

Visit at MIT:  
Daniel Jackson's Lectures on  
Coupling  
Karl Lieberherr

<http://ocw.mit.edu/6/6.170/f01/index.html>

# Decoupling I and II

- Daniel Jackson (son of Michael Jackson, author of the Jackson Design Method) has two lectures on coupling in his SE class.
- Because we are also focussing on coupling (LoD), let's learn from the MIT software engineering course.
- The LoD also encourages the use of Java interfaces to reduce coupling.

# Overview

- Specifications: Essential for decoupling
- Uses and depends relationships
- Decomposition is beneficial
  - Division of labor
  - Reuse
  - Modular analysis
  - Localized change

# Alternative to Functional decomposition

- A much better strategy is to develop a system structure consisting of multiple parts at a roughly equal level of abstraction.

# Uses relationship

The most basic 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*.

# 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?
- *Construction Order*. The uses diagram helps determine what order to build the parts in.

# Problem with uses diagram

- 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.
- It would be much better if reasoning about a part, for example, required looking at only the parts it refers to.

# 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*.

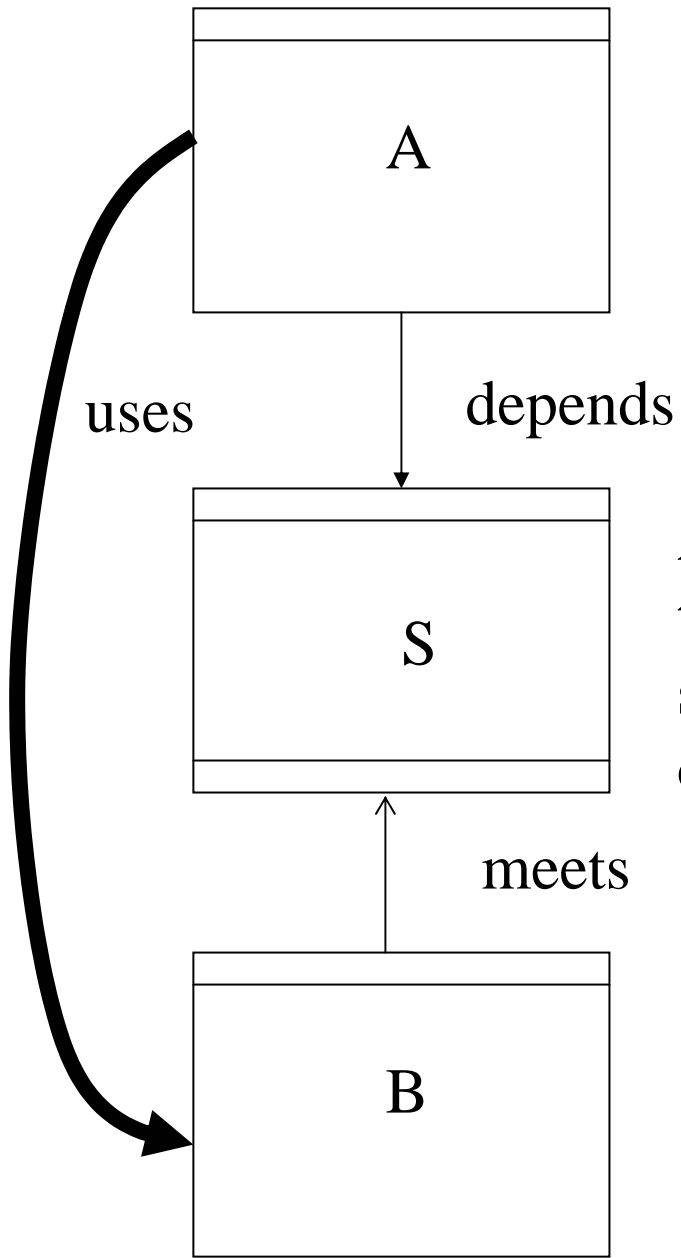


# Specifications

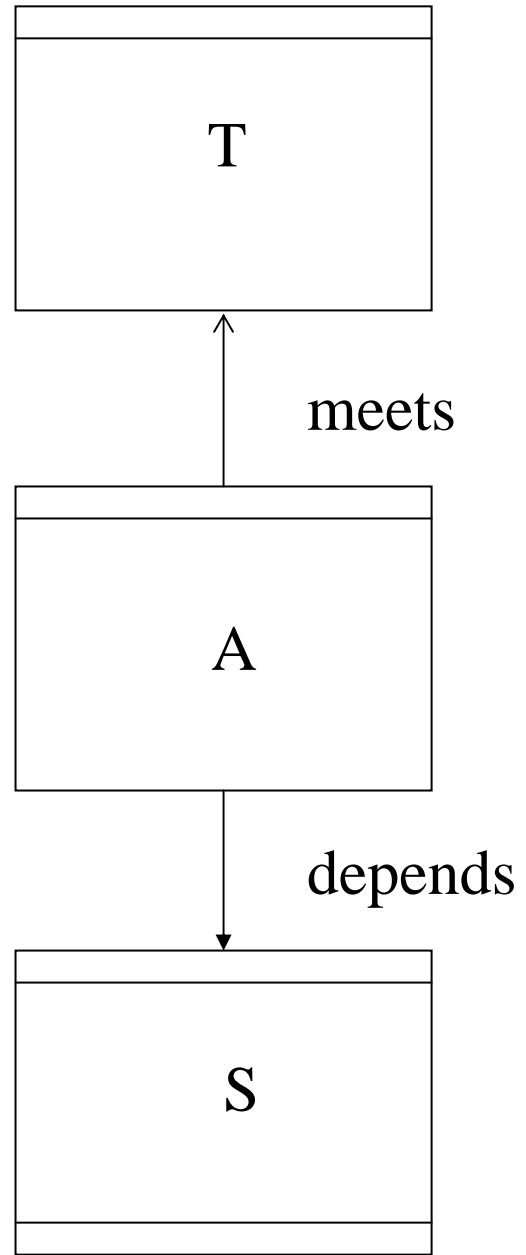
- A specification cannot be executed, so we'll need for each specification part at least one implementation part that behaves according to the specification.
- Our diagram, the *dependency diagram*, therefore has two kinds of arcs. An implementation part may *depend* on a specification part, and it may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

# From uses to depends

- 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*.



A depends on  
the  
specification  
of B



# What has improved?

- 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.

# Reuse

- *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.

# Interpret as dependence diagram

- 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  $B$ .

# 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 dependencies 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.

# quantity and quality

- 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  $B$ .



# Techniques to reduce coupling

- 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.

# 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.

# 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.

# 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.

# Command Pattern (Callbacks)

- See design pattern book

# 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.

# Coupling Due to Shared Constraints

- 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.
- We need AOSD to do this well!

# Conclusion

- 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 the correctness of the properties locally, and be immune to the addition of new parts.



# Example: Instrumenting a program

- we'll study some decoupling mechanisms in the context of an example that's tiny but representative of an important class of problems.
- we want to report incremental steps of a program as it executes by displaying progress line by line.

# First solution

- Intersperse statements such as  
*System.out.println (o + "Message 1");*  
at 5 places.

# Problems

- What if we want to print to a file or time stamp the messages?

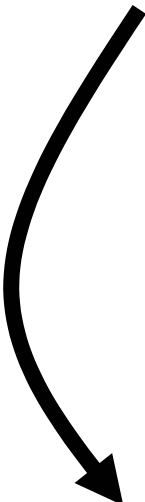
*System.out.println (o + "Message2: " +  
new Date ());*

- 5 lines to change

*StandardReporter.report(m,o);*

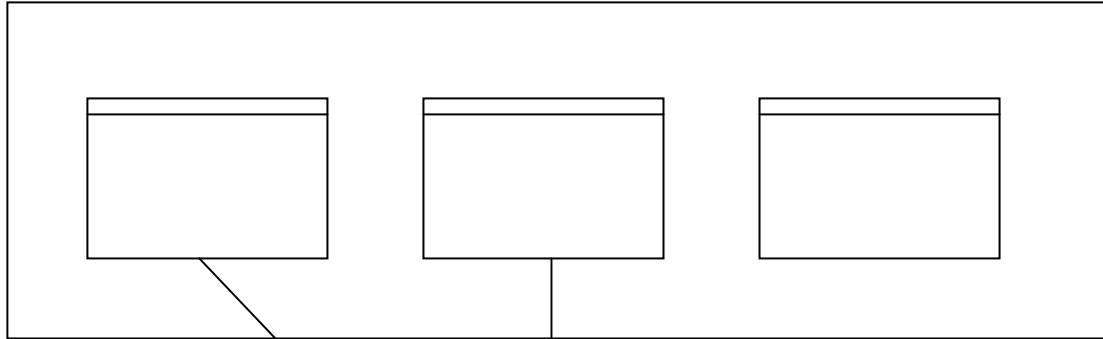
## Second solution

```
public class StandardReporter {  
    public static void report (String msg, Object o) {  
        System.out.println (o + msg);  
    }  
}
```

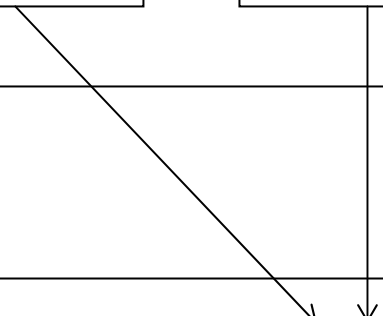
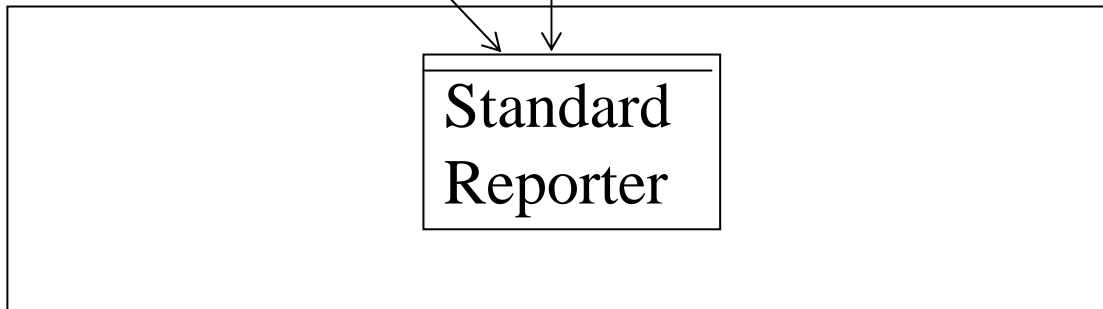


```
public class StandardReporter {  
    public static void report (String msg, Object o) {  
        System.out.println (o + msg + " at: " + new Date ());  
    }  
}
```

core



ui



# Problem

- This scheme is far from perfect though. Factoring out the functionality into a single class was a good idea, but the code still has a dependence on the notion of writing to standard out.

# Third Solution

```
public interface Reporter {  
    void report  
    (String msg, Object o);  
}
```

```
void download (Reporter r, ...) {  
    ...  
    r.report ("message3", o );  
    ...  
}
```

```
public class StandardReporter implements Reporter {  
    public void report (String msg, Object o) {  
        System.out.println (o + msg + " at: " + new Date ());  
    }  
}
```

# Third Solution

```
public class VisitorReporter implements Reporter {  
    Visitor rv;  
    public VisitorReporter (Visitor c) {rv = c;}  
    public void report (String msg, Object o) {  
        o.print(rv);  
        System.out.println (msg + " at: " + new Date ());  
    }  
}
```

```
s.download (new VisitorReporter (new PrintVisitor()), ...);
```





# Decoupling successful!

- We've successfully decoupled the output mechanism from the program, breaking the dependence of the core of the program on its I/O.
- The key property of this scheme is that there is no longer a dependence of any class of the *core* package on a class in the *ui* package.

# Useful idiom!

- This idiom is perhaps the most popular use of interfaces, and is well worth mastering.

# Connection to Law of Demeter

- OF/CF
- Count as violations:
  - sending a message to global variable
  - calling a global method

# Connection to the Law of Demeter

- In this example, following the Law of Demeter (minimizing violations) invites the introduction of a Java interface.
- This is an interesting kind of transformation that improves adherence to LoD.

# Connection to Law of Demeter

- 1. Solution
  - 5 references to global variable `System.out`
- 2. Solution
  - 5 references to global method `StandardReporter.report(..)`
  - 1 call to global variable `System.out`
- 3. Solution
  - 1 call to global variable `System.out`

# Connection to Law of Demeter

- 1. Solution
  - OF(5 calls to global variable on different lines) 5
  - CF(5 calls to global variable) 5
- 2. Solution (abstraction by parameterization)
  - OF(5 calls to global method on different lines + 5 calls to global variable on one line) 6
  - CF(5 calls to global method + 1 call to global variable) 6
- 3. Solution
  - OF(5 calls to global variable on one line) 1
  - CF(1 call to global variable) 1

# Conclusions

- The LoD also encourages the use of interfaces. Points in the same direction as Jackson's lecture.
- Jackson points out: Decoupling means minimizing both the quantity and quality of dependences. How can we improve the LoD to deal with quality? Currently we focus on quantity.



# Conclusions

- What other transformations exist that reduce violations to the Law of Demeter?