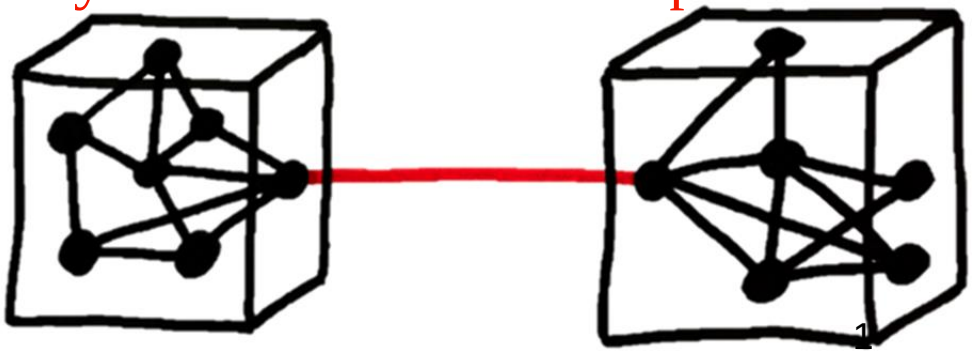
Chap7 (Schach) Cohesion and Coupling

- *Cohesion* - degree of interaction *within* a module
- *Coupling* - degree of interaction *between* modules
  - Want strong relationships between components *within* modules:
    - Aids in the localization of faults
    - Clearer insight on how parts of the system could be reused
  - Want minimal relationships *between* modules:
    - Modules will be relatively independent
    - Modifications to one subsystem will have little impact on the other modules



But Wait…. What's a module?

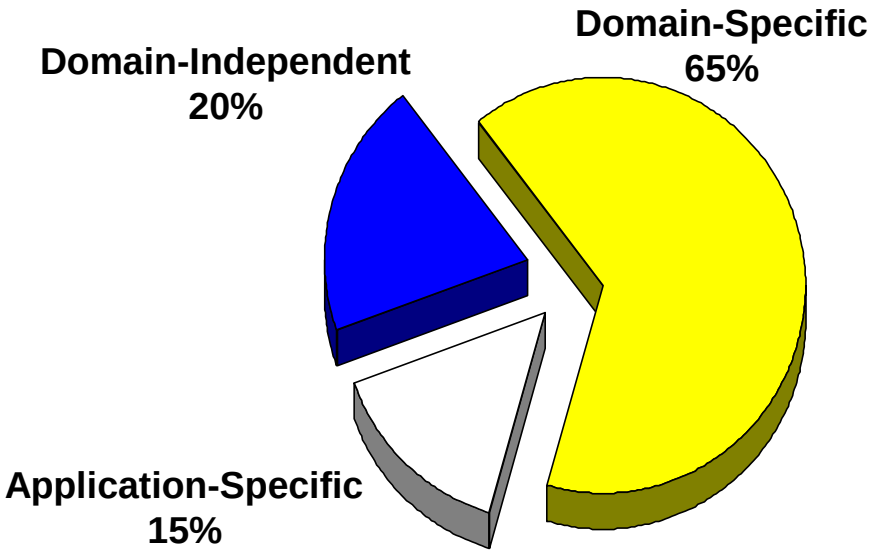
- *1974: "a set of one or more contiguous program statements having a name by which other parts of the system can invoke it, and preferably having its own distinct set of variable names."* Sounds very generic but is actually too restrictive.
- *1979: "a lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier."* Our definition for this course.
- But wait, sir… What's that *mean*?
  - Classical/Procedural Paradigm:
    - a function or procedure
  - Object Oriented Paradigm:
    - a class
    - a method/function within an class

Cohesion/coupling **impacts** Software Reuse

- *Software Reuse*: *Using* components from one product in the developing a different product
  Goal: Spend less on software production, but (hopefully) get increased quality.

- What (exactly) is reused?
  - Application system reuse
  - Sub-system reuse
  - Class reuse
  - Functional (low-level) reuse

- Typically, software breaks down into three areas that impact reuse:



Cohesion: *Why* are these things together?

- *Ideal:* A module implements a single logical function or entity and all parts of the module contribute to this implementation.

  - Measure of the strength of the relationships between functionality within a given module

  - Why do we care about a module's cohesion?

Seven Levels of Cohesion

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

- **Wanted:** High (strong) cohesion so that the module will have a high *potential* for reuse.

(1) Coincidental Cohesion

- Arise from rules like: "Every module will consist of between 35 and 50 statements"
- Lines of code not really related, just bundled together Example:

```
short printNextLine(String inStr, short x, short y)
{   System.out.println(inStr);
    resetScreen();
    return x+y;
 }
```

  - What is printNextLine() intended to do?

- Why is Coincidental Cohesion so bad?
  - Difficult to maintain, not reusable
  - Easy to fix: Break into separate modules, each performing more cohesive tasks

(2) Logical Cohesion

A module contains related actions, *caller selects* which one to use

- Example 1:

```
someOperation (op code, dummy 1, dummy 2, dummy 3);
   // dummy 1, dummy 2, and dummy 3 are dummy
   // variables, not used if op code is equal to 7
```

- Example 2: One Module performs all input and output
- Example 3: OS/2
  - A version of OS/2 had a logically cohesive module with 13 different actions. Interface contained 21 arguments used to determine action selection.

- Why is logical cohesion so bad?
  - The interface is difficult to understand
  - Code for more than one action may be intertwined
  - Difficult to reuse

(3) Temporal Cohesion

- A series of actions related *only* by time (not sequence)
  - Example: **`init()`** performs several functions that are *only* related because they need to be done before other processing.

```
void init() {
  openAccountDB();
  openTransactionDB();
  resetTransactionCount();
  println("Update in progress");
  …
}
```

- Why is temporal Cohesion so bad?
  - Actions only weakly related to each other; they are more strongly related to the "other" modules.
  - Difficult to freely alter the "other" modules without also affecting a temporally cohesive module.

(4) Procedural Cohesion

- Performs a series of operations related by the *sequence* of steps to be followed.
- In other words, the order matters, and changing the order would mean that the module would no longer function correctly.
- Example:

```
void readPartNumberAndUpdateRepairRecordOnFile() {
…
}
```

- Why is Procedural Cohesion so bad?
  - Actions are still only weakly connected, so module is not so reusable
  - Better: Break into separate modules

(5) Communicational Cohesion

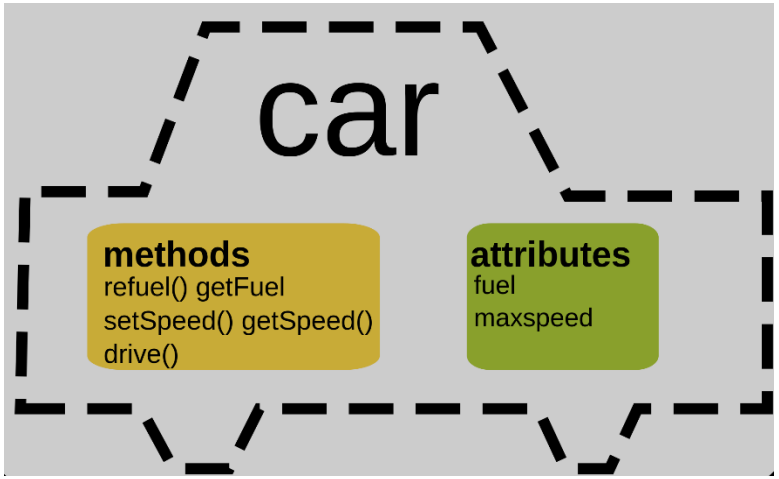
- Elements of a module perform a series of actions, and all operate on the *same input* data

  - Example 1
    - update record in database *and* write it to audit trail
  - Example 2
    - write error message to screen, then to error log file

- Why is Communicational Cohesion so bad?
  - Main drawback is lack of reusability.
  - Can't reuse the module unless all of its actions are needed in the new system.

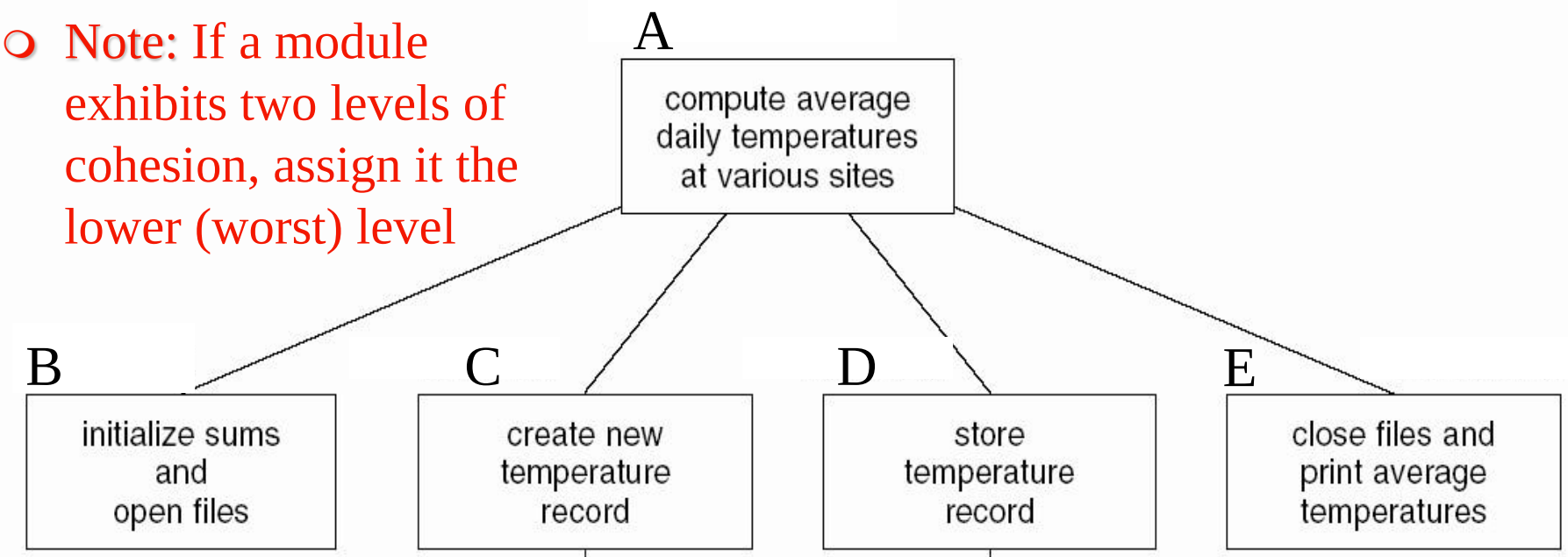
Good forms of cohesion

- **(6) Functional:** Module performs exactly ***one (and only one)*** operation or achieves a single goal.
  - Great for isolating faults, but not very practical.
  - Overkill for OOP (writing classes with only one method)
- **(7) Informational:** Performs a number of operations, each with its own entry point, with independent code for each operation, all performed on the same data structure.
  - Examples:
    - a *well designed* OO class
    - an implementation of an abstract data type



ICE: Determine Cohesiveness based on Description

- Note: If a module exhibits two levels of cohesion, assign it the lower (worst) level

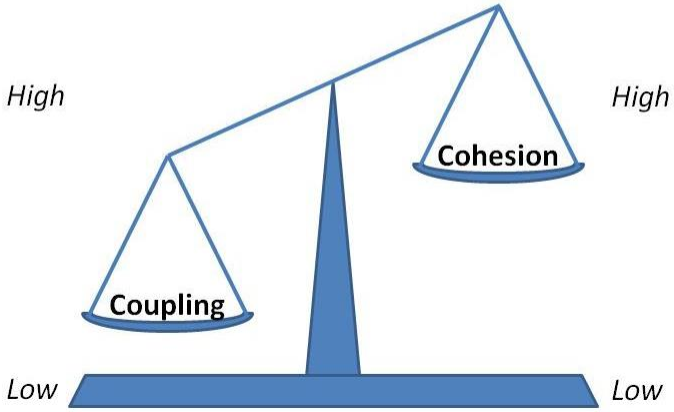


7.	Informational cohesion	(Good)
	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

ICE: Does good cohesion come from just using OO?

- High degree of cohesion *is* a feature of *properly designed* object-oriented systems.
  - OO design, if done right, does lead naturally to components that are cohesive.
  - **Question:** How should we design OO classes in order to increase their cohesiveness?

Relationship between Coupling and Cohesion

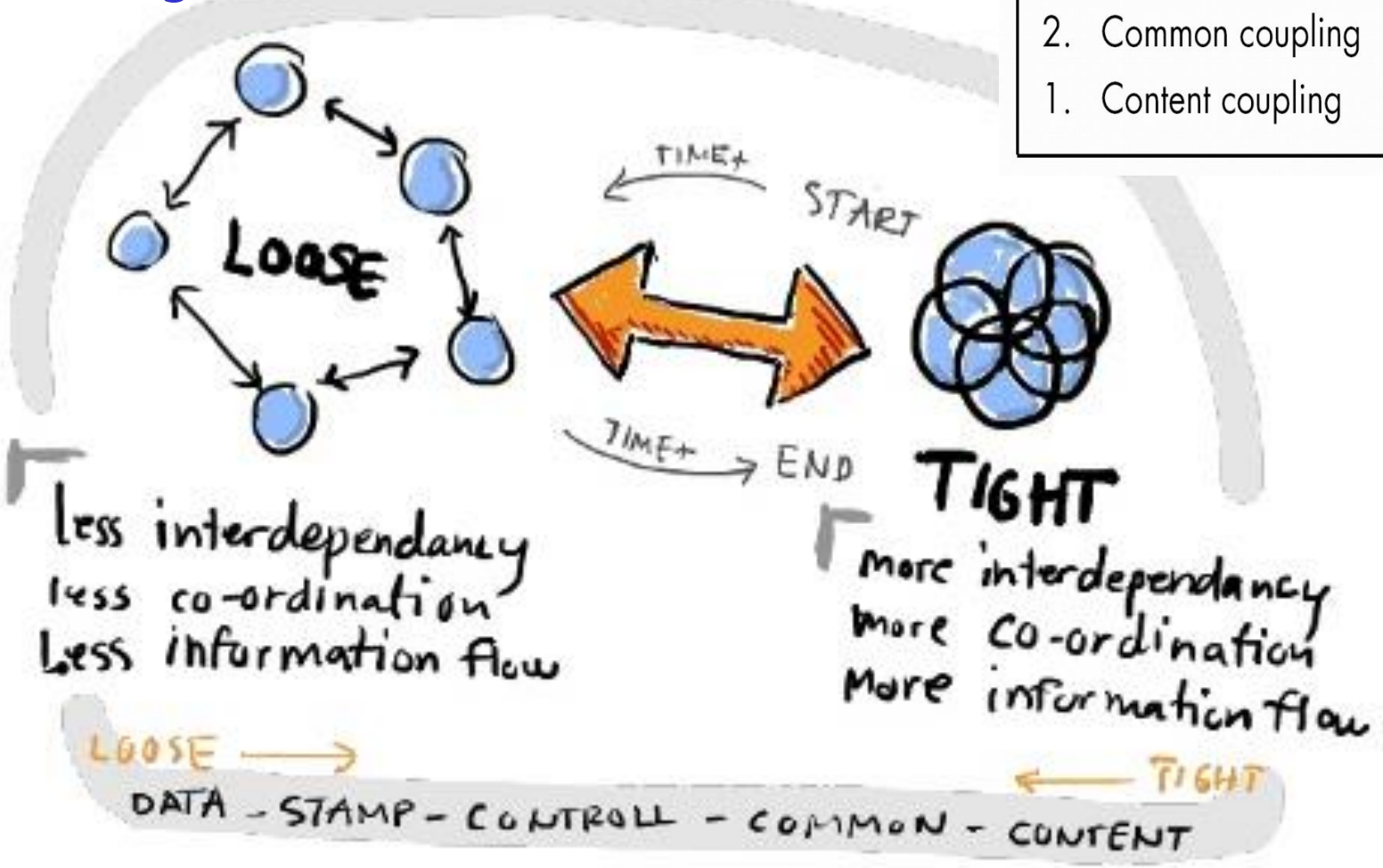


Five levels of Coupling

Couplings: Interactions *between* modules

- ***Loose* -> good for maint/reuse**
- ***Tight* -> bad for maint/reuse**

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

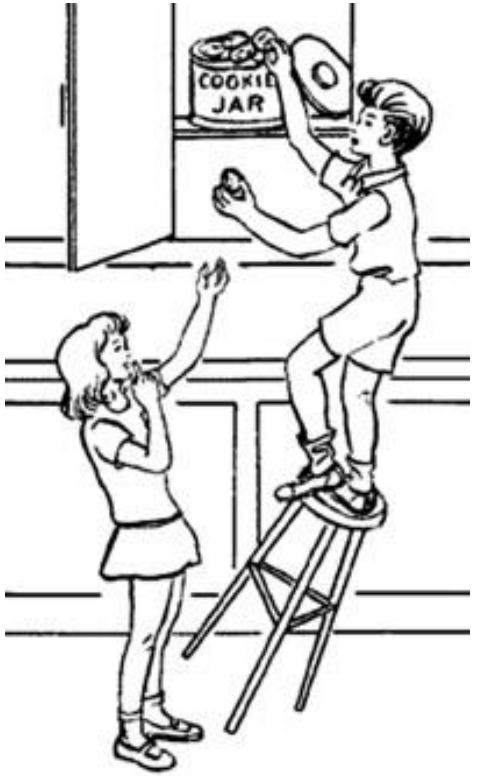


Content Coupling

- ***(1) Content** coupling* - one module directly references the contents of the other.

```
public class Multiplier {
  public static int multiplier = 1;
  public static int multiplierOf(int x) {
        return (x* multiplier); }}

public class MultiplierUser {
  int user() {
    Multiplier.multiplier=3;
    return Multiplier.multiplierOf(2));}}
```



- Class `MultiplierUser` *modifies* the value of `multiplier` directly so they are *content-coupled*
- Other examples?

(2) Common coupling

- Modules directly access same data. Make use of shared variables and read/write to the shared variables.
  - `sumOfTran()` sums shared arrays and thus is common-coupled with any other modules referring to these arrays.

```
void sumOfTran() {
  DB[0]=0; CR[0]=0;
  for (short i=1; i<= numAcct; i++){
    DB[0] += DB[i]; CR[0] += CR[i];
  }
  return;
}
```

Other examples: global variables.

- Results in code with poor readability, generally a hack to circumvent scoping/visibility issues

(3) Control Coupling

- ***Control** coupling* - One module tells another module what to do via the info it passes to the calling module.

```
short mySwitch(char opCode, short x, short y){
  switch (opCode){
    case '+' : return (x+y);
    case '-' : return (x-y);
    case '*' : return (x*y);
    case '/' : return (x/y);
    default: return 0;}
}
```

- mySwitch()'s caller passes a control flag and is control coupled with mySwitch().
- Why is this so bad?

The two modules are *not* independent

- Caller must be aware of mySwitch's internal structure.
- if mySwitch is altered during maintenance, Caller must be made aware of the changes.

(4) Stamp Coupling

- Data structure argument, but callee only ever operates on just part of that data structure. Example,

```
short sumOfFirstTwo(short number[]) {
  switch (number.len) {
    case 0: return 0;
    case 1: return number[0];
    default: return (number[0]+number[1]);
  }}
```

- sumOfFirstTwo() operates only on first two elements of array parameter and is thus stamp-coupled with the caller.

  - Passing entire array when only one or two cells are (at most) needed
  - Would rather *not* have module able to access entire array if it is only supposed to need a single cell of array

(5) Data Coupling

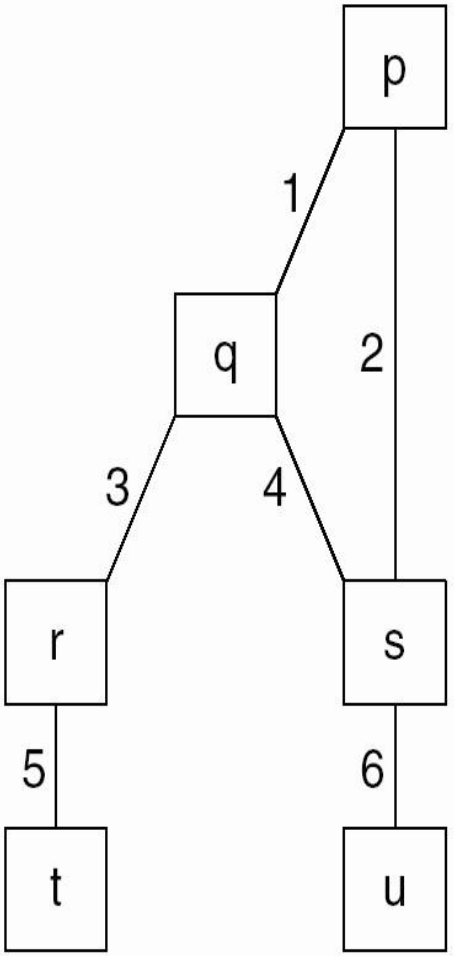
- Arguments are either
  - a simple argument (integer), or
  - a data structure in which **all** elements are used by the callee
- Also: a class accessing *it's own* data members.

```
short sumOfArray(short number[]){
  short result = 0;
  for (i=0; i< number.len; i++)
    result += number[i];
  return result;
}
```

- `sumOfArray()` accesses every element parameter, so is data coupled with caller

- With data coupling, changing module is less likely to cause fault in calling module, results in easier maintenance.
- Should strive for Data Coupling in OO software development

ICE: Determine Couplings given Interface descriptions



Example: 1. When p calls q's interface, p passes one argument, an aircraft type, and q passes back a "status flag".

p, t, and u access the same database in update mode

Number	In	Out
1	aircraft type	status flag
2	list of aircraft parts	—
3	function code	—
4	list of aircraft parts	—
5	part number	part manufacturer
6	part number	part name

5. Data coupling (Good)
4. Stamp coupling
3. Control coupling
2. Common coupling
1. Content coupling (Bad)

Conditions for software development with reuse

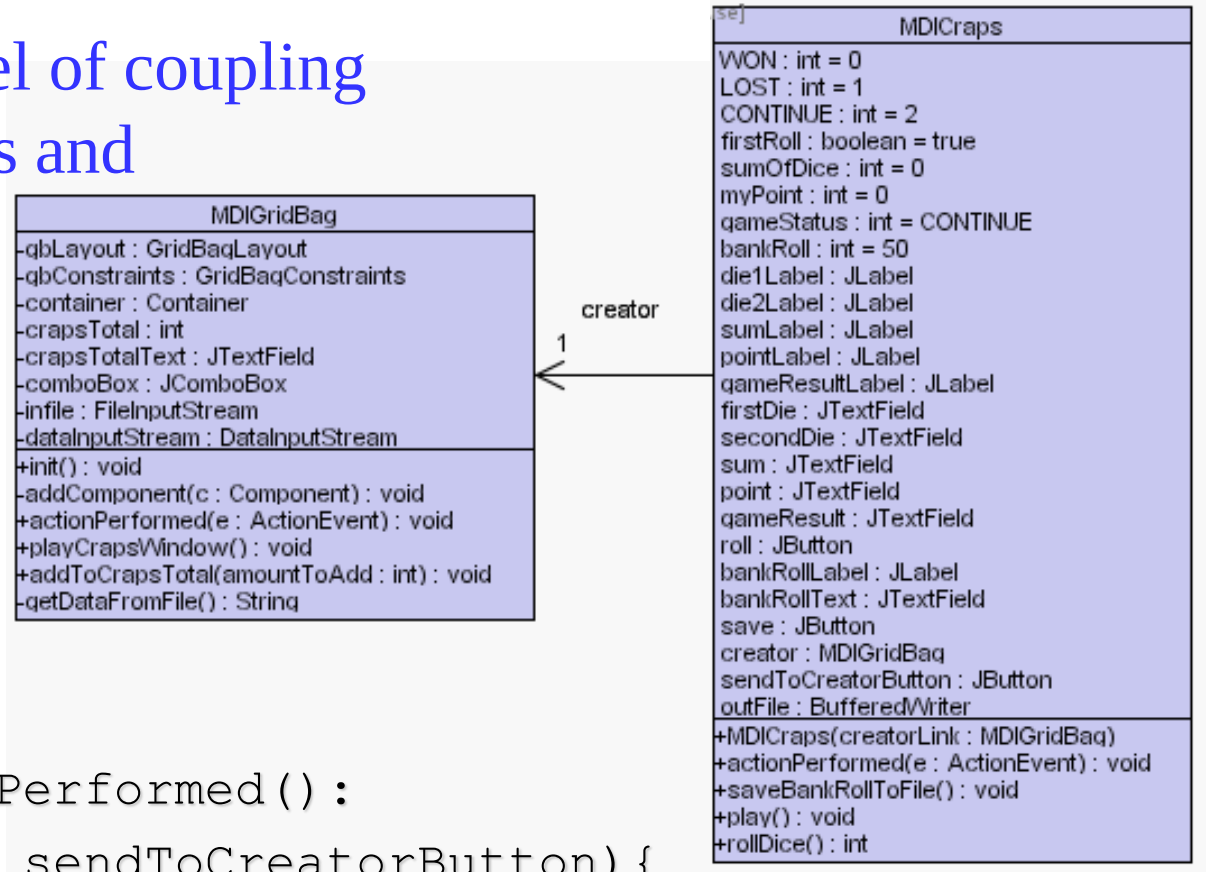
- Must be possible to find appropriate reusable components.
  - Re-user must have confidence that the components will behave as specified and be reliable.
  - Component documentation to help re-user understand and adapt them

- Possible adverse affects of inheritance on reuse?
  - With inheritance, code not collected together in one place
  - Reuse of improper inheritance can lead to extra, unwanted functionality.
  - OOP has a lot going for it, but inheritance has not proven to be a silver-bullet answer to re-usability.

Consider our Craps Software

- **ICE:** Evaluate level of coupling between MDICraps and MDIGridBag

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)




```
//In MDICraps actionPerformed():
  if(e.getSource()== sendToCreatorButton){
     creator.AddToCrapsTotal(bankRoll);
     bankRoll = 0;
     bankRollText.setText
        (Integer.toString(bankRoll));}}
```

What if:

```
creator.crapsTotal+=bankRoll;
```