)

This isn't to say, of course, that viewing a system hierarchically is a bad idea. It's just not possible to develop it that way.

2.1.4 A Better Strategy

A much better strategy is to develop a system structure considering of multiple parts at a roughly equal level of abstraction. You refine the description of every part at once, and analyze whether the parts will fit together and achieve the desired function before starting to implement any of them. It also turns out that it is much better to organize a system around data than around functions.

Perhaps the most important consideration in evaluating the decomposition into parts is how the parts are coupled to one another. We want to minimize coupling – to *decouple* the parts – so that we can work on each part independently of the others. This is the topic of our lecture today; later in the course, we'll see how we can express properties of the parts and the details of how they interact with one another.

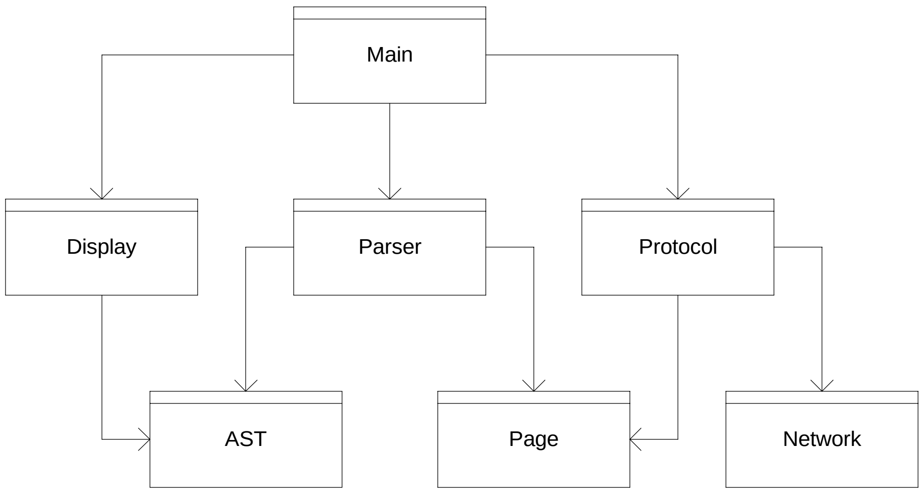
2.2 Dependence Relationships

2.2.1 Uses Diagram

The most basic notion relationship between parts is the *uses* relationship. We say that a part A uses a part B if A refers to B in such a way that the meaning of A depends on the meaning of B. When A and B are executable code, the meaning of A is its behaviour when executed, so A uses B when the behaviour of A depends on the behaviour of B.

Suppose, for example, we're designing a web browser. The diagram shows a putative decomposition into parts:

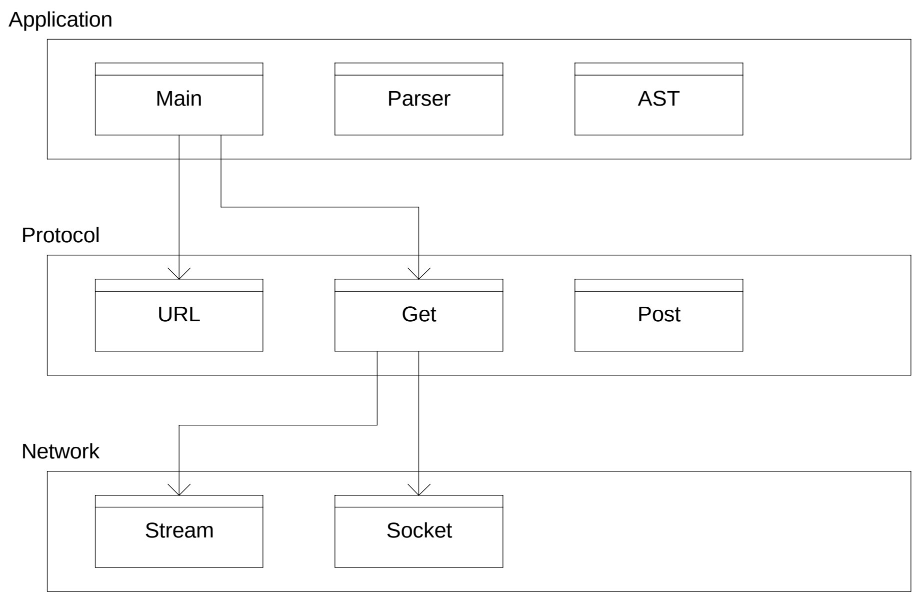
The *Main* part uses the *Protocol* part to engage in the HTTP protocol, the *Parse* part to parse the HTML page received, and the *Display* part to display it on the screen. These parts in turn use other parts. *Protocol* uses *Network* to make the network connection and to handle the low-level communication, and *Page* to store the HTML page



received. *Parser* uses the part *AST* to create an abstract syntax tree from the HTML page – a data structure that represents the page as a logical structure rather than as a sequence of characters. *Parser* also uses *Page* since it must be able to access the raw HTML character sequence. *Display* uses a part *Render* to render the abstract syntax tree on the screen.

Let's consider what kind of shape a uses graph may take.

- *Trees.* First, note that when viewed as a graph, the uses-diagram is not generally a tree. Reuse causes a part to have multiple users. And whenever a part is decomposed into two parts, it is likely that those parts will share a common part that enables them to communicate. *AST*, for example, allows *Parser* to communicate its results to *Display*.
- *Layers.* Layered organizations are common. A more detailed uses-diagram of our browser may have several parts in place of each of the parts we showed above. The *Network* part, for example, might be replaced by *Stream*, *Socket*, etc. It is sometimes useful to think of a system as a sequence of layers, each providing a coherent view of some underlying infrastructure, at varying levels of abstraction. The *Network* layer provides a low-level view of the network; the *Protocol* layer is built on top of it, and provides a view of the network as an infrastructure for processing HTTP queries, and the top layer provides the application user's view of the system, which turns URLs into visible web pages. Technically, we can make any uses-diagram layered by assigning each part to a layer so that no *uses* arrow points from a part in

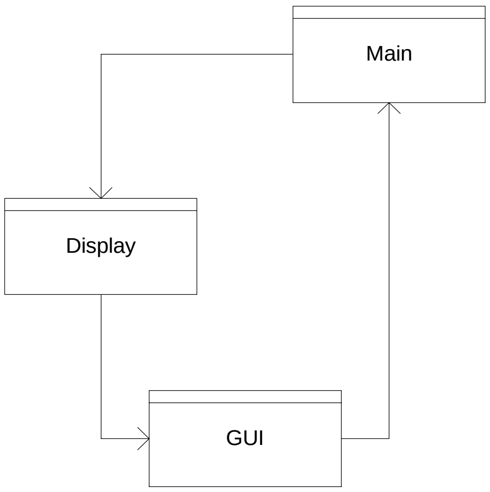


some layer to a part in a higher layer. But this doesn't really make the program layered, since the layers have no conceptual coherence.

- *Cycles*. It's quite common to have cycles in the uses-diagram. This doesn't have to mean that there is recursion in the program. Here's how it might arise in our browser design. We haven't considered how *Display* will work. Suppose we have a *GUI* part that provides functions for writing to a display, and handles input by making calls (when buttons are pressed, etc) to functions in other parts. Then *Display* may uses *GUI* for output, and *GUI* may use *Main* for input. In object-oriented designs, as we'll see, cycles often arises when objects of different classses interact strongly.

What can we do with the uses-diagram?

- *Reasoning*. Suppose we want to determine whether a part *P* is correct. Aside from *P* itself, which parts do we need to examine? The answer is: the parts *P* uses, the parts they use, and so on – in other words all parts reachable from *P*. In our browser example, to check that *Display* works we'll need to lok at *Render* and *AST* also. Conversely, if we make a change to *P*, which parts might be affected? The answer is all parts that use *P*, the parts that use them, and so on. If we change *AST* for example, *Display*, *Parser* and *Main* may all have to change. This is called *impact analysis* and it's important during maintenance of a large program when you want to make sure that the consequences of a change are completely known, and you want to avoid retesting every part.



- *Reuse.* To identify a subsystem – a collection of parts – that can be reused, we have to check that none of its parts use any other parts not in the subsystem. The same determination tells us which how to find a minimal subsystem for initial implementation. For example, the parts *Display*, *Render* and *AST* form a collection without dependences on other parts, and could be reused as a unit.
- *Construction Order.* The uses diagram helps determine what order to build the parts in. We might assign two sets of parts to two different groups and let them work in parallel. By ensuring that no part in one set uses a part in another set, we can be sure that neither group will be stalled waiting for the other. And we can construct a system incrementally by starting at the bottom of the uses diagram, with those parts that don't use any other parts, and then move upwards, assembling and testing whenever we have a consistent subsystem. For example, the *Display* and *Protocol* parts could be developed independently along with the parts they use, but not *Display* and *Parser*.

Thinking about these considerations can shed light on the quality of a design. The cycle we mentioned above, (Main–Display–GUI–Main), for example, makes it impossible to reuse the *Display* part without also reusing *Main*.

There's a problem with the uses diagram though. Most of the analyses we've just discussed involve finding all parts reachable or reaching a part. In a large system, this may be a high proportion of the parts in a system. And worse, as the system grows, the problem gets worse, even for existing parts which refer directly to no more parts than they did before. To put it differently, the fundamental relationship that underlies *uses* is transitive: if *A* is affected by *B* and *B* is affected by *C*, then *A* is affected by *C*. It would be much better if reasoning about a part, for example, required looking at only at the parts it refers to.

The idea of the uses relation, and its role in thinking about software structure, was first described by David Parnas in *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, Vol. SE-5, No 2, 1979.

2.2.2 Dependences & Specifications

The solution to this problem is to have instead a notion of dependence that stops after one step. To reason about some part *A*, we will need to consider only the parts it depends on. To make this possible, it will be necessary for every part that *A* depends on to be complete, in the sense that its description completely characterizes its behaviour. It cannot itself depend on other parts. Such a description is called a *specification*.

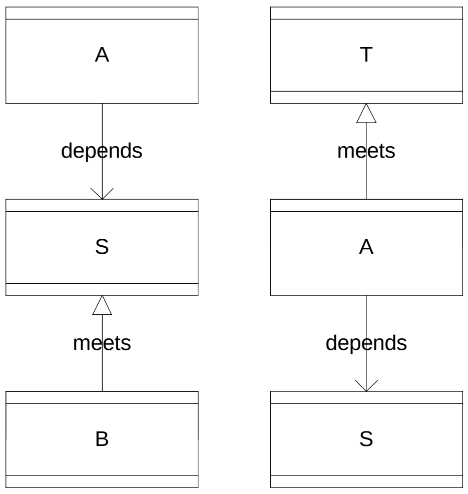
A specification cannot be executed, so we'll need for each specification part at least one implementation part that behaves according to the specification. Our diagram, the *dependency diagram*, therefore has two kinds of arcs. An implementation part may *depend* on a specification part, and it may *fulfill* or *meet* a specification part.

In comparison to what we had before, we have broken the *uses* relationship between two parts *A* and *B* into two separate relationships. By introducing a specification part *S*, we can say that *A* depends on *S* and *B* meets *S*. The diagram on the left illustrates this; note the use of two double lines to distinguish specification parts from implementation parts.

Each arc incurs an obligation. The writer of *A* must check that it will work if it is assembled with a part that satisifies the specification *S*. And 'works' is now defined by explicitly by meeting specifications: *B* will be usable in *A* if it works according to the specification *S*, and *A* will be deemed to work if it meets whatever specification is given for its intended uses – *T* say. The diagram on the right shows this. It's the same depends-meet chain centered on an implementation part rather than a specification part.

This is a much more useful and powerful framework than *uses*. The introduction of specifications brings many advantages:

- *Weakened Assumptions*. When *A* uses *B*, it is unlikely to rely on every aspect of *B*. Specifications allow us to say explicitly which aspects matter. By making specifications much smaller and simpler than implementations, we can make it much easier to reason about correctness of parts. And a weak specification gives more opportu-

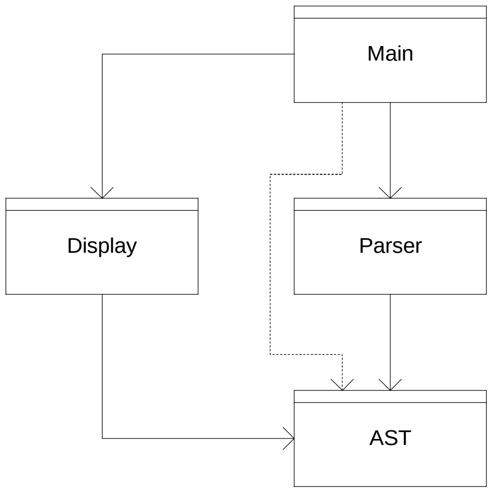


nities for performance improvements.

- *Evaluating Changes*. The specification *S* helps limit the scope of a change. Suppose we want to change *B*. Must *A* change as well? Now this question doesn't require looking at *A*. We start by looking at *S*, the specification *A* requires of the part it uses. If the new *B* will still meet *S*, then no change to *A* will be needed at all.
- *Communication*. If *A* and *B* are to be built by different people, they only need to agree upon *S* in advance. *A* can ignore the details of the services *B* provides, and *B* can ignore the details of the needs of *A*.
- *Multiple Implementations*. There can be many different implementation parts that meet a given specification part. This makes a market in interchangeable parts possible. Parts are marketed in a catalog by the specifications they meet, and a customer can pick any part that meets the required specification. A single system can provide multiple implementations of a part. The selection can be made when the system is configured, or as we shall see later in the course, during execution of the system.

Specifications are so useful that we'll assume that there is a specification part corresponding to every implementation part in our system, and we'll conflate them, drawing dependences directly from implementations to implementations. In other words, a dependence arc from *A* to *B* means that *A* depends on the specification of *B*.

So whenever we draw a diagram like the one of our browser above, we'll interpret it as a dependence diagram and not as a uses diagram. For example, it will be possible to have teams build *Parser* and *Protocol* in parallel as soon as the *specification* of *Page* is



complete; its implementation can wait.

Sometimes, though, specifications are design elements in their own right and we'll want to make explicit their presence. Java provides some useful mechanisms for expressing decoupling with specifications, and we'll want to show these. Design patterns, which will be studying later in the term, make extensive use of specifications in this way.

2.2.3 Weak Dependences

Sometimes a part is just a conduit. It refers to another part by name, but doesn't make use of any service it provides. The specification it depends on requires only that the part exist. In this case, the dependence is called a *weak dependence*, as is drawn as a dotted arc.

In our browser, for example, the abstract syntax tree in AST may be accessible as a global name (using the Singleton pattern, which we'll see later). But for various reasons – we might for example later decide that we need two syntax trees – it's not wise to use global names in this way. An alternative is for the *Main* part to pass the *AST* part from the *Parse* part to the *Display* part. This would induce a weak dependence of *Main* on *AST*. The same reasoning would give a weak dependence of *Main* on *Page*.

In a weak dependence of *A* on *B*, *A* usually depends on the name of *B*. That is, it not only requires that there be some part satisfying the specification of *B*, but it also requires that it be called *B*. Sometimes a weak dependence doesn't constrain the name.

In this case, *A* depends only the existence of some part satisfying the specification of *B*, and *A* will refer to such a part using the name of the specification of *B*. We will see how Java allows this kind of dependence. In this case, it's useful to show the specification of *B* as a separate part with its own name.

For example, the *Display* part of our browser may use a part *UI* for its output, but need not know whether the *UI* is graphical or text-based. This part can be a specification part, met by an implementation part *GUI* which *Main* depends on (since it creates the actual GUI object). In this case, *Main*, because it passes an object whose type is described as *UI* to *Display*, must also have a weak dependence on the specification part *UI*.

2.3 Techniques for Decoupling

So far, we've discussed how to represent dependences between program parts. We've also talked about some of the effects of dependences on various development activities. In every case, a dependence is a *liability*: it expands the scope of what needs to be considered. So a major part of design is trying to minimize dependences: to *decouple* parts from one another.

Decoupling means minimizing both the quantity and quality of dependences. The quality of a dependence from *A* to *B* is measured by how much information is in the specification of *B* (which, recall from above, is what *A* actually depends on). The less information, the weaker the dependence. In the extreme case, there is no information in the dependence at all, and we have a weak dependence in which *A* depends only on the existence of *B*.

The most effective way to reduce coupling is to design the parts so that they are simple and well defined, and bring together aspects of the system that belong together and separate aspects that don't. There are also some tactics that can be applied when you already have a candidate decomposition: they involve introducing new parts and altering specifications. We'll see many of these throughout the term. For now, we'll just mention some briefly to give you an idea of what's possible.

2.3.1 Facade

The facade pattern involves interposing a new implementation part between two sets of parts. The new part is a kind of gatekeeper: every use by a part in the set *S* of a part in the set *B* which was previously direct now goes through it. This often makes sense in a layered system, and helps to decouple one layer from another.

In our browser, for example, there may be many dependences between parts in a protocol layer and parts in a networking layer. Unless all the networking parts are platform-independent, porting the browser to a new platform may require replacing the networking layer. Every part in the protocol layer that depends on a networking part may have to be examined and altered.

To avoid this problem, we might introduce a facade part that sits between the layers, collects together all the networking that the protocol layer needs (and no more), and presents them to the protocol layer with a higher-level interface. This interface is, of course, a new specification, weaker than the specifications on which the protocol parts used to rely. If done right, it may now be possible to change the parts of the networking layer while leaving the facade's specification unchanged, so that no protocol parts will be affected.

2.3.2 Hiding representation

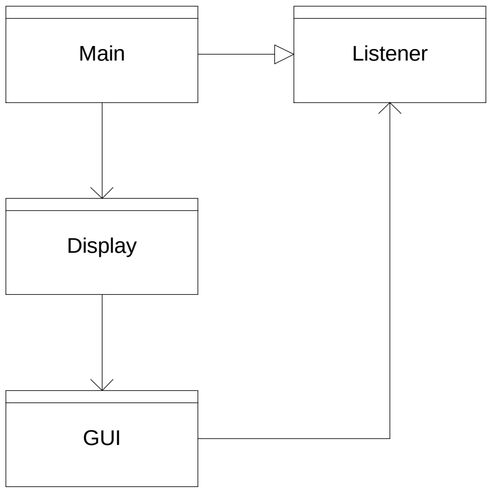
A specification can avoid mentioning how data is represented. Then the parts that depend on it cannot manipulate the data directly; the only way to manipulate the data is to use operations that are included in the specification of the used part. This kind of specification weakening is known as 'data abstraction', and we'll have a lot to say about it in the next few weeks. By eliminating the dependence of the using part *A* on the representation of data in the used part *B*, it makes it easier to understand the role that *B* plays in *A*. It makes it possible to change the representation of data in *B* without any change to *A* at all.

In our browser, for example, the specification part associated with *Page* might say that a web page is a sequence of characters, hiding details of its representation using arrays.

2.3.3 Polymorphism

A program part *C* that provides container objects has a dependence on the program part *E* that provides the elements of the container. For some containers, this is a weak dependence, but it need not be: *C* may use *E* to compare elements (eg, to check for equality, or to order them). Sometimes *C* may even use functions of *E* that mutate the elements.

To reduce the coupling between *C* and *E*, we can make *C* polymorphic. The word 'polymorphic' means 'many shaped', and refers to the fact that *C* is written without any mention of special properties of *E*, so that containers of many shapes can be produced according to which *E* the part *C* uses. In practice, pure polymorphism is rare, and *C*



will at least rely on equality checks provided by *E*. Again, what's going on is a weakening of the specification that connects *C* to *E*. In the monomorphic case, *C* depends on the specification of *E*; in the polymorphic case, *C* depends on a specification *S* that says only that the part must provide objects with an equality test. In Java, this specification *S* is the specification of the *Object* class.

In our browser, for example, the data structure used for the abstract syntax tree might use a generic *Node* specification part, which is implemented by an *HTMLNode* part, for much of its code. A change in the structure of the markup language would then affect less code.

2.3.4 Callbacks

We mentioned above how, in our browser, a *GUI* part might depend on the *Main* part because it calls a procedure in *Main* when, for example, a button is pressed. This coupling is bad, because it makes intertwines the structure of the user interface with the structure of the rest of the application. If we ever want to change the user interface, it will be hard to disentangle it.

Instead, the *Main* part might pass the *GUI* part at runtime a reference to one of its procedures. When this procedure is called by the *GUI* part, it has the same effect it would have had if the procedure had been named in the text of the *GUI* part. But since the association is only made at runtime, there is no dependence of *GUI* on *Main*. There will be a dependence of *GUI* on a specification (*Listener*, say) that the passed procedure must satisfy, but this is usually minimal: it might say, for example, just that the procedure returns without looping forever, or that it does not cause procedures within *GUI* itself to be called. This arrangement is a *callback*, since *GUI* 'calls back' to *Main* against the usual direction of procedure call. In Java, procedures can't be passed, but the same effect can be obtained by passing a whole object.

2.4 Coupling Due to Shared Constraints

There's a different kind of coupling which isn't shown in a module dependency diagram. Two parts may have no explicit dependence between them, but they may nevertheless be coupled because they are required to satisfy a constraint together.

For example, suppose we have two parts, *Read*, which reads files, and *Write*, which writes files. If the files read by *Read* are the same files written by *Write*, there will be a constraint that the two parts agree on the file format. If the file format is changed, both parts will need to change.

To avoid this kind of coupling, you have to try to localize functionality associated with any constraint in a single part. This is what Matthias Felleisen calls 'single point of control' in his novel introduction to programming in Scheme (*How to Design Programs, An Introduction to Programming and Computing*, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, MIT Press, 2001).

David Parnas suggested that this idea should form the basic of the selection of parts. You start by listing the key design decisions (such as choice of file format), and then assign each to a part that keeps that decision 'secret'. This is explained in detail with a nice example in his seminal paper *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No. 12, December 1972 pp. 1053–1058.

2.5 Back to Dijkstra: Conclusion

Dijsktra's warning that the chance of getting a program right will drop to zero as the number of parts increases is worrying. But if we can decouple the parts so that each of the properties we care about is localized within only a few parts, then we can establish their correctness locally, and be immune to the addition of new parts.