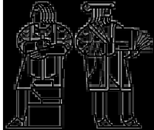
6.170 Lecture 17
MDDs



John Chapin
MIT EECS
3-13-2000

Outline

1. Motivation for MDDs
2. Syntax of MDDs
3. How to construct MDDs

1. Motivation

MDD (Module Dependency Diagram) shows
- Code organization
- Dependencies between modules

Why focus on dependencies?
- reasoning about correctness
- propagation of changes

Uses for MDDs:

Design:	choosing a module structure
Implementation:	division of labor
Testing:	integration test plan
Maintenance:	what needs to change

Reminder about design notations

Object models (last lecture)
- design notation for describing a problem
- no implementation artifacts

MDDs
- design notation for describing an implementation
- usually very different from OM for that system

Need both together to understand design
- along with state machines, architecture, etc.
- each is a partial description
- each is precise and abstract

Definition of dependency

Say:
- A depends on B iff
- a change in B's behavior can change A's behavior

This approach ignores encapsulation
- `main` uses `WtDiGraph` uses `Table` uses `Vector`
- designed so `main` doesn't know `Vector`

In MDDs:
- A depends on B iff
- a change in B's specification may require A to change

If A uses X and X uses B, A does *not* necessarily use B

2. Syntax of MDDs

A graph
- nodes are modules
- arcs are dependencies

Modules

class

interface

static procedure

Always have specification
Sometimes have implementation

Dependencies (1 of 3)

A strongly uses B
A depends on the specification of B

ptest → Point

```
// effects: returns the argument value
public static int ptest(int val) {
  Point p = new Point(val, 4);
  return p.x();
}
```

Notes
Interfaces cannot strongly use other modules

Dependencies (2 of 3)

A weakly uses B
A relies on existence of B

Rectangle - - - -> Point

```
interface Rectangle {
   Point lowerleft();
   Point upperright();
}
```

Notes
Cannot weakly use a procedure

Dependencies (3 of 3)

A is a subtype of B
spec of A is stronger than spec of B

```
class ColorPoint extends Point {
   ...
}

class Vector implements Clonable {
   ...
}

interface Mixer extends Gadget {
   ...
}
```

Point ← ColorPoint

Clonable ← Vector

Mixer → Gadget

3. How to construct MDDs

Rule: A module that names another module uses it

weak use:	only name appears
strong use:	calls a method
subtype:	if declared to implement or extend (and if actually a subtype)

Rule: A module that does not name it does not use it
Exception: hidden dependencies

Interesting cases: what about when module
uses only the constructor
uses only the supertype
extends Object
has a hidden dependency

When module uses only the constructor

```
public static Point makepoint() {
  Point p = new Point();
  return p;
}
```

Using the constructor is a strong use

makepoint → Point

The factory pattern is used to break this dependency

When module uses only the supertype

```
public static int ptest(Point p) {
   return p.x();
}
class ColorPoint extends Point {
   ...
}
interface Rectangle {
   Point lowerleft();
   Point upperright();
}
```

`ptest` might call `x()` on a `ColorPoint`.
`lowerleft()` might return a `ColorPoint`.

Your turn: draw an MDD for the above modules

When a module uses a supertype (answer)

ptest
Point
Rectangle
ColorPoint

No weak use of ColorPoint
- ptest, Rectangle ok if ColorPoint deleted

No strong use of ColorPoint
- ptest, Rectangle unchanged if ColorPoint spec changed as long as it remains a subtype of Point

When a module extends Object

Object normally omitted from MDD

Sometimes a class overrides a method of Object
- Eg. `equals`, `hashCode`, or `toString`

pseudo-rule: MDD must show Object in places where
- a class overrides a method of Object, and
- a mistake in this code could cause an error, and
- some module uses this method but does not use the class directly

real rule: MDD must show Object in places where
- designer was thinking about Object's spec when developing design

When a module extends Object (example)

```
static void btest(hashtable h) {
   h.insert(new MyObj());
}

class hashtable {
   public void insert(Object val){
      int code = val.hashCode();
      ...
   }
}

class MyObj {
   public int hashCode() {
      return ...;
   }
}
```

hashtable
btest
Object
MyObj

When a module extends Object (counterexample)

```
class MyObj {
   public int toString() {
      return ...;
   }
}

static void warning(string s, Object o) {
   System.out.print(s);
   System.out.print(o.toString());
}
```

Value returned by toString not used in computation
- Would not show MyObj extends Object in this case

When a module has a hidden dependency

Real rule:
- If the designer had to think about B's spec when implementing A
- then A has a strong use of B even if B not named

```
static void version1() {
   Point p = getPoint();
   p.draw();
}

static void version2() {
   Point p = getPoint();
   p.setColor();   // required by ColorPoint.draw()
                   // No effect if not a ColorPoint
   p.draw();
}
```

Summary

Dependences

- Are generally bad
- MDD shows where they are
- When reviewing design, check each to see if it is necessary