Module 5: Design & Implementation

Objectives:

1. Understand what object-oriented design is and how it differs from procedural design.
2. Be aware of the existence of several object-oriented design techniques.
3. Know what a design pattern is, and be familiar with at least 2 examples of design patterns.
4. Know what a use case is.
5. Understand why UML was developed, and be introduced to some of the diagrams UML uses.
6. Understand some of the fundamental principles of user interface design.
7. Know why prototyping is especially useful in user interface construction.
8. Understand the importance of using the appropriate means to present information to users.
9. Be aware of some of the common pitfalls encountered in user interface design.
10. Understand that software engineering != writing code.

Reading:

Sommerville, Chapter 7 & Web Chapter 29 (Interaction Design, available on the companion website: http://iansommerville.com/software-engineering-book/files/2014/07/Ch_29-Interaction_design.pdf).

Assignment:

Participate in the group discussion for Module 5.

Quiz:

Take the quiz for Module 5.

Lecture:

The Design Phase

The design phase is where it is determined *how* the software requirements are to be satisfied.

Design includes system modeling and application architecture, but goes much further. For example, the requirements may state that the software should keep track of all employees at an organization. System modeling will determine the boundaries of how users should be able to access employee information, and may determine that all employee information should be stored in a database located on a central server, with nightly backups to an on-site as well as an off-site location. It may be decided that the architecture for the software will be transactional in nature. In the design phase, it will be determined what flavor of database will be used (Oracle, SQL Server, MySQL, etc.), the data types that will be used to store the various pieces of employee information, how the required security measures will be implemented, and so on. As design progresses, requirements that were missed during requirements elicitation may be discovered, questions may arise concerning the feasibility of certain requirements, and other issues may arise, all of which will prompt temporary suspension of at least a portion of the project while these issues are sorted out with the customer. This possibility of change illustrates the major pitfall of the waterfall process model, and supports the presence of the customer during development as prescribed by many of the agile process models.

A complete history of software design is beyond the scope of this course. However, as software engineering has evolved, the notions of structured design and modular programming have proven themselves essential. In this module, we'll talk about several of the most commonly encountered design concepts.

Information Hiding

The principle of information hiding was proposed by David Parnas in the early 1970's [1]. The essential argument is that a module, A, that invokes a method of a separate module, B, should not need to know the implementation details of the method, or of Module B. Rather, it should only need to know what inputs to provide and what type of output is produced. Unrestricted use of global variables is usually a good example of bad information hiding. The public visibility of global variables means they can be modified from anywhere in a program, which may be appropriate for some variables, but not for others. If the value of a variable is supposed to be determined by following some algorithm, modifying the variable directly short circuits that algorithm, and could lead to an unstable system state.

Information hiding is often erroneously confused with encapsulation. Encapsulation entails grouping together a set of related components (e.g., attributes, methods/functions, etc.) into a unit. An object-oriented class is a good example of encapsulation. The confusion with information hiding arises because many of the components that are encapsulated are hidden from public scope. The hidden components are representative of information hiding. However, it is also possible to make all the components publicly visible, or global, in which case the components are still encapsulated, but very little, if any, information hiding exists.

Module Coupling and Module Cohesion

Following the same line of thinking as information hiding are two properties every module has: **coupling** and **cohesion**. I talk about coupling here, and cohesion in the next section. Coupling refers to the degree to which one module interacts with another module. There are six levels of coupling, described below, in order of increasing coupling:

1. uncoupled — the modules do not interact with each other at all
2. data coupling — only individual data units are passed (e.g., simple arguments) from

one module to another

3. stamp coupling — an entire data structure, such as a list, is passed from one module to another

4. control coupling — a "control flag" is passed to allow one module to control the behavior of another module

5. common coupling — the modules access common or global data

6. content coupling — a module directly modifies another module's data or execution branches from one module directly into another module

In general, you should always try to keep the amount of coupling between modules to a minimum. Don't pass an entire data structure if only one specific item in that data structure is needed. Avoid excessive use of global data. Never allow execution to pass from one module directly into another module (e.g., with a GOTO statement). With modern high-level programming languages, content coupling has pretty much been wiped out. Unless you are writing a trivial program, you cannot have zero coupling; *some* amount of coupling is necessary in order to have a functional program.

Cohesion refers to the degree to which a module performs one, and only one, function. Modules that perform multiple functions can be confusing to understand, and can cause unforeseen problems. There are seven levels of cohesion, described below in order of increasing cohesion.

1. coincidental — the various functions of a module are not related; for example, consider a class containing methods that have nothing to do with each other, or with the purpose of the class

2. logical — logically related functions are placed in the same module, although they can have different purposes; for example, an input reader class having functions for reading data from several different types of devices - each function reads data, but the way they read the data will be different

3. temporal — multiple functions are related only through timing; e.g., the functions may be responsible for initializing system attributes on system startup

4. procedural — different functions related only because they must be performed in a certain sequence to accomplish a goal; e.g., functions that read data, validate data, process data, write data - these activities must be done in a certain order

5. communicational — an example of this is when unrelated data are read in together to avoid excessive disk accesses

6. sequential — output from one function of a module is needed as input by another function of the module

7. functional — every function of a module is essential to accomplish exactly one goal, and all essential functions are in the same module; this is the ideal situation

You should always try to achieve maximum cohesion. Doing so minimizes the responsibilities of your modules, and helps to localize faults, which makes debugging easier. From a purist standpoint, it also makes for a more elegant solution.

In summary, the goal is to maximize module cohesion while minimizing module coupling, but

it's a balancing act. Sometimes it's necessary to trade off on one in order to optimize the other.

Object-Oriented Design

Object-oriented design (OOD) focuses on the representation of components as discrete, cohesive, units, commonly called classes. OOD also provides for functional descriptions of the relatedness of similar entities through abstraction and inheritance. OOD is far too broad a field to cover in any significant depth in a software engineering course, so we only have time to barely scratch the tip of the iceberg, so to speak. This is unfortunate, but if there is any specific topic you would like to see covered in more depth, let me know, and maybe I can post some additional material, or we can open up a discussion forum to talk about it.

Numerous object-oriented design techniques have been proposed over the years. Each technique had its pros and cons. Some could be used only with certain object-oriented languages. The list below will give you an idea of some of the techniques that have been developed and used throughout the last 20 years or so. A brief history of object-oriented design can be found at http://uml.tutorials.trireme.com/ [2].

1. Booch (by Grady Booch)
2. Object Modeling Technique (OMT)
3. Object-Oriented Software Engineering (OOSE)
4. CRC (Class-Responsibility-Collaborations)
5. Business Object Notation (BON)
6. Martin-Odell
7. Coad-Yourdon
8. MOSES
9. Object-oriented Systems Analysis (OSA)
10. Syntropy
11. Semantic Object Modeling Approach (SOMA)
12. Shlaer-Mellor

In 1994 there were reportedly at least 72 distinct object-oriented design techniques [2]. There were numerous overlaps between the various methods, and none of the approaches could really be used universally. Each technique had its own loyal faction of supporters. In an effort to standardize the field of object-oriented design, the creators of the Booch, OMT, and OOSE techniques got together and decided to build a single technique that combined the best of all the existing methods. What they came up with is the first version of the Unified Modeling Language, commonly known as UML. Those of you who are interested in finding out more should visit the official site of the Object Management Group (OMG), the official site of the creators of UML [3].

One of the primary goals of UML was to provide an object-oriented design technique that was independent of any programming language. A second primary goal was to make the technique extensible, meaning that users could change it by adding their own constructs as needed. UML has become very popular in certain sectors, although it has not yet received universal acceptance. This could be due largely to the fact that UML is quite large and complex, and there is a pretty steep learning curve involved in mastering it. You must thoroughly understand the UML specification in order to use it to its fullest potential. You also have to master at least one object-oriented language into which to translate your design. Finally, you need to have, and know how to use, one or more CASE tools that support UML, since it relies very heavily on diagrams.

UML uses a class diagram to represent the static architecture of the system. Sommerville

uses several examples of class diagrams in the text, and you've probably had prior experience with them in your job or one of your other classes. The static architecture can be supplemented by using interaction diagrams, which are based on use cases. A **use case** is a view of the system as seen from the perspective of a particular user, which can be a human or some other software application that interacts with the system. Different users will use the system in different ways. Together, all the use cases for a system should cover all the functionality of the system's requirements.

UML state diagrams can be used analogously to state-transition diagrams to show how the system goes from state to state. For a more detailed look, an interaction diagram can be used to show precisely how certain objects will interact to accomplish a particular function. In total, there are over a dozen or so diagrams UML supports. It's not always necessary to use every diagram for every project, although in my personal opinion a class diagram should be mandatory for the design of all object-oriented systems.

Unfortunately, we don't have time to cover UML extensively in this course, but I will mention a couple of things about CASE tools that support UML. Some, like Rational Architect, are exorbitantly expensive if you want to use them for commercial purposes, but others can be downloaded and used for free. For Java, two decent choices I have seen are Poseidon for UML [4] and ArgoUML [5]. Microsoft Visio supports the creation of class diagrams, but not surprisingly it is limited to Microsoft-centric languages for its default data types. Some tools will take a class diagram and generate skeleton code automatically. Skeleton code consists of the parts of a class' source code such as the class header, method headers, attribute declarations, and the getter and setter (often called accessor and mutator) methods. Basically, skeleton code refers to any code that can be generated automatically from a design document.

Abstraction

If you've ever taken an object-oriented programming class, you've undoubtedly heard about the concept of abstraction. (If this is old hat to you, feel free to skip this section). Abstraction simply means you take a design and extract those elements that are common to other related designs. As a simple contrived example, suppose you're writing an object-oriented toolkit for analyzing geometrical shapes. You come up with several classes for various two-dimensional shapes: circle, triangle, square, rectangle, ellipse, et al. One requirement states that you need to provide a way to compute the area of these shapes, so you add a method called area() to each class. You come up with the following UML class diagram:

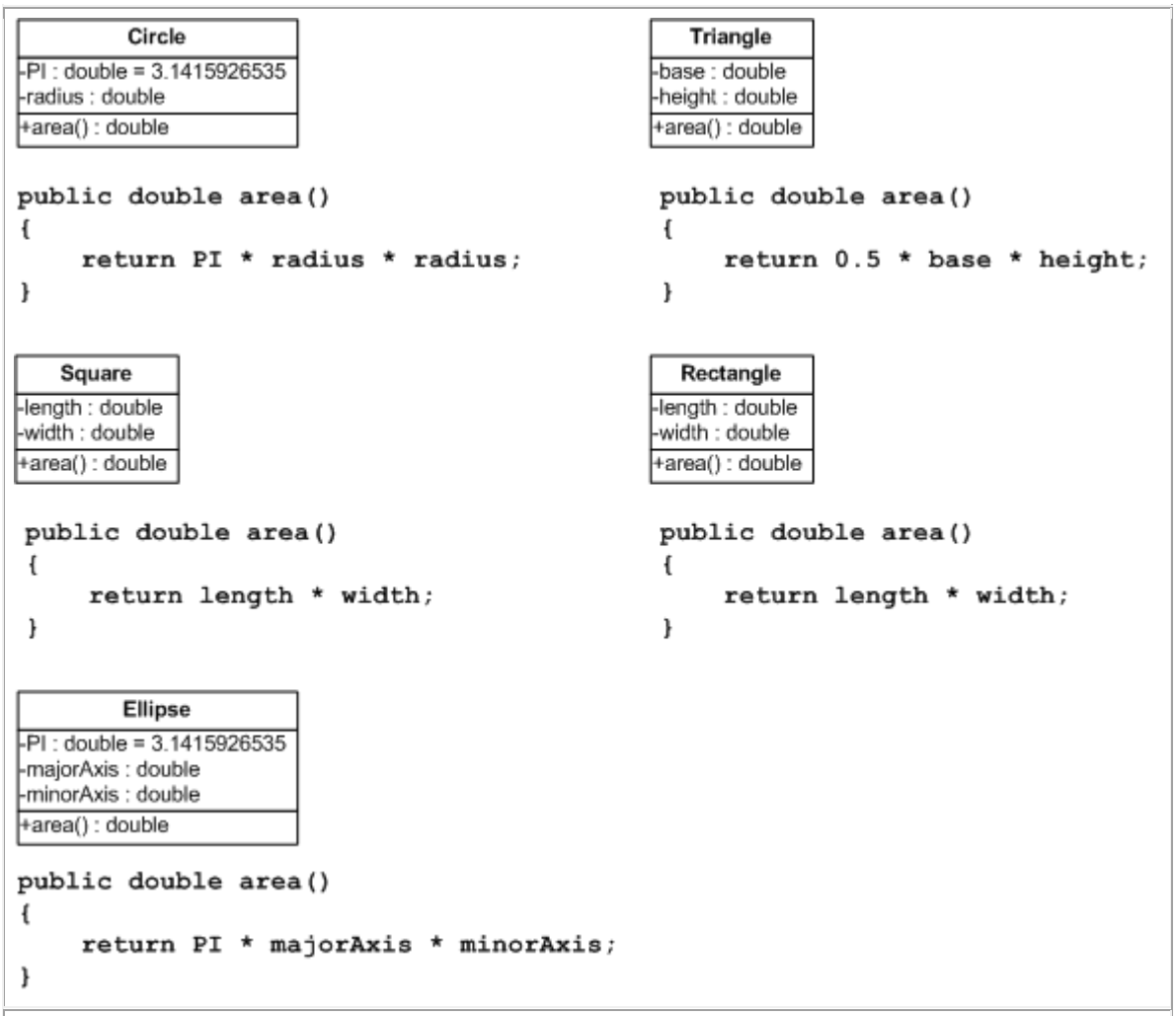

Figure 5.1. Initial class diagram for geometry toolkit.

Looking at this class diagram, it's easy to spot the commonalities. First of all, every shape has an `area()` method; only the implementation differs from shape to shape. Second, it can be seen that a square is simply a special case of a rectangle, and a circle is just a special case of an ellipse. If we extract the commonalities, we might end up with a class diagram like the one shown in Figure 5.2. Here, an abstract class, `Shape2D`, has been created and all other shapes are derived from this class. The `area()` method, common to all shapes, has been pulled out of the individual shape classes and placed into the `Shape2D` class. Using polymorphism, each subclass can inherit the method and override it appropriately. For the shapes that are special cases, `Circle` and `Ellipse`, I've shown them in the diagram, but there are arguments that can be made for and against including them. On one hand, they don't contain any additional functionality, so they are redundant in that respect. On the other hand, it may be desirable to allow users of the toolkit to be able to declare variables of type `Circle` or `Square`, rather than using `Ellipse` or `Rectangle`.

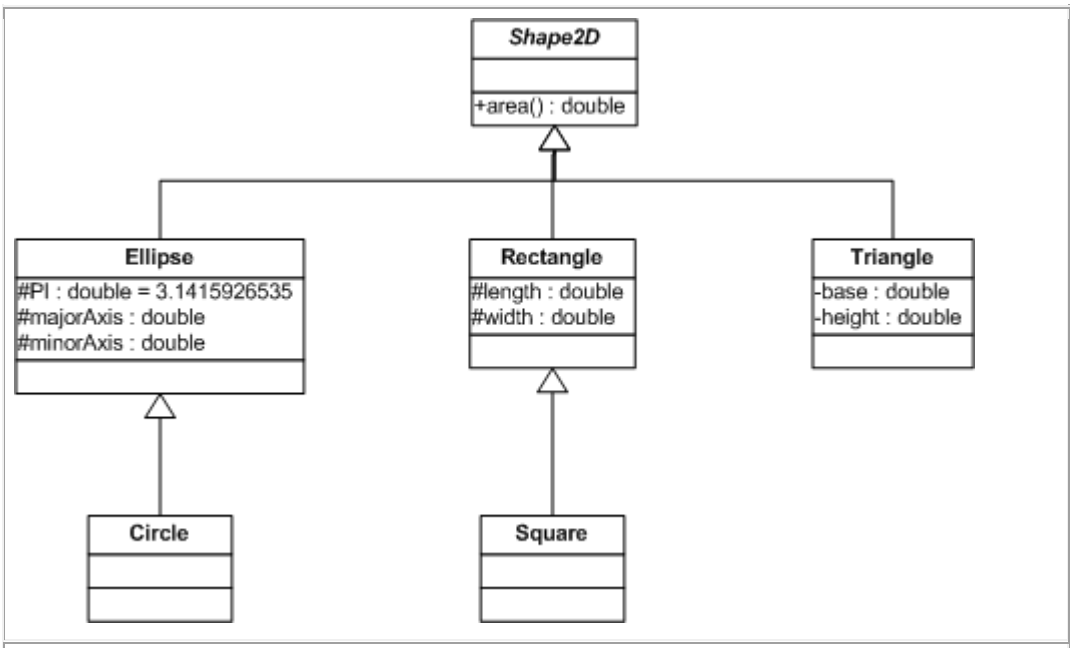

Figure 5.2. Class diagram for geometry toolkit, after commonalities have been extracted.

ï¿½

Pros and Cons of Object-Oriented Design

Object-oriented design has been touted by many in the field as the new paradigm for software development. Of course, not everyone agrees. Advantages of object-oriented design include:

1. It more closely approximates the way people think about real world problems.
2. It promotes good programming techniques such as encapsulation, modularization, and information hiding.
3. It can help to localize faults during debugging.
4. It promotes the development and use of repositories of reusable components.

Disadvantages include:

1. Designing a problem based on the way people tend to think about the problem may not be the best design strategy.
2. Developers accustomed to designing in a procedural manner sometimes have trouble switching to an object-oriented mindset.
3. Complex systems can be difficult to trace since execution jumps from one source file to another (i.e., consecutively executed statements will not always appear in consecutive order)

4. Significant overhead can be incurred during runtime, since more indirect addressing frequently occurs than in procedural systems.

Design Patterns

There are certain design problems that recur frequently in software development. One simple example of such a problem is traversing a collection of elements. The type of collection may vary, as well as the types of elements in the collection, but the solution is the same - you need to start at one end of the collection and traverse all the elements in the collection one at a time. While you can always create a solution to this problem anew every time the problem is encountered, it's tantamount to continually re-inventing the wheel. Why not find a way to create a standard solution that can be reused? In fact, this has been done for many recurring design problems, and the resulting solutions have been named, **design patterns**.

We don't have time to cover design patterns in significant depth, but there is time to introduce you to a few patterns. The iterator pattern I mentioned above is one example. Recall from your data structures course that each data structure stores its elements in different ways, and thus requires a different implementation for traversing those elements. Rather than try to keep track of all the different implementations, an iterator can be created that has a standard interface that allows you to use the same set of method calls regardless of which data structure you're traversing. The implementation details are encapsulated in the iterator, and are written only once.

There are times when a component does what you need, but doesn't provide an interface compatible with your components. You can use the **adapter pattern** to build an "adapter component" that provides the interface you need. For example, suppose you have a Java class that takes objects as input, and you need your class to accept output from a class that only provides primitive data types. Java provides a set of wrapper classes, one for each primitive data type, that allow you pass primitive data types as objects.

If you are designing an event-driven system, you will probably have a model and one or more components that need to be able to detect changes in the model's state, perhaps to update the user interface. In this case, you might consider using the **observer/observable** pattern. With this pattern, the model is designed as an observable component, and the components that need to detect changes in the model's state are designated as observers of the model. This pattern is often used as part of a larger pattern, the Model-View-Controller (MVC) pattern. The purpose of the MVC pattern is to decouple the model from the user interface, which allows you to do things like swap out certain user interface components without needing to modify the model in any way. For example, with a bank account system, you might use a GUI when your system is to be accessed by a personal computer, but you might use a text-based interface when your system is accessed by a remote terminal such as an ATM.

Certain design problems recur frequently when building software. For example, there are many situations where it's necessary to traverse a data structure, possibly to search for a particular item, or perhaps to insert or delete an item at a specific location. Recurrent problems that require similar solutions, but possibly in a different context, can be solved using a common solution. Such a solution is known as a **design pattern**. There are many design patterns in software engineering. I will only present two of them here.

The Iterator Pattern

The mechanism used for traversing a data structure depends on the underlying

implementation of the data structure. For example, an array-based list might be traversed using the following code:

```
for (int i = 0; i < n; i++)
{
    Object obj = theArray[i];
    doSomethingTo(obj);
}
```

By contrast, a linked list might be traversed using this code:

```
Node node = header.next; // assumes a header node is used to reference the
first node in the list
while (node != null)
{
    Object obj = node.getItem();
    doSomethingTo(obj);
}
```

Note that despite the differences in the two code samples above, both pieces of code do precisely the same thing: they both traverse a data structure, in this case a list. The problem is that in order to write correct code for traversing a data structure, you need to know how the data structure is implemented internally. Unfortunately, data structures can be implemented in a large variety of ways, so having to remember all this information can be problematic. Ideally it would be nice to be able to write the exact same code to traverse any data structure. Enter the iterator pattern. An iterator is simply a mechanism for traversing a collection. The goal is to create a data structure-specific iterator for each type of data structure, but "hide" the knowledge of the data structure's internal representation, and provide a common set of methods that can be used for all data structures. Using an iterator, you can traverse any data structure (array list, linked list, binary tree, hashtable, etc.) using the same set of methods, which might look something like this:

```
Iterator itr = dataStructure.iterator(); // create the iterator, specific for
the data structure
while (itr.hasNext()) // loop while there are more items to traverse
{
    Object obj = itr.getCurrent(); // get the current item
    doSomethingTo(obj);
    itr.next(); // advance to the next item
}
```

The advantage is that you only need to know this latest piece of code in order to traverse any data structure. The consistency provided by the pattern makes coding easier, and reduces errors.

The Adapter Pattern

Sometimes you will have a component that technically does what you need it to do, but because of the way it was written it is difficult or inconvenient for other components to use. One example of this is in the Java language. Java provides several primitive data types, such as `int`. Suppose you need to have an `ArrayList` that stores `int` values. You cannot directly store a primitive `int` value into an `ArrayList`, since an `ArrayList` only accepts references to an `Object`. You need some way to either adapt the `ArrayList` to store primitive `int` values, or you need to find some way to adapt primitive `int` values so they can be stored as `Objects`. As it turns out, Java provides this adapter for you. The `Integer` class stores a primitive `int` value, but by virtue of its being a class, it can be referenced as an `Object` in an `ArrayList`. To retrieve the primitive `int` value, you use the method, `getInt()`. Incidentally,

the `Integer` class is also called a "wrapper" class because it "wraps" another component, namely the primitive `int` value. Java provides similar wrappers for the other primitive data types.

If you are interested in learning about other design patterns, the book by Gamma [7] is a good reference.

User Interface Design

Software that will be used directly by humans must provide some way for humans to interact with, and use it. The user interface is the user's gateway to the functionality the software provides. Considerable effort must go into the design of the user interface, because if the user interface is no good, people are less likely to use the software, regardless of its benefits. Sommerville notes six design principles that should be taken into account when designing a user interface. I repeat them here, and give an example for each principle.

1. User Familiarity

   The language and features of the interface should be recognizable to the people who will use the software. For example, if you are building an interface for software used to create music CD's, it is probably better for users to have a menu command called, "Burn CD", than to have one called, "Write .WAV bytes to optical disk". Even the term "burn" is meaningless unless you understand how data are written to a CD. Nevertheless, the term has successfully made itself a part of our culture.

2. Consistency

   Commands that can be performed in multiple ways should be given the same label. For example, "Burn CD" should be labeled as such whether it appears in the main menu, a popup menu, or on a button. When designing software product families, commonly used commands such as opening and closing files, copying and pasting, etc., should all be labeled consistently throughout each product in the family, and should use the same set of shortcut keys.

3. Minimal Surprise

   The user interface should never produce any message or result that the user does not expect. Unfortunately, these are still all too common in much of today's software. Whenever the user executes a command, there should always be some clear notification that the command was executed successfully, or that the attempt failed. Users should never have to ask the question, "Did it do what I told it to?"

4. Recoverability

   The interface should allow users to recover from errors. This refers to user errors, not software failures. For example, providing an "Undo" feature allows the user to erase the most previous action in case the user accidentally does something wrong. All destructive actions (delete, cut, overwrite, etc.) should prompt the user if these actions cannot be undone.

5. User Guidance

   If the user needs assistance, there should be help available, preferably in a context-sensitive format. If something goes wrong, meaningful error messages should be

displayed. Error messages like, "Illegal stack operation at address 0x038A" are of little use to most users.

6. User Diversity

   Creators of a user interface should realize that the software may be used by people with varying degrees of experience, people from different cultures, and people with different physical capacities. Even though it's seldom possible to build an interface that will satisfy everyone, care should be taken not to alienate or offend any potential users.

Together, all the aforementioned principles comprise what is commonly referred to as "user-friendliness". They are also a measure of software's *usability*, one of the key characteristics used to determine software quality.

Prototyping

User interface construction lends itself very well to the prototyping process model, since it's usually difficult to determine precisely how the customer wants the interface to be. Mock-ups can be created very quickly and shown to the customer. These mock-ups can be static images created with an application such as Photoshop, or even drawn on paper. Alternatively, applications such as Visual Basic can be used to quickly build a dynamic, pseudo-functional interface that will allow users to play around with it and get a feel for what it will actually be like to use the software. Dummy data can be used, if necessary, to enhance the realism of the prototype. One word of caution, however, is in order. If the prototype appears to be too "real" it may lead the customer to believe that the back-end is also finished, and they may try to move up the release date or demand additional features. This generally stems from ignorance on the part of the customer as to what is involved in building software. Customers and end users typically only see the front end, and fail to realize that the front end is nothing more than the control panel.

At what point in development can (should) the user interface be developed?

In a well-designed system, once the requirements have been finalized, the user interface should be able to be developed in parallel with the rest of the system. Back end development should not depend on the user interface, and user interface development should not depend on completion of the back end. In general, the user interface should be very loosely coupled to the back end. It should be possible to swap out one version of the user interface with another version without any loss of functionality. The Model-View-Controller design pattern illustrates this point. The view represents the user interface, the model represents the back end, and the controller provides a link between the two. If designed well, a graphical view could be swapped with a text-based view, for example. The functionality of the software is still all there, it's just accessed differently.

From a testing standpoint, it can be helpful to have portions of the user interface completed prior to testing, although this isn't always possible. The user interface can serve as the test harness, and reduce the amount of scaffolding code that must be written.

GUIs

Most software built for use in personal computers uses a graphical user interface, or GUI. GUIs are popular because they provide the very valuable "WYSIWYG" (What You See Is What You Get) capability. Anyone who has used older word processors, such as WordPerfect prior to version 6.0, probably remembers seeing text of various colors, parts of which may be highlighted in other colors, on a blue background. As a user, you have to remember that white text is normal, yellow text is bold, a certain color of highlight means the text is underlined, etc. GUIs show the text exactly as it is supposed to appear in print form. GUIs also provide extremely valuable capabilities such as drag-and-drop, multiple windows, clickable icons, etc.

However, there is a tradeoff for using a GUI. That tradeoff is slower running time. Graphical artifacts take longer to draw on the screen, and each open window consumes memory. If enough windows are opened it will eventually force the operating system to begin using virtual memory, which itself causes a performance hit. As computers have gotten faster, this decrease in performance has become less noticeable, but it is there.

Customizable User Interfaces

In an effort to appease all users, some developers provide user interfaces that are customizable or extensible. Most software already provides multiple ways to accomplish the same task (main menu, context menu, shortcut key). If you want to use a different shortcut key for a particular function, you can change it. If a custom toolbar is desired, it can be created by removing icons for unwanted or unused tools, and adding icons for tools that are not part of the default toolbar. The products of the Microsoft Office suite all allow this capability. Some software, such as Dreamweaver, even allows you to add your own commands to the main menu. (For those of you unfamiliar with Dreamweaver, it is a program used for creating web-based content. Its main menu can be extended by writing a custom function in Javascript and adding it to the Command menu).

Some degree of GUI customization is expected. There are people who, due to some physical ailment, are unable to use a mouse. Such individuals must therefore be able to do everything they need to do using the keyboard or some other input device. People with vision problems often need the ability to enlarge the text on the screen in order to read it. These types of customization are more appropriately categorized as accessibility features.

Extensive customization can present development problems. Every control of the user interface, whether it is a menu command, push button, scrollbar, etc., must be tested to ensure that it works properly. This means that each user interface artifact must be capable of being manipulated with reasonable ease. The target area must be sufficiently large. Expecting users to click on controls that are only a single pixel wide is probably not a good idea. Proper functioning also means that when a particular artifact is chosen, the correct function is performed. If there are three different ways to perform a function, all three ways must be tested, and they must all produce the same result.

Coding

Sommerville doesn't really have a chapter in his text dedicated to the coding phase of software development, so I'll devote part of the lecture to this topic. Part of the purpose of this course is to dispel any ideas you may have that software engineering is simply a glorified title for code hacker. Many people still have an unfortunate tendency to equate coding with software engineering to the point they consider the two to be synonymous. There exist notions of software engineers as introverted geeks who sit in tiny cubicles for 15-20 hours a

day doing nothing but writing code. Sadly, in many cases this is closer to truth than fiction, which makes it all the more difficult to relay a distinction between software engineering and coding. To give credit where it's due, there are some code hackers out there who are good enough to design some of their programs on the fly, and still end up with reasonably good products. I don't think most of us can do that, except for relatively simple problems. Writing software is easy; writing high quality software can be quite hard.

Probably the reason why the code receives so much focus is because it is the code that actually gets compiled and translated into the working software. If you're following the Extreme Programming process model, you are taught that the code should be self-documenting for the most part, and that supplemental documentation is largely a waste of time because no one will ever look at it. Movies and television also tend to glorify programmers as "geeks" who do nothing but write code all day, and eventually become so good at it they don't need the shackle of documentation or design—they're just good enough to whip something out in a matter of hours (or maybe a few days if the problem is really complex). In the early days of software engineering, where all the computing was done on time-shared mainframes, and the software engineers of the day were only allotted a precious, limited amount of time to actually compile and run their code, those engineers spent a lot more time on design than may have been apparent. They had to, because there was no telling how long they might have to wait before they were able to try again. So they would review their code for faults while they waited for their next turn on the mainframe.

As personal computers became more widely available, and as they became faster, it became possible to take shortcuts. Since people tend to be short on patience, rather than spend a lot of time trying to figure out on paper whether or not something would work, programmers could just run the code and let the errors (or lack thereof) tell them how they did. Unfortunately, it's easy to end up with code that appears to work, at least for all the test cases that have been run, but then crash later when a case is encountered that exposes a design flaw. The market's demand for rapid software availability, combined with the insatiable corporate thirst for profit, only exacerbates the situation by promoting more time spent on writing code. This could partially explain why a lot of software written in the 1960's and 1970's seemed to be more stable than software written later. Of course, another driving factor is the fact that some software is becoming ever more complex as it evolves.

An interesting philosophy regarding coding relates to the notion of an executable specification, which I mentioned briefly in a previous module. With an executable specification, functional software is generated automatically from the requirements specification. There essentially is no separate design phase, except possibly to deal with a few non-functional requirements, because all the design details follow naturally from the requirements. Source code can be generated, compiled, and tested automatically. Alternatively, machine code can be generated directly from the specification, eliminating the need for source code. In order for this to work, the requirements must be expressed in such a way that they are readable by a software application. Typical human languages contain too many ambiguities to work for this. What we would need is a language that is capable of expressing the requirements unambiguously. Such languages exist, and we will talk about them later when we cover formal methods in more detail. For now, suffice it to say that this is not a viable approach for most software projects because for formal methods to work requires the customer to know, completely and unambiguously, what they want the software to do, and this is never the case. Also, it is exceedingly difficult and costly to correctly define requirements using formal methods, so the motivation to use them is low.

Coding Styles and Readability

I only want to spend a brief amount of time on this topic. I'm sure all of you have been exposed to at least one style of coding style and documentation. Many styles exist, and each

has its own cadre of loyal supporters. Some styles have advantages over others, but I doubt anyone could prove that one style is "best". The goal of any coding style is to make the code readable and understandable to *anyone* (i.e., other software engineers) who may have a need to look at it. It's always a good idea to assume that at some point in time someone may have to make sense of another person's code, even though they may not have been a member of the team that originally developed the project. Even if no one but the author will ever need to look at the code, it's never a good idea to assume memory is infallible. An author may look at code they wrote sometime in the past and wonder what the code does, or why it does what it does.

Code Obfuscation

Obfuscated code is code that has been written in such a way that it is not easily intelligible. This contradicts the notion that code should be readable and understandable by others. There are two primary reasons why someone might want to obfuscate (or "shroud") their code:

1. *Security*. Obfuscation deters reverse engineering by preventing others from easily understanding how a particular problem was solved by simply looking at the source code. This is good if you want to have an edge over competitors, although the edge is likely to be only temporary. It can also deter hacker attacks by making it difficult for a hacker to pinpoint vulnerabilities in the code. Of course, the assumption here is that a human will be looking at the code. Obfuscated code will not hinder a computer program designed to analyze code, because even though obfuscated code may look strange to a person, it must still follow the syntactical grammar of the language in which it is written.

2. *Recreation*. Those who are interested in a challenge may try to write cleverly obfuscated code simply for enjoyment, or to enter a contest. One of my personal favorites is the following obfuscated C code which, when executed, displays the lyrics to the "12 Days of Christmas":

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+
,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')()#}w'){){nl]'/+#n';d}rw'
i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n'
')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m
.vpbks,fxntdCeghiry"),a+1);}
```

References

1. Parnas, D. L., On the criteria to be used in decomposing systems into modules, *Communications of the ACM 15*:12 (Dec. 1972), 1053-8.

2. Trireme International, Ltd. UML Tutorial. http://uml-tutorials.trireme.com/ (date unknown).
3. Object Management Group website. http://www.omg.org.
4. Website for Gentleware, creators of Poseidon for UML. http://www.gentleware.com
5. Website for Tigris.org, creators of ArgoUML. http://argouml.tigris.org/
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

ï¿½