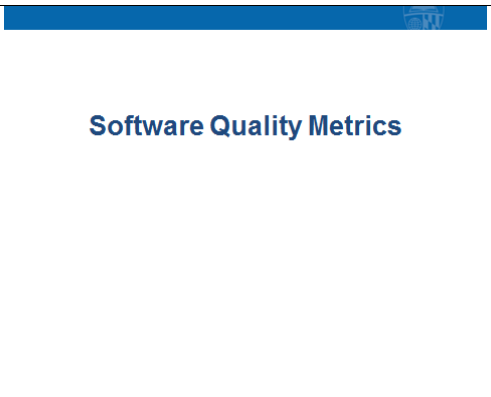
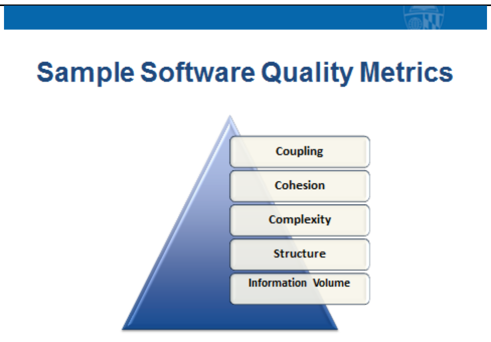
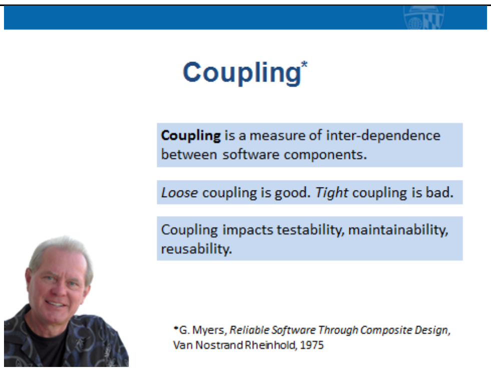
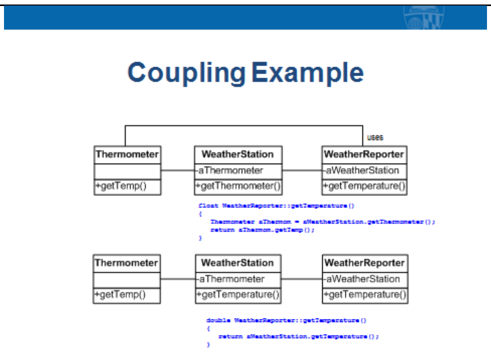
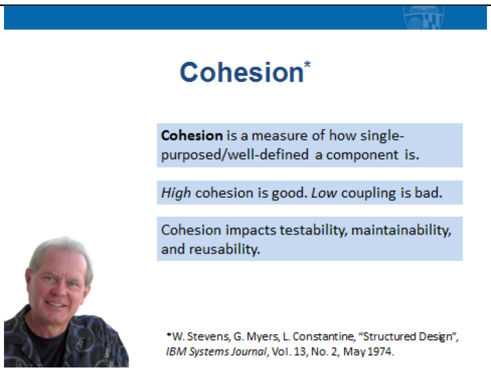
1	 Software Quality Metrics	
2	 Sample Software Quality Metrics — Coupling; Cohesion; Complexity; Structure; Information Volume	In this lecture, I'll continue our discussion of specifying & measuring software quality by illustrating several software quality metrics. I'm going to discuss several commonly used software quality metrics. These metrics are used in the design and coding phases to help build software products that have good quality characteristics…as we saw in the testability example.
3	 Coupling* — **Coupling** is a measure of inter-dependence between software components. *Loose* coupling is good. *Tight* coupling is bad. Coupling impacts testability, maintainability, reusability. *G. Myers, *Reliable Software Through Composite Design*, Van Nostrand Rheinhold, 1975	A very important software engineering metric that has been around for a long time…and is still very relevant today…is coupling. It's an example of a classification metric. Coupling is a measure of inter-dependence between software components. Any components that interact with each other are coupled to some degree. The more dependent components are on each other, the more they know about each other's inner workings…the tighter the coupling. The less dependent they are on each other, and the less they know about each other's inner workings…the looser the coupling. Loose coupling is good. Tight coupling is not good. In practice, we want our designs to be as loosely coupled as possible. In object-oriented software engineering coupling occurs as the result of inheritance, message connections, and containment. We can control coupling in class design by careful use of inheritance, avoiding unnecessary message connections and unnecessary containment, and by using strong information hiding. In traditional, non object-oriented software

		engineering coupling occurs through the use of global data and shared memory. We can control coupling in component design by avoiding the use of global data, avoiding passing memory address information, using minimal number of data parameters, and by passing parameters using copies of data values. Coupling impacts testability, maintainability, and reusability. It can also impact reliability. Let’s take a look at an example.

4	 **Coupling Example**
	 uses
	Thermometer
	+getTemp()
	WeatherStation
	-aThermometer
	+getThermometer()
	WeatherReporter
	-aWeatherStation
	+getTemperature()
	Thermometer
	+getTemp()
	WeatherStation
	-aThermometer
	+getTemperature()
	WeatherReporter
	-aWeatherStation
	+getTemperature()
	`float WeatherReporter::getTemperature()`
	`{`
	`Thermometer aThermom = aWeatherStation.getThermometer();`
	`return aThermom.getTemp();`
	`}`
	`double WeatherReporter::getTemperature()`
	`{`
	`return aWeatherStation.getTemperature();`
	`}`
		Suppose we need to design an application that periodically reports temperatures. And let’s suppose we came up with the following design. We have a Thermometer class that is used by a WeatherStation class to get periodic temperature readings. The WeatherStation class maintains a reference to a Thermometer, and its getThermometer method returns a reference to its thermometer. We also have a WeatherReporter class that is responsible for reporting the temperatures. The WeatherReporter maintains a reference to the WeatherStation class, and its getTemperature method instantiates a thermometer, assigns it to reference the thermometer the WeatherStation maintains a reference to, and then calls the Thermometer class getTemp method to get the temperature. The Weather reporter class is coupled to both the WeatherStation and the Thermometer class…and the coupling to the Thermometer class really shouldn’t be necessary.
		
		Let’s look at a design that improves the coupling…in fact, that eliminates the unnecessary coupling altogether.
		
		In this design, the WeatherStation maintains the Thermometer reference and provides a getTemperature method that returns the current temperature to any clients. The WeatherReporter class simply invokes the WeatherStation getTemperature method when it needs to. Now, the WeatherReporter class is not coupled to the Thermometer class at all.

5



Cohesion*

Cohesion is a measure of how single-purposed/well-defined a component is.

High cohesion is good. *Low* coupling is bad.

Cohesion impacts testability, maintainability, and reusability.

*W. Stevens, G. Myers, L. Constantine, "Structured Design", *IBM Systems Journal*, Vol. 13, No. 2, May 1974.

Another important classification metric is cohesion. Cohesion is a measure of how single-purposed or well-defined a software component is. The more single-purposed or well-defined a component is the higher its cohesion. The more multi-purposed or ill-defined a component is the lower its cohesion.

High cohesion is good. Low cohesion is not good. In object-oriented software engineering, cohesion is applied at the class level as well as at the method level.

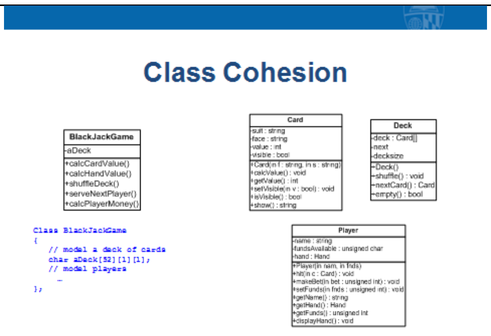
At the class level, a highly cohesive class will represent easy-to-understand things or concepts from the problem domain or solution domain. A class with low cohesion will be harder for a developer to describe to another person…including another developer or other stakeholder.

At the method level, a highly cohesive method will do a single thing. A method with low cohesion will do multiple things or things that are unrelated.

Cohesion also impacts testability, maintainability, and reusability.

Let's look at an example.

6



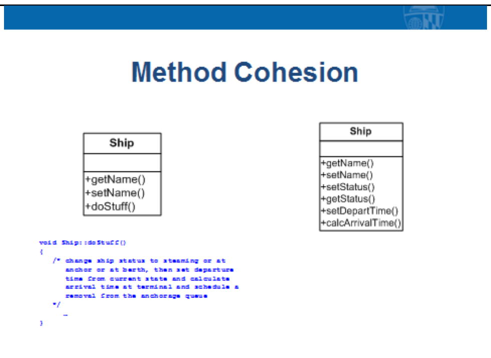
Class Cohesion

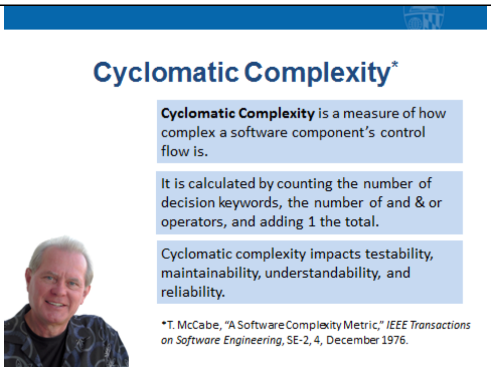
```
Class BlackJackGame
{
   // model a deck of cards
   char aDeck[52][1][1];
   // model players
     …
};
```

Here's an example that illustrates class cohesion. Suppose we were designing software to implement a blackjack card game.

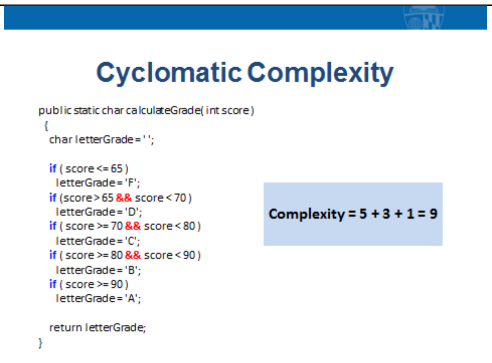
In this design, all the components for the blackjack game are modeled using a single class. The class name is okay, but it's doing to much…it has low cohesion. For example, it is taking responsibility for modeling a deck of cards, modeling the playsers, shuffling the deck, calculating the value of a player's hand, and so forth.

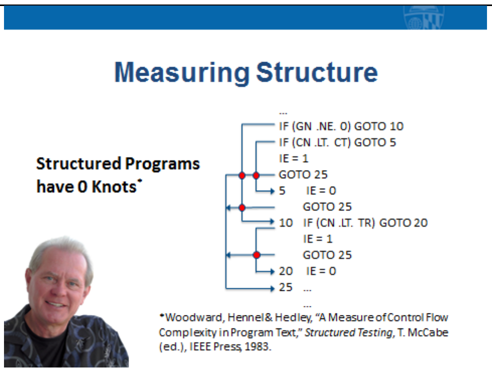
A better design would model the different aspects of the game as separate classes. We'd have a Card class to model a playing card, a Deck class that would hold a set of cards, and so forth. Each class has high cohesion…each class represents a clear component with well-defined responsibilities.

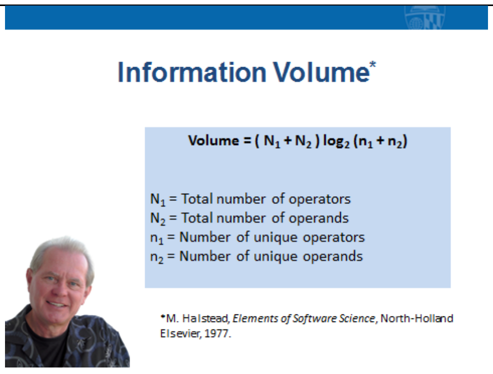
7	
	Method Cohesion
	 Ship
	+getName()
	+setName()
	+doStuff()
	Ship
	+getName()
	+setName()
	+setStatus()
	+getStatus()
	+setDepartTime()
	+calcArrivalTime()
	`void Ship::doStuff()`
	`{`
	`/* change ship status to steaming or at anchor or at berth, then set departure time from current state and calculate arrival time at terminal and schedule a removal from the anchorage queue */`
	`…`
	`}`
		Here's an example that illustrates method cohesion. Let's suppose we're building an application that simulates ships at a port. A natural class for such an application would be a Ship class.
		
		Look at the names of the methods in this class. What the getName and setName methods do are easy to understand. They each do a single thing…and they have high cohesion. But, what about the doStuff method…what does it do? We get absolutely no clue from its name. If we look inside the method, it's doing a whole bunch of things…and it has low cohesion.
		
		A better design would be something like this. There are more methods, but each one performs a single, well-defined function…each has high cohesion.

8	 **Cyclomatic Complexity*** **Cyclomatic Complexity** is a measure of how complex a software component's control flow is. It is calculated by counting the number of decision keywords, the number of and & or operators, and adding 1 the total. Cyclomatic complexity impacts testability, maintainability, understandability, and reliability. *T. McCabe, "A Software Complexity Metric," *IEEE Transactions on Software Engineering*, SE-2, 4, December 1976.	A very well-known metric that has been around for some time now is cyclomatic complexity. Cyclomatic complexity is a metric that measures how complicated the control flow logic for a software component is…where a component is the equivalent of a function, or a subroutine in traditional programming languages, and a class method in object-oriented software engineering. Cyclomatic complexity can be easily calculated from pseudo-code or actual code. The formula is simple…complexity is equal to the number of decision keywords plus the number of ands and ors plus 1. Decision keywords correspond to language constructs that would cause control flow to depart from a straight sequential flow…keywords like if, while, do-while, for, switch, case, go to. Cyclomatic complexity impacts testability, maintainability, understandability, and reliability. McCabe and others have reported that components with very high complexity are likely to be problematic…error prone, require more testing, and are harder to understand and change. Software engineers have historically used 10 as the upper bound for complexity when designing components. I can tell you from my own work that cyclomatic complexity is correlated to the things I just mentioned. A project that I worked on for a software vendor client invloved analyzing about a million lines of code for several of the company's software products. Those products that had high cyclomatic complexity components were by far the most

		problematic components for my clients.

9	
	Cyclomatic Complexity
	`public static char calculateGrade( int score )`
	`{`
	`char letterGrade = ' ';`
	`if ( score <= 65 )`
	`letterGrade = 'F';`
	`if (score > 65 && score < 70 )`
	`letterGrade = 'D';`
	`if ( score >= 70 && score < 80 )`
	`letterGrade = 'C';`
	`if ( score >= 80 && score < 90 )`
	`letterGrade = 'B';`
	`if ( score >= 90 )`
	`letterGrade = 'A';`
	`return letterGrade;`
	`}`
	Complexity = 5 + 3 + 1 = 9
		Here's an example of calculating cyclomatic complexity for a method written in Java programming language. The method, named calculateGrade, accepts an integer test score and calculates the associated letter grade, passing it back to whatever component called it.
		
		The decision keywords in this example are all if-statements. There are 5 of them and I've highlighted them in blue. This method also contains three logical "and" operators that I've highlighted in red. So…for this method…the complexity is 9…5 decision keywords plus 3 "ands" plus 1.

10	
	Measuring Structure
	Structured Programs have 0 Knots*
	`...`
	`IF (GN .NE. 0) GOTO 10`
	`IF (CN .LT. CT) GOTO 5`
	`IE = 1`
	`GOTO 25`
	`5     IE = 0`
	`      GOTO 25`
	`10    IF (CN .LT. TR) GOTO 20`
	`      IE = 1`
	`      GOTO 25`
	`20    IE = 0`
	`25    ...`
	`...`
	*Woodward, Hennel & Hedley, "A Measure of Control Flow Complexity in Program Text," *Structured Testing*, T. McCabe (ed.), IEEE Press, 1983.
		There are a number of metrics that have been used in software engineering that measure whether a software component like a function or subroutine is structured…that is, whether it follows the rules of structured programming.
		
		One such metric is the program knots metric, which I've illustrated graphically here. There are a set of formulas that can be used to make this calculation, but they are complicated…and for our purposes the visual example will suffice. And, in practice, tools exist that calculate the metric.
		
		A "knot" exists whenever one control flow path crosses over another. In this example I wrote a little Fortran code because it's really easy to write unstructured code with that language. I traced the control flow paths with arrows, and wherever one control flow path crosses over another I drew in a little red circle corresponding to a knot. This code has four knots so it is not structured.
		
		Unstructured code also impacts testability, maintainability, understandability, and reliability. The more unstructured a code component is…the more difficult it will be to test, maintain, and understand…and the more error-prone it's likely to be.

11	 **Information Volume*** **Volume = $(N_1 + N_2) \log_2 (n_1 + n_2)$** N_1 = Total number of operators N_2 = Total number of operands n_1 = Number of unique operators n_2 = Number of unique operands *M. Halstead, *Elements of Software Science*, North-Holland Elsevier, 1977.	One more example of a computational metric is Halstead’s Information Volume metric. It’s how much information is contained in a software component based on a count of operators and operands, as illustrated here. Halstead found that this metric impacts the amount of maintenance and unit test effort. For a specific component, the metric just computes a number. The number by itself is not very revealing…it must be correlated to other statistics such as hours to test or hours to debug in order to make it useful in practice.