Recommended books:

Agile Software Development: Principles, Patterns, and Practices

The clean coder

Code complete 2

Metrics:  
- modularity means split software in components having A)high cohesion (module's operations are closed to each other)  
(best - functional cohesion ; communication and sequential cohesion ; coincidental cohesion/logical cohesion are worst)  
and B) low coupling - classes don't have to know to much against the rest (communication interface usually is involved). This allows reusability and maintanance without too much concerns.

1.SOLID

S - single object responsability (e.g. StatusReport print example, separate the data from presentation).  
O - software entities are open for extension but closed for modification  
L - liskov substitution principle:   
\* prooves that you can't test in isolation you have to test with client setup to make sure this is correct.  
\* gives you hint if deriving a type (T) from base (S) is the best way to go or not. If you can't substitute at any time base with deriving and   
keep the same behavior (see the rectangle / square example) then you should NOT use inheritance.  
Instead, use aggregation of types or just make a common class for both called I and implement the problematic operations in both,  
while I would keep the properties (behaviors) that would hold for both.  
  
\* Design by contract and Meyers:   
- use pre and pose conditions for your methods  
- a derived type (T) which overrides a method could have less restriction on pre condition but must have at least the   
same strength over post codition as the parent. E.g.   
Rectangle::setWidth(int w )  
m\_width = w;  
assert(m\_width == w && m\_height == old.m\_height); // should not modify the height, invariant is that they work in parallel  
A wrong impl of Square which overrides both width and height will modify both.  
  
  
  
I - interface segregation:   
\* If you derive from an interface (IWorker with ops work, eat) then you have a HumanWorker -> IWorker and a RobotWorker -> IWorker, and robot  
doesn't have to eat => we have a fat interface. Ideally, we need to split IWorker in IWorkerThatEat, IWorkerWithRecharge and eventually both  
derived from both IWorker.  
D - depedencies inversion principle  
\*

Details about:

3. Inversion of control

- frameworks

- GUI example

4. Dependency injection review

* It is one example of inversion of control
* It helps you to decouple your app objects from each other

how do you fit together this web controller architecture with that database interface backing when they were built by different teams with little knowledge of each other. E.g java containers with Spring framework (abstraction layers for web, jdbc, transactions and persistence):: https://www.tutorialspoint.com/spring/spring\_architecture.htm

Full tutorial here: https://martinfowler.com/articles/injection.html

1. Movie finder example, movieLister depends both on the interface and implementation. What if someone wants to plugin his own impl ?An Sql reader not a txt for instance.
2. Inversion of control (IoC) through Dependency injection

Objective:  ensure that any user of a plugin follows some convention that allows a separate assembler module to inject the implementation into the lister.

-------

Similarly to AOP it allows you to define method interceptors and pointcuts to cleanly decouple code that implements functionality that shoul de separated.