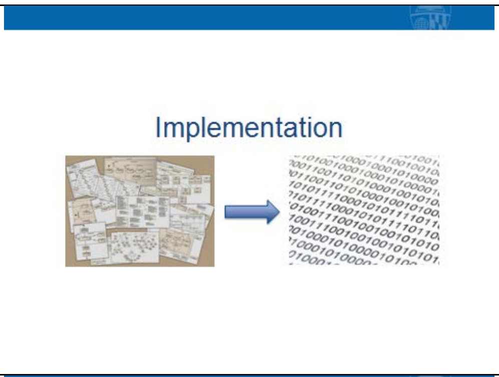
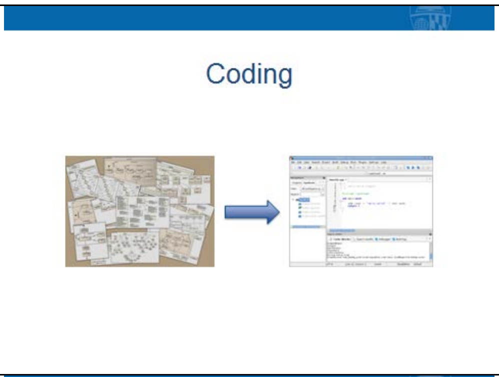
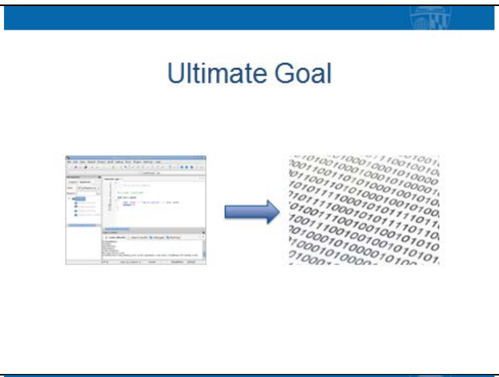
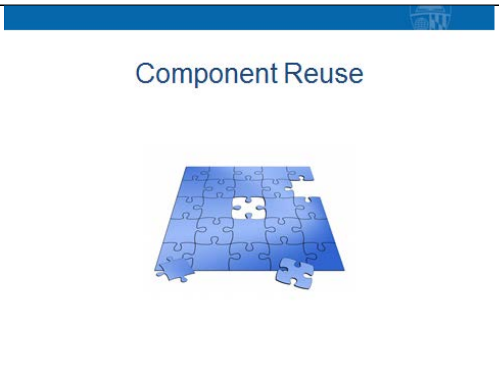
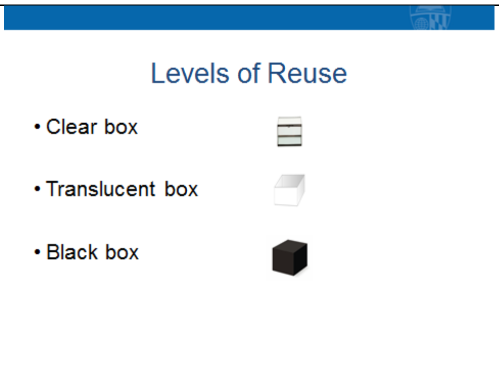
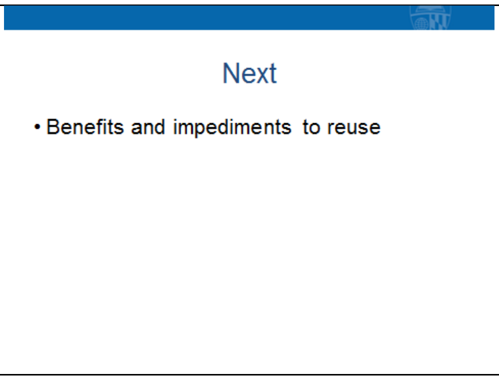
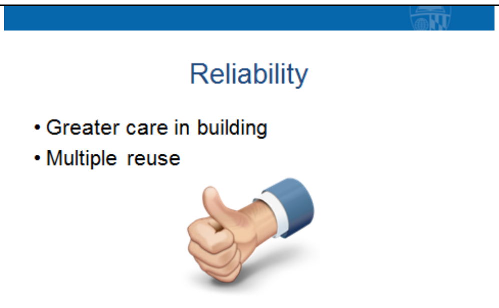
1-1	 Implementation	There are two more parts to this module. In this video and the next two, we will be discussing the implementation of the code – the translation of the design diagrams into runnable code. The last part of the module consists of one video on the subject of maintenance.
2	 Coding	Normally, we think of implementation as the translation of the design models into code using some sort of programming language, an integrated development environment, and other programming tools. That is, our goal is to write code.
3	 Ultimate Goal	But, of course, our ultimate goal is to produce ones and zeros that go into the computer and control its actions. It turns out that there are two main ways of producing the code. One is to write the code by hand, and the other is to re-use code that already exists.
4	Hand-Crafted Code	Let me use an analogy. A few years ago I remodeled my kitchen. I took everything out down to the bare walls and re-did all of the cabinets, plus electrical and plumbing. There were two ways I could have accomplished this task. Let me talk about the way I didn't go. That would have been to go to the lumber yard and buy a pile of lumber, some nails, glue and so forth. I already had most of the power tools needed – table saw, jig saw, nail gun, etc. I would have had to measure and cut each piece of wood and put the cabinets together piece by piece. I then would have had to sand, stain and varnish all of the wood surfaces. It would have been time consuming. The quality of the results would depend on the quality of the design I was working from as well as my skill as a carpenter. (In my case, not so good.)

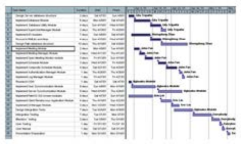
5	Reuse	The method I used was what even most professional carpenters do these days. I went to the building supply store and ordered pre-built cabinets. I used the design that I had drawn to determine how many of each size cabinets to order. I got to choose the type of wood and design of the doors that I wanted. After a few weeks, all of the cabinets were delivered in a big truck. I had to store them in my living room. The boxes took up the entire room. Installing the cabinets was fairly easy. You just had to make sure they were level and plum, and fit together evenly. Each cabinet only needed a few screws into the studs to hold it in place. The quality of the workmanship on the cabinets was superb. Much better than I could have done on my own. The hinges were even already installed on the doors. The finish on the wood was beautiful. The time to complete the job was a fraction of the time the stick built approach would have taken, and the quality of the result was better. So, as I said, this is an analogy to software implementation. The stick built approach to cabinet making is analogous to hand crafting code line by line. The approach of constructing the kitchen out of pre-built cabinets is analogous to implementing code by reusing pre-built components.
6	Defect Prevention	Let me pose a question. In software, where do the defects come from? The original bug, in the ENIAC computer was a real insect that had crawled into one of the relays. But in software today, the bugs don't crawl into the software. They are put there by the programmers. And I might say, we're pretty good a putting them in there. So, if bugs are built into our software when we program it, how can we avoid them? The answer is: don't program. That's right. Don't write software line by line. That is what is error-prone, and leads to bugs. Instead, build software out of trusted, well designed, tested, quality components.

7	 Component Reuse	Component reuse involves some hand-crafted software plus some reused pre-built components. Using a building analogy again, the components are like bricks, and the hand crafted code is like the mortar that is used to cement them together. The bulk of the wall is made up of prebuilt bricks, but the overall shape of the wall is determined by the way in which the bricklayer puts the bricks and mortar together. The components in component reuse have well defined interfaces, so we know how to make use of them in our code. We know how to invoke their services, and we know what those services do. We may or may not know how they work. Recall our discussion of encapsulation. It isn't necessary that we know how they work.
8	Framework Reuse	There is another type of reuse, called framework reuse, which is sort of the exact opposite of what we just described. What we reuse is partially built software, with a few holes in it that we need to fill in. The framework will provide the specification of the components that we must build to fill in the holes in the framework. Think of this style of reuse as tailoring. What we are reusing is the bulk of an application, but we tailor the details by inserting our own custom details. An example of this might be an Internet storefront application, similar to the one we are using as an example in this course. Most of the capability of one these systems is standard, and can be used by almost any kind of web store, regardless of what they are selling: books, toys, food, software, hardware … anything. So, for example, the shopping cart capability would be a standard part of the framework. The tailoring would amount to plugging in your own database of items to be sold, access to your inventory subsystem, and access to your accounting subsystem. The framework would specify the interface required for it to communicate with each of these components. We just have to write the components to fit into the holes in the framework specified by the interfaces. This way of looking at reuse is called inversion of control. In component reuse, the code that we write is in control. In framework reuse, the code that we reuse is in control.

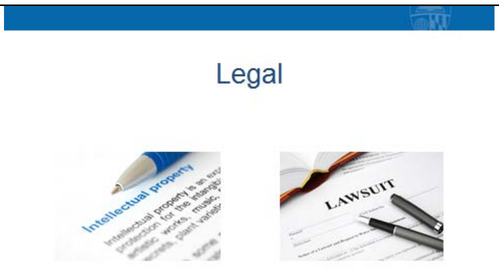
		Sometimes this is called the "Hollywood Principle." -- "Don't call us. We'll call you."
9	 Levels of Reuse • Clear box • Translucent box • Black box	How much do we have to know about the software that we reuse? There are three levels of this. In black-box reuse, we can't see the code. The only thing we have access to is the specification of how to use the reusable software. We must rely on good documentation of the reusable code in order to use it properly. The advantage is that if a new version of the reusable software is created, it is rather easy to swap out the new for the old. The disadvantage is that it is hard to find reusable software with good documentation. Clear-box reuse is the opposite. Not only can we see the code, but we can make changes to it. The advantage of this is that you can read the code and figure out what it does. You can also tailor it to your own needs. The down side of this is that if you change it, you own it. If they come up with a new version of the reusable software, you can't just swap the old out for the new. You'd have to tailor the new version as well. Translucent box reuse is a combination of these two. You can see the code, so you can figure out how it works and how to use it, but you can't change it. This gives you the advantages of both black box and clear box reuse.
10	 Next • Benefits and impediments to reuse	In the next two videos, we will talk some more about the benefits and the impediments to reuse.

2-1	Benefits of Reuse	It is our contention that software reuse – building software out of reusable pieces, whether by component reuse or framework reuse, is a good thing. We will explore some more of the benefits of reuse in this video.
2	 Reliability • Greater care in building • Multiple reuse	Reliability is an aspect of software quality, as we will see later in this course. Basically reliable software is software that will run a long time without failing. And when it does fail, we can easily fix it. Software reliability, as well as hardware reliability, is measured as "mean time to failure," or MTTF. This is also sometimes called Mean Time Between Failures (MTBF), but this, I think, is a more pessimistic way of looking at it; sort of like it fails frequently. There are two main reasons why reused software has higher reliability than hand-crafted software. One is that, if I am building some software that I know will be reused by other people; I will take more care to do a good job. You know it's going to get a lot more attention than other stuff you write, so you try to do a good job. That includes doing a good job of designing, coding, testing, documenting, and advertising it to make it available to others. This extra care and attention should result in higher quality software. The other reason reused software has higher reliability is because of something we will cover in our module on assuring software quality. As we said, reliability is measured as mean time to failure. In a testing situation, the average time software will run during testing is related to the MTTF. The more we run the software without it failing, the higher the reliability. So, if we test the software for an hour and it doesn't fail, then we have a data point for measuring the reliability. If it runs for two hours, the reliability goes up. If we run it for two days, even better. The longer we run the software without failures occurring, the higher the reliability. If we test the software for twice as long, we will get much better reliability. Makes sense, right? Run it ten times as long, even better. Think about this: What if we took ten copies of the software and ran them on different test cases all at the same time. The effect would be the same as

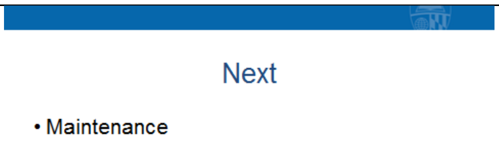
		running ten different test cases on one copy of the software, one after the other. In other words, we can speed up the testing by running multiple tests in parallel. Now suppose, we have some software that is being reused several places. It is running well, without failure in all of those implementations. That would have the same effect as the parallel testing to improve its reliability. In fact, because the multiple instances of the software are being reused in different environments, this is even better proof of the reliability. Here's a corollary: Don't every buy model one point zero of anything. The reliability will be much better on version 2 or higher.
3	Less Expensive 	The obvious thing that people think about when they hear about reuse is the cost savings. It's true. Even though the pre-built cabinets I bought for my kitchen were expensive, the cost of labor to hire the project to be done by a carpenter building the cabinets piece by piece would have been much more expensive. It's almost always cheaper to reuse hardware components than it is to custom design and build our own. The same goes for software. It's cheaper, and it's usually quicker to get your system working. And, as we know, time is money.
4	Capture Expert Knowledge 	There are a couple of other benefits to reuse. Remember the old player pianos. They were run by piano rolls – rolls of paper with holes punched in them. As the paper ran past a read head, sensors would detect the holes and play corresponding notes on the piano. They were actually a precursor to punched tapes and cards for computers. Who do you think they got to record those holes in the master roll? Not some average piano player. No, they got the best piano players around. I'm including in the student notes a link to a You tube video of a piano playing a piano roll recorded in 1921 by Ignace Jan Paderewski, one of the greatest piano players and composers in the nineteenth and twentieth centuries. https://www.youtube.com/watch?v=Vf11cO2WqJY

		By the same token, who would we get to write the reusable software? The world experts. For example, who do you think developed the swing components for Java? How about the google map software framework that is reused in a lot of applications?
5	Standardization	Another benefit from reuse is standardization. If everyone uses the same components or frameworks, all of the applications should be similar. You can see this standardization in video games. Player movement by running and jumping is sort of standard across multiple games. These are sometimes called platform games. Once you learn how to move the player in one of these games, it works pretty much the same way in the others. It's not hard to imagine that the same underlying code is used for multiple games from the same manufacturer. Multiple applications that use the same UI toolkit can benefit from this sort of standardization of the user interface, making it easier for users to learn how to use the software, or remember how to use it after some time away from it.
6	Less Development Time 	We pointed this out before, but due to the fact that it's easier to reuse software than it is to create it by hand, there is less schedule risk. The only thing is we can't necessarily plan on reusing software. We don't often do the search for the reusable software until when we are doing the design. But, if we do our planning assuming we will be hand crafting the software, we can get a schedule benefit if we are, in fact, able to find reusable software instead of writing it ourselves.
7	Next • Impediments to reuse	Of course, there's the other side to the coin. We'll look at some of the impediments to reuse in the net video.

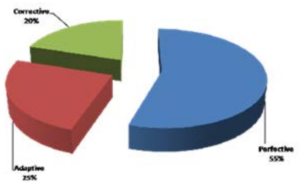
3-1	Impediments to Reuse	If software reuse is so great, why don't we all practice it more? There are some impediments that sometimes get in the way of reusing software. If we can work these out, then we might see more software reuse.
2	Creating Components	Internal reuse is where one part of an organization creates reusable software, which is then reused by another part of the organization. Usually this software is not created expressly to be reused, but is harvested from existing applications. As part of the review process, candidate parts of applications are identified as possibly reusable. Someone has to take these software parts and sort of polish them up to make them more reusable. Sometimes the organization that does this is called a software foundry. Their job is to harden the software to make it more reusable. Sort of like what a blacksmith does to steel in a foundry. There are two main types of changes the foundry will make to the software. One is to remove parts of it that make it too specific to the application for which it was written. The other is to add capabilities that were not needed for the original application, but will make it generally more reusable. The problem is, most people don't know how to do either of these things. Usually a specially trained part of the organization is needed to do this job. It shouldn't be left up to the original developers. If this is not done right, the reuse effort will die on the vine due to a lack of reusable software.
3	Who Pays?	Then there's the question about who pays for this extra work to make the reusable software. The manager of the organization that originally developed the application won't normally have much incentive to spend extra effort that will benefit some other part of the organization. There has to be motivation from the top down in order to make this happen. This usually involves funding for reuse, including the establishment of the software foundry that we talked about.

4	NIH 	Even if reusable software exists, most people will resist using it. Most likely it isn't an exact fit. Some design changes might be necessary to allow for the reusable software to be employed. If the choice is between coding it myself or reusing someone else's code, the code I write myself is almost always better than the reused code in terms of being an exact fit to the design. However, it's a question of "Is good enough good enough?" I think there one of those Pareto effect things going on here. An 80-20 rule. For eighty percent of the software, it doesn't matter if it isn't perfect. Let's focus on the 20 percent that does matter. Rather than spending our effort on all of the software, thus diluting our effort on the critical 20 percent, let's focus our efforts on the 20 percent that matters, and implement the rest with reusable software.
5	 Legal 	Legal issues will be showstoppers. There are two main kinds of legal problems with reusable software. One is intellectual property rights. Say I write some reusable software. Someone else in the organization reuses it, and their application is very successful; making a lot of money for the company. They get a big bonus and a promotion. Am I entitled to any of that, since part of their application was actually written by me? I don't know. These things ought to be worked out in advance. This is why we have lawyers. The other issue is liability. This will completely stop any reuse effort unless we can fix it. So I write some reusable software. It is reused by another organization. Their application ends up causing loss of money or information, or injury or death to people. Who do you think the lawyers are going to come after? I think we have this issue pretty much settled as an industry. It is why we always have to agree to the licensing contract before we can use almost any software. A standard part of these contracts is always that we agree to assume all risks in using the software.

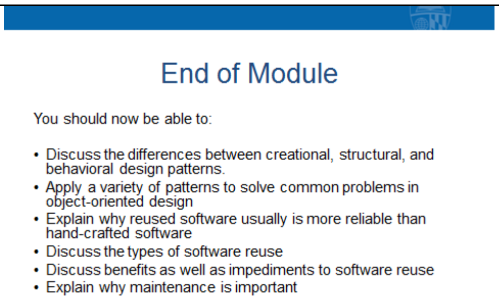
6	Trust 	A lot of time, we can search for reusable software on the Internet. The big problem with this is how do we know it's any good? So I do a search, based on keyword criteria. Say I find half a dozen candidate pieces of software that I might be able to reuse in my application. How do I decide which, if any, are not full of bugs or worse yet, viruses? I certainly can't take the developer's word for it. I could spend time testing all of the candidates, but this is tedious and may not find all the defects or viruses anyway. There are two models in the real world for how we could handle this. One is what I call the Consumer Reports model. Consumer Reports is an organization that independently tests products in their own laboratories and publishes the results in their magazine. Consumer reports magazine doesn't accept advertising, so what you get is unbiased reports of their test findings. Using this model for software, we could think of having an independent organization that would test reusable software and report on the results. Everyone would benefit from this by not having to do the testing themselves. The other model is user feedback. Say you're looking for a movie to watch. Or maybe you want to buy a new printer. There are lots of choices. How do you decide? Well, fortunately there are ratings on these things provided by other customers. So you look at the ratings on the various movies. On some sites, you can see ratings by professional movie critics, as well as ratings by other movie viewers like yourself. After you've seen the movie, you can add your own review. Same for the printer. You can look up how well people who bought each printer liked it. They even allow for the reviewers to put in comments that can help you decide. We can do this with software. In fact, this already exists. Look at Play Store for Android apps, and the App store for Apple apps. You can search for apps based on keywords. When you find candidates, you can look at the ratings and comments from other people who have already downloaded and used the

		apps.
7	 ## Next • Maintenance	That’s it for our treatment of software implementation. Once the software is implemented and tested, it is delivered to the end user. That’s not the end. It’s just the beginning of the rest of the life cycle of the software. As a matter of fact, only a fraction of the total life cycle of software is the development. The rest of the life cycle is the usage of the software by the end users. Hopefully this is a long time. In order to ensure this, we need to do maintenance on the software to keep it up to date. That’s our next topic.

4-1	Software Maintenance 24 HOUR SERVICE 7 DAYS A WEEK INCLUDING HOLIDAYS!	This video is about software maintenance. What is software maintenance? Is it like car maintenance? We have to do maintenance on a car, like changing the oil every five thousand miles or so, because cars are mechanical and things wear out and need to be replaced. Software doesn't wear out. Still, we know that organizations spend a lot of effort maintaining software that they have produced. Let's look at • the reasons why we need to do maintenance, • what it costs, and • what to do about it.
2	The Challenge of Maintenance "Maintenance is a good thing." -- Sam Schappelle 	When I got started in software, I used to think that having to maintain our software was sort of a negative. It was a penalty for making mistakes when we developed the software, and had to fix them after it was delivered. If the customer was initially dissatisfied with the software, we would have to do maintenance on it to make it right. If there were defects in the software we had to fix them. I looked at how maintenance was regarded in my organization (not very highly). Who were the people that were put onto maintenance? New hires were assigned to maintenance so they could "learn the software" before they were allowed to work on new software development. The people who didn't keep up with the newer technologies were not allowed to work on new development, so they were stuck in maintenance. You were always dealing with someone else's code. There was never good documentation. Maybe you were lucky enough to find some contact information. You call the number and as, "Is Marsha there?" They say, "Marsha left the company two years ago." The code might have been written in a language that no one knows any more. It may have been designed to run on machines or operating systems that are obsolete.

		You might not even have the source code, let alone any documentation or test cases. The software may have been thrown together and its structure is so poor that any changes lead to ripple effects that actually make things worse. As I say, I used to think that maintenance was a negative. But at some point I came to realize that maintenance was actually a good thing. Why do I say that? For one thing, consider this: Say we write a piece of software and deliver it to the customer. They never ask for any changes to it. So there is zero maintenance required. Is that a good thing? Well, why do you think they never requested any changes? Most likely because nobody used it. For whatever reason. Probably poor quality. Now that's a real negative. So I realized that there was a correlation between the quality of software and how much was spent on maintenance. The better products actually received bigger maintenance budgets than the poorer ones. In other words, if I were developing software, I would hope that it would have a big maintenance budget because that would mean that it was good software. Why is this? Why does more successful software require more maintenance? We have to look at the kinds of maintenance activity to see why.
3	Kinds of Maintenance 	There are three kinds of maintenance that we perform on software. Corrective maintenance is fixing bugs. It accounts for about 20 percent of maintenance effort. Adaptive maintenance is when we make changes to keep up with new technology such as new hardware or operating systems. This happens all the time. We aren't really changing the application; we're just extending its life. Adaptive maintenance is about 25 percent of all maintenance activity. The third kind of maintenance is what we call perfective maintenance. This means that we are making changes to improve the software. We add new features and change things to make them work

		better. These changes are usually requested by the customers. They are happy with the software and demand more. That’s why I say that maintenance is a good thing. I would **hope** that my customers would be happy enough with my product to ask for more capability. Here’s an analogy. Maintaining software is like upgrading your home sound system. As technology changes, your equipment becomes obsolete. So you replace your speakers with better ones. You add a subwoofer. You add on a better CD player, then an MP3 player. You add Bluetooth speakers for the patio. Sometimes your equipment breaks and you just need to either get it repaired or replace it. You probably know someone who is always upgrading their sound system. Perhaps it’s you. These people have different reasons why they do this. One is that they like using their sound system, but adding new technology will enable them to enjoy it more. Another is that they like to keep up with the latest in technology, maybe to impress their friends, or maybe just because they can afford it. Whatever the reason, they are always making changes to their equipment. That’s the way it is with software. We need to make changes to keep up with new technologies. We need to fix broken software. We need to add new capability that we didn’t even know about when we first wrote the software.
4	Cost of Maintenance 	Well, all this maintenance does cost something. Surveys have shown that 60% to 80% of the total life cycle cost of software is in maintenance. That is, for each $1.00 we spend producing a software product; we spend between $1.50 and $4.00 maintaining it over its lifetime. Another thing to keep in mind is that we are continually building more software that will need to be maintained. And if folks are doing things right, that software will have a long life time. Thus the number of lines of code that will need to be maintained grows very rapidly each year. And there is no stopping this growth. So what are we to do?

5	 Return on Investment	The only thing we can do is to reduce the unit cost of maintenance. In other words, we need to find ways to reduce that 60% to 80% of the life cycle cost of maintenance to some lower figure. Anything we can do to lower the per-year maintenance cost will have a multiplying effect. It's sort of like an investment that will pay back returns for the rest of the life of the software. What can we do to make our software easier and cheaper to maintain? Well, I think this is what we have been talking about the entire semester. If you apply the software engineering principles and techniques we have been discussing in this course, it should have the effect of making it easier and cheaper to maintain our software in the future.
6	 End of Module You should now be able to: • Discuss the differences between creational, structural, and behavioral design patterns. • Apply a variety of patterns to solve common problems in object-oriented design • Explain why reused software usually is more reliable than hand-crafted software • Discuss the types of software reuse • Discuss benefits as well as impediments to software reuse • Explain why maintenance is important	We'll have more to say on this in the next couple of modules, on software quality. But for now, we have come to the end of this module. Here is what you should now be able to do. • Discuss the differences between creational, structural, and behavioral design patterns. • Apply a variety of patterns to solve common problems in object-oriented design • Explain why reused software usually is more reliable than hand-crafted software • Discuss the types of software reuse • Discuss benefits as well as impediments to software reuse • Explain why maintenance is important There is a quiz that you need to take. You should work with your group to discuss opportunities for reuse on the project: reusing design patterns or reusing software components or frameworks.