

Decoupling

6.170 Lecture 2 Summary

Spring 2002: Devadas/Guttag

Reading: Chapter 1, 13:1-3 of *Program Development in Java* by Barbara Liskov

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.

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.

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*.

1.1 Why Decompose?

What benefits come from dividing a program into smaller parts. Here are some:

- *Division of labor*. 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 building it in small parts. Each time a part is complete, it can be analyzed for correctness.
- *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.

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. 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.

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 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.

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.

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.

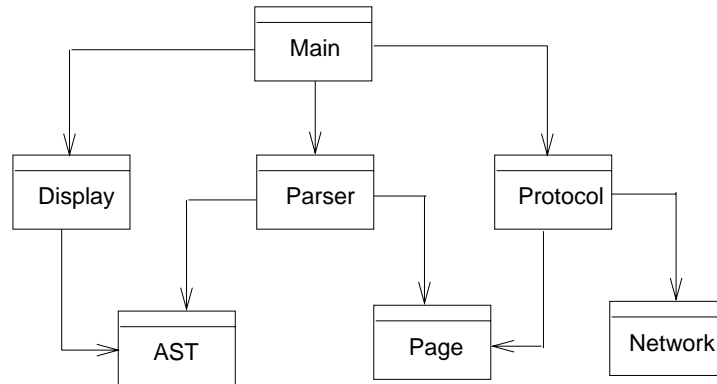
2 Dependence Relationships

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 .

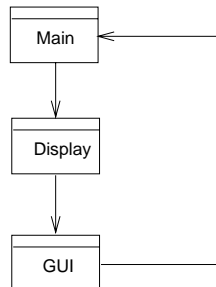
When A and B are executable code, the meaning of A is its behavior when executed, so A uses B when the behavior of A depends on the behavior of B .

Suppose, for example, we are 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.

A uses graph is usually not a tree because reuse implies that a part may have multiple users. A uses graph may be layered. Finally, there may be cycles in the uses graph.



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 use *GUI* for output, and *GUI* may use *Main* for input.

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 . 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.
- *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 how to find a minimal subsystem for initial implementation. For example, the parts

Display 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.

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.

2.2 Dependences and Specifications

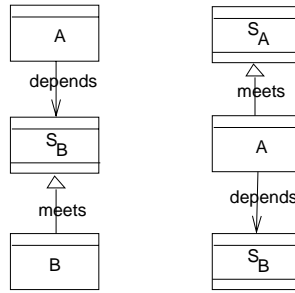
The solution to this problem is to have instead a notion of dependence that stops *after one step*. To reason about some part *A*, we will need to consider only the parts it depends on. To make this possible, it will be necessary for every part that *A* depends on to be complete, in the sense that its description completely characterizes its behavior. 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 satisfies 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 opportunities 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.

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.

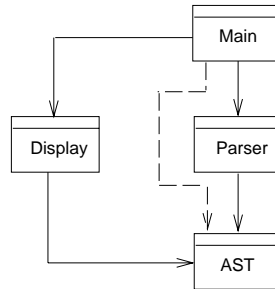
Sometimes, though, specifications are design elements in their own right and we'll want to make explicit their presence. Java provides some useful mechanisms for expressing decoupling with specifications, and we'll want to show these. Design patterns, which will be studying later in the term, make extensive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

2.3 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 or a dashed 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*.



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*.

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.

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. They are:

- Hiding representation: By eliminating the dependence of the using part *A* on the representation of data in the used part *B*, it is easier to understand the role that *B* plays in *A*.
- Facade design pattern: Interposing a new implementation part between two sets of parts can help decouple one layer from another in layered system.
- Polymorphism: A program part *C* that provides container objects has a dependence on the program part *E* that provides the elements of the container. The specification that connects *C* to *E* is weakened using polymorphism. Rather than *C* depending on the specification of *E* in the monomorphic case, *C* depends on a specification *S* that says only that the part must provide objects with, say, an equality test, in the polymorphic case.
- Callbacks: A part *A* might pass another part *B* at runtime a reference to one of its procedures. When this procedure is called by the *B*, it has the same effect it would have had if the procedure had been named in the text of *B*. But since the association is only made at runtime, there is no dependence of *B* on *A*. This arrangement is a *callback*, since *B* ‘calls back’ to *A* against the usual direction of procedure call.