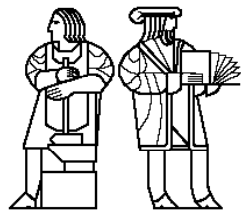


6.170 Lecture 16

Module dependences



Michael Ernst
MIT EECS



Module dependences

Namespaces

Access control

Module dependence diagrams (MDDs)

Reducing dependences

procedures

global variables

interfaces



Package structure

Hierarchical decomposition of a software system

Encourages understanding

- High-level view of structure, omitting details
- One part in isolation from the others

Allows reuse of names

System may or may not be built top-down

Java packages correspond to directories

- java, java.lang, java.util, ...
- In 6.170: ps1, ps2, ...
- Your final project may more effectively use modules



Access control

Java classes, methods, and fields can be annotated as:

public: accessible anywhere (that it can be named)

private: accessible only within the declaring class itself

protected: accessible within the class and subclasses

no annotation: accessible anywhere within the package

- protected** is *more* accessible than default access
- for interfaces, no annotation means **public**

Access control helps to maintain representation invariants

Can use getter and setter methods to access private variables

- permit access
- perform extra checking/logging
- typically not used within the class



Access control summary

Accessibility	Visibility			
	public	protected	package	private
Same class	yes	yes	yes	yes
Class in same package	yes	yes	yes	no
Subclass in different package	yes	yes	no	no
Non-subclass, different package	yes	no	no	no



Module dependences

A depends on B if A names or uses B

Object models (OMs) indicate dynamic (run-time) dependences

indicate all possible run-time configurations, so all possible run-time uses

Module dependence diagrams (MDDs) indicate static (compile-time) dependences

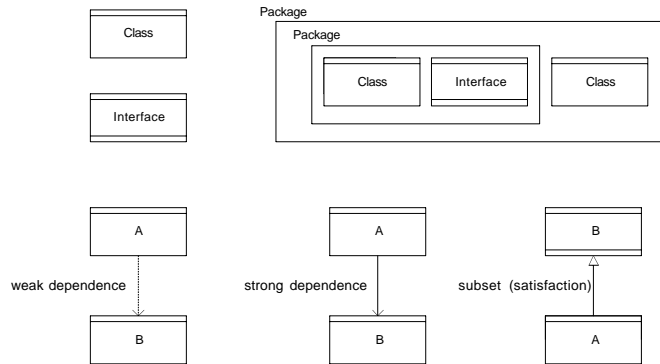
indicate which classes are used in the code, and how dependences are weak if a class is merely named
dependences are strong if its methods or fields are used

OMs and MDDs are often similar

differences indicate subtleties of the code or design



Module dependence diagram syntax



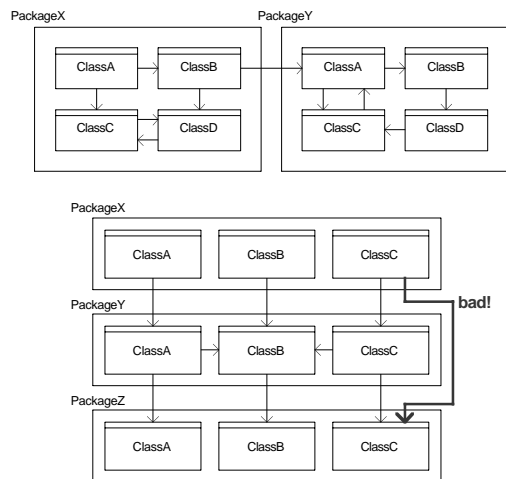
Michael Ernst/John Guttag

Spring 2001

Slide 7



Typical module dependences



One goal: reduce dependences

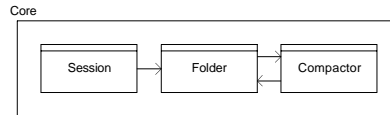
Michael Ernst/John Guttag

Spring 2001

Slide 8



Running example: output from an email client



Indicate progress of an operation

One solution:

`System.out.println("Starting download");`
throughout the program



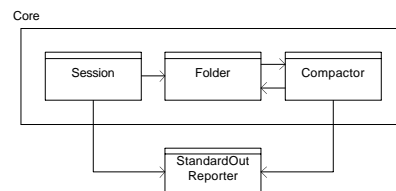
Problem: new output format requires many changes

Suppose that we wish to timestamp messages

Solution: procedural abstraction

"Single point of control" principle

```
public class StandardOutReporter {
    public static void report(String msg) {
        System.out.println(msg + " at " + new Date());
    }
}
```



Folder does not depend on
StandardOutReporter



Problem: assumes that output is performed via println

Suppose that we wish to support output to a GUI window

```
JTextComponent outputArea = ...;  
outputArea.setText(msg);
```

Solution: use an interface to decouple messages from how they are communicated to the user

```
public interface Reporter {  
    void report(String msg);  
}  
  
void download(Reporter r, ...) {  
    r.report("Starting download");  
}
```



Creating Reporters

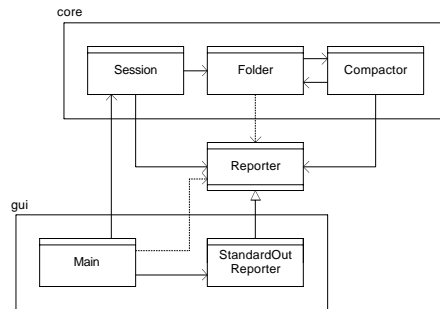
```
public class StandardOutReporter implements Reporter {  
    public void report(String msg) {  
        System.out.println(msg + " at " + new Date());  
    }  
}  
  
public class JTextComponentReporter implements Reporter {  
    JTextComponent comp;  
    public JTextComponentReporter(JTextComponent comp) {  
        this.comp = comp;  
    }  
    public void report(String msg) {  
        comp.setText(msg + " at " + new Date());  
    }  
}
```

In the client (Session):

```
s.download(new StandardOutReporter(), ...);
```



Module dependence diagram for interface version



Folder now depends
on Reporter

**No dependence on StandardOutReporter (or on
JTextComponentReporter)**

**System is more flexible at the cost of more complexity
(more dependences)**



Digression: Abstract classes vs. interfaces

An abstract class is similar to an interface:

can declare unimplemented abstract methods

Abstract classes have additional capabilities:

declare constructors
declare fields
implement some methods

Advantages of interfaces:

Constrain less
Sometimes there is no common code to be factored out
Java has single inheritance of classes, but multiple
inheritance of implementations



Problem: Folder depends on Reporter

We must pass the Reporter through from Session

Folder shouldn't care about Reporter
bloats parameter lists
lots of code to change

Solution: global variables (well, static fields)

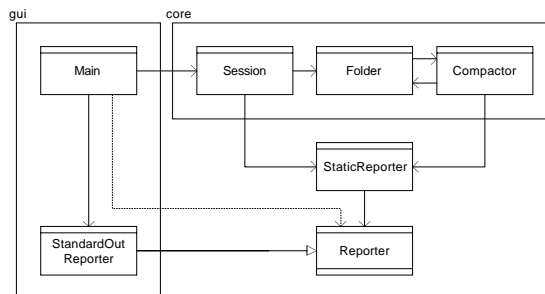
```
public class StaticReporter {  
    private static Reporter r;  
    public static void setReporter(Reporter r) {  
        this.r = r; }  
    public static void report(String msg) {  
        r.report(msg); } }  
}
```

Must initialize in Session:

```
StaticReporter.setReporter(new  
    StandardOutReporter(), ...);
```



MDD with static field holding reporter



Eliminates weak dependence of Folder on Reporter



Problems with global variables

Globals can be modified anywhere

Code is hard to understand and reason about

Only really works if exactly one global object

What about using different panes for different kinds of output?

Prohibits certain kinds of concurrency

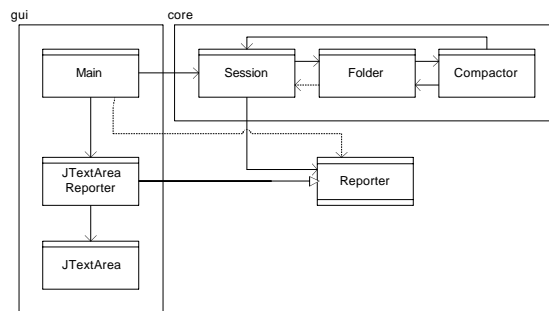
What about simultaneous downloads from multiple folders?

Be very suspicious of global variables



Dynamic configuration

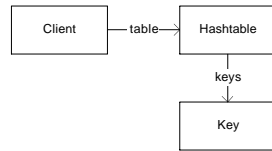
Use a single Reporter per Session, but determine it at run time rather than at compile time





Comparing OMs and MDDs for Hashtable

Object model:



Module dependence diagram:

