Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

It is recommended that you prioritize the readings: studying the “primary” ones first and then looking at as many of the “secondary” ones as you can. Unless otherwise noted, readings are from Systems Analysis and Design: An Object-Oriented Approach with UML, by Dennis, Wixom, & Tegarden (6th Edition, 2021). In the readings listed below, you are given a page range for the reading, but you are only required to read the subsections that are itemized. If no subsections are mentioned, you are required to read the entire page range.

Module 6 Study Guide and Deliverables

Readings:

Primary Reading for Module 6:

The following readings should be completed after reading the module parts that they pertain to.

- Pages 310-314: Method Specification
- Pages 60-65: Project Effort Estimation

Secondary Reading for Module 6:

The following readings are not required, however they provide additional depth and examples for concepts in this module.

- Pages 138-148: Business Process Modeling with Activity Diagrams
- Techniques for Estimating – Planning Poker (opens PDF), Mountaingoatsoftware (12 pages)

The *Encounter* video game case study will be referenced in this module.

- Requirements for the *Encounter* Video Game (opens PDF)
- Design of the *Encounter* Video Game (opens PDF)

Assignments:

- There is no Draft Assignment for this module.
- Assignment 6: Part 4 of Term Project due Thursday, February 29 at 6:00 am ET

Course Evaluation:	Course Evaluation opens on Monday, February 26, at 10:00 AM ET and closes on Sunday, March 3, at 11:59 PM ET. Please complete the course evaluation. Your feedback is important to MET, as it helps us make improvements to the program and the course for future students.
Live Classroom:	• Tuesday, February 20 from 8:00-10:00 pm ET - Class Lecture • Wednesday, February 21 from 8:00-9:00 pm ET - Assignment Preview • Live Office: Saturday, February 24 from 1:00-2:00 pm ET

Module 6 Objectives

In Module 4 you learned UML notation; in Module 5 you put UML to practice and used it to describe architecture—the high-level software design. This week you will complete the process by learning techniques for specifying the full design details. This week's lectures complete the description of specification methods for communicating detailed system designs.

By reading the lectures and completing the assignments, you will be able to:

- Create detailed designs sufficient for implementation
- Assess and explain reuse
- Formulate detailed sequence and data flow diagrams
- Specify algorithms in detail
- Judge how and where to apply IEEE standards for expressing detailed designs
- Know and explain the fundamentals of estimation

Part I: Detailed Design Principles

Introduction

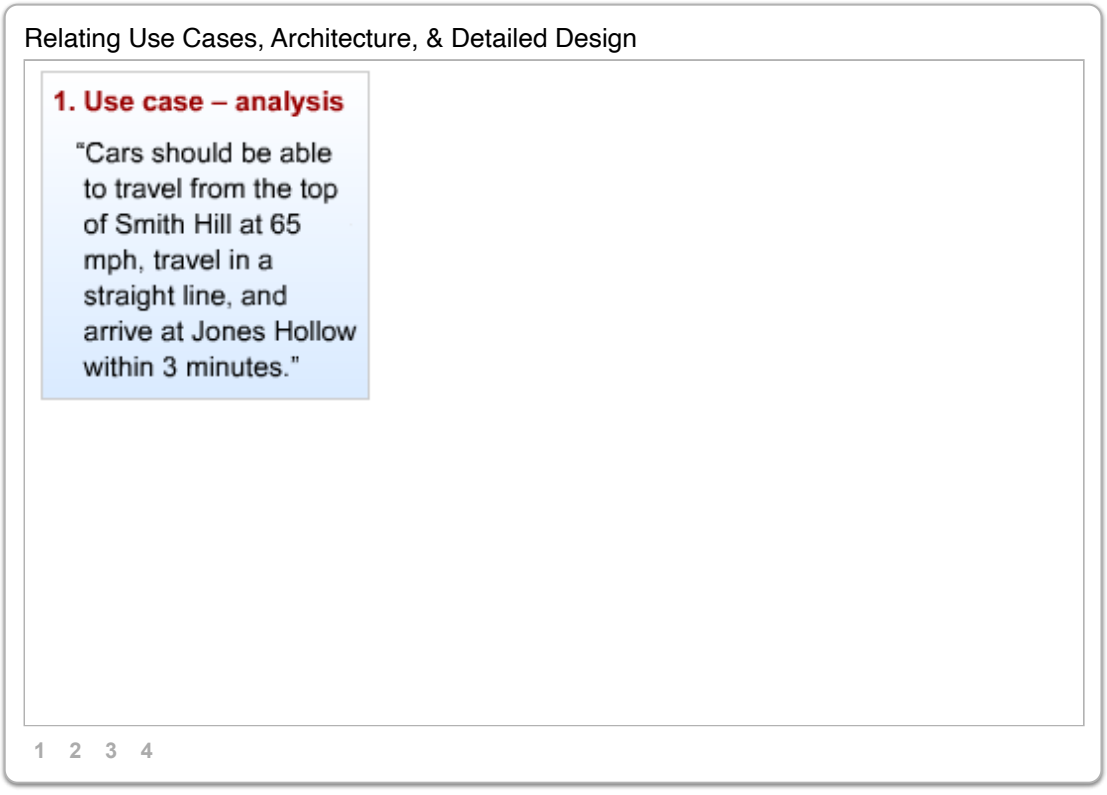
We will start by relating the specification of the detail design process to the document artifacts that have already been developed. Previous document artifacts described business objects (mostly domain classes identified bottom-up in conjunction with requirements analysis) and the architecture (mostly abstract classes identified by looking at the system top-down).

The goal of this lecture is to cover the following topics:

- Detailed design based on use cases
- Detailed design against known interfaces
- Detailed design through reuse of off-the-shelf components
- Detailed design with sequence diagrams and data flow diagrams

Relating Use Cases, Architecture, and Detailed Design

The relationship between the use cases, architecture, and detailed design can be understood by analogy with the process of designing a bridge. Use cases would be part of the requirements for the bridge. Based on the requirements, engineers would then select an architecture (e.g., a suspension bridge) that best suits the application. After the architecture selection, designers would develop the detailed design to implement the required use cases with the desired architecture. This process, together with an architecture choice, is illustrated below.



Many factors in addition to actual business requirements go into making a detailed design. In the bridge analogy, the requirements may not have specified a guard rail or the support needed beneath the roadway, for example. Construction codes and stress analysis typically dictate these. The required addition of details is no less prevalent in completing an IT system.

In the IT process, each corresponding stage of system development accumulates information about the final class model. In step one, use cases are specified as part of the requirements. In step two, these, together with other sources, are used to identify the business objects (domain classes or entity classes). In step three, we develop the system architecture, as described in Module 5. The last step is to verify that the architecture and detailed design support the required use cases. For the bridge analogy, we verify that cars can indeed use the bridge design to travel from Smith Hill to Jones Hollow as specified. For IT design, we verify that the classes and their methods specified by the detailed design include all necessary algorithms and supporting data.

A criticism of this design is its excessive and arguably incorrect use of inheritance. For example, the Address hierarchy implies that Business Address and Shipping Address both have exactly the same attributes as their superclass Address. The diagram also implies that Email Address has the same attributes as its superclass Address. This is simply incorrect.

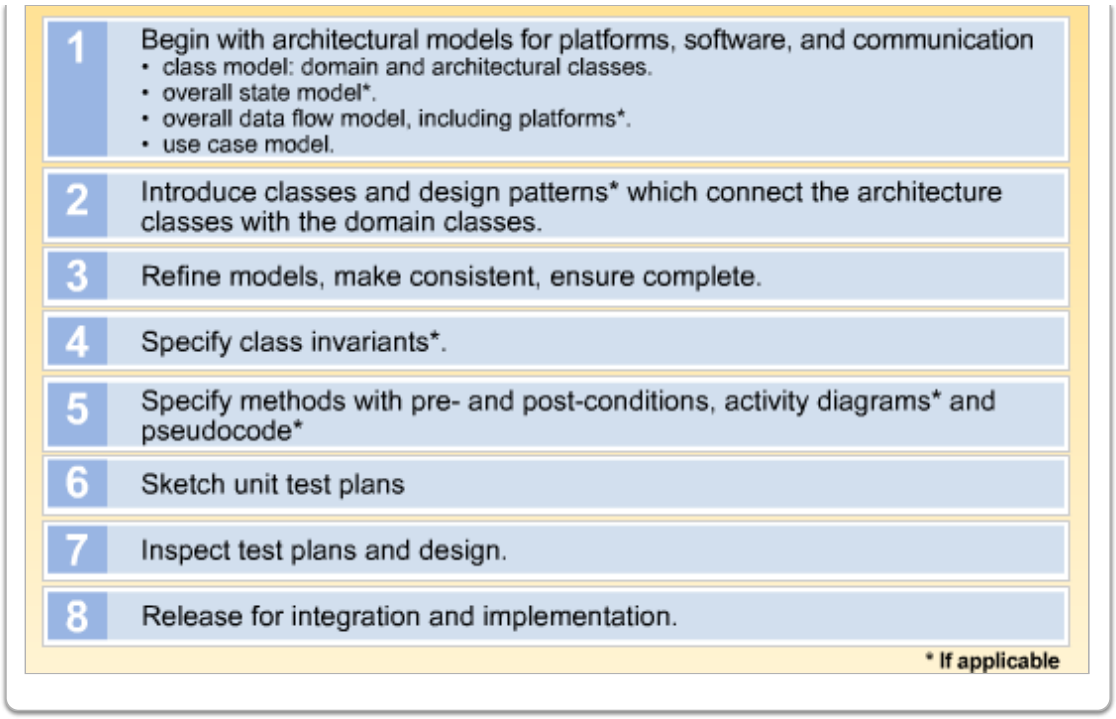
A Typical Road Map for the "Detailed Design" Process

Detailed design starts with the results of the architecture phase and ends with a complete blueprint for the programming phase. The blueprint would include details about the classes, their relationships, data fields, and methods. The roadmap below shows a typical sequence of steps taken to perform detailed design. In particular, step two of the roadmap creates the classes that associate the architecture on one hand with the business objects on the other hand, as illustrated in the previous section. Design patterns (like the Façade and State design patterns mentioned in Module 5) may help in doing this.

It is advisable to start the detailed design process with those aspects of the design that present the most risk. For example, in designing our *Encounter* video game, we might consider it risky to modularize in the manner described earlier in this course (with all game characters in one package, etc.). This should be settled as soon as possible by specifying the details of the interface methods so that we can get an intimate idea of how this modularization works out and whether it is characterized by low cohesion and high coupling. If the use of the State design pattern were perceived as a greater risk, we would specify its details first. In many cases, however, the order of specifying detailed design does not really matter.

Step three of the roadmap includes checking that we have a complete design. It also includes ensuring that the object model supports the use cases. Step six continues the practice of specifying a test as soon as each element is specified.

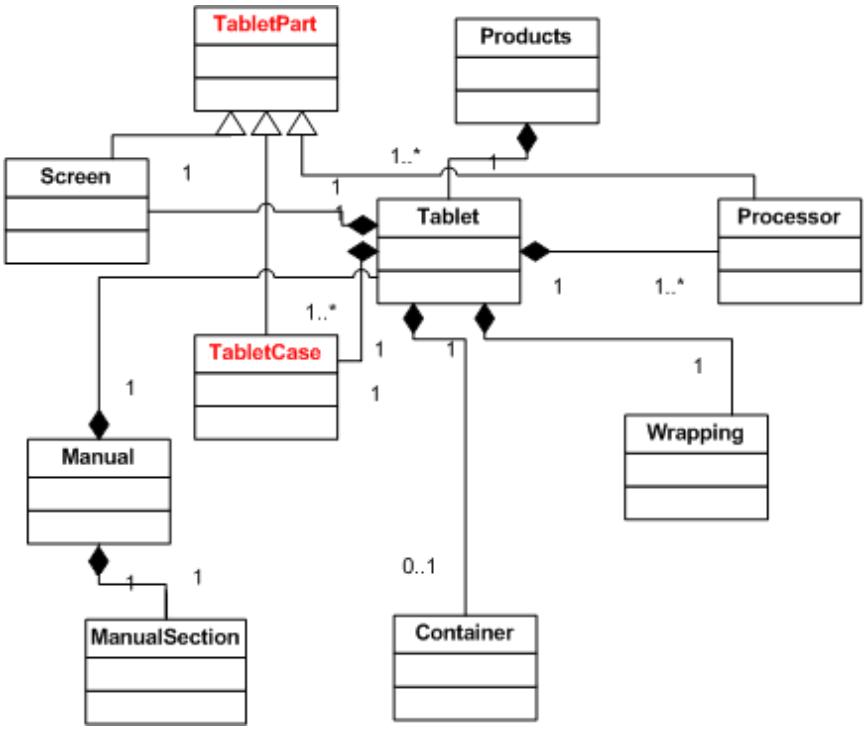
Roadmap for Project Management



Introducing Design Classes for Design Completeness

As explained in the previous module, architectures provide class (and other) models at a high level. It is then necessary to specify the classes that complete the design. To the extent possible, we introduce these prior to coding. However, developers generally introduce classes in addition to these as they uncover the need for additional classes. Common sources of this class completion process are (1) useful generalizations and (2) useful specializations of the existing classes.

Consider, for example, the model shown in Module 5 for a system that controls inventory for a factory that makes tablet computers. The classes shown in red are examples of additional ones that contribute to completion of the class model. *TabletPart* is a general class introduced by the analyst's realization that several of the classes possess commonality. *TabletCase* (in which to carry around the tablet) is a part that the analyst realizes makes up a tablet product.



Designing Against Interfaces

The idea of designing against interfaces is like employing a contract. In fact, one rigorous form of designing to interfaces is called *Design by Contract*. The methodology is developed by Bertrand Meyer, the designer of an excellent (but commercially unsuccessful) object-oriented programming language, Eiffel. Here, the term "interface" means a set of methods that a class provides to other classes in an application. For example, class Account might provide a method *withdraw()*, which will be called by another class, e.g., SessionManager. Each method has its own signature. The method signature includes the data types of the method parameters, preconditions (which the caller must ensure before the call to the method), and postconditions (what work the method would do for the caller if the precondition is satisfied). For example, method *withdraw()* might have the parameter withdrawalAmount of type double, the precondition that the value of withdrawalAmount does not exceed the value of the "balance" data field of class Account, and the postcondition that the balance is decreased by the value of withdrawalAmount.

As mentioned earlier, the term "interface" is used to denote several different things in programming. The most important meaning is described above. It is a set of methods that other classes can call. Another meaning is a set of method parameters with preconditions and postconditions. This is what we called a method signature above. Another meaning denotes a set of methods that is available for a programmer who uses a package, a class library, an application layer, or an application (as in API) in the program. Yet another meaning denotes an abstract class-like construct in Java that other classes would "implement," but this topic is too programming-oriented to be discussed here.

At the architectural scale, we have seen how the components provide interfaces that other components use. In web services using SOAP, a WSDL is used to specify an interface. The WSDL provides schema typed constructs to specify the required format of inputs and outputs. (Web services follows a standard protocol referred to as SOAP. SOAP supports XML-based messages over HTTP.)

At the software level, the program element (e.g., the *Customer*class) supplying the functionality guarantees to provide functions with specified names, parameter types, and return types (e.g., *void bill (void)* and *boolean printAccounts(String accountType))*. The programmer using these program elements can design his application without having to know how the functionality is implemented. We have discussed this concept in the context of design patterns where patterns have clients. Also, the *Façade* design pattern is a way of providing a clean interface to a package of classes.

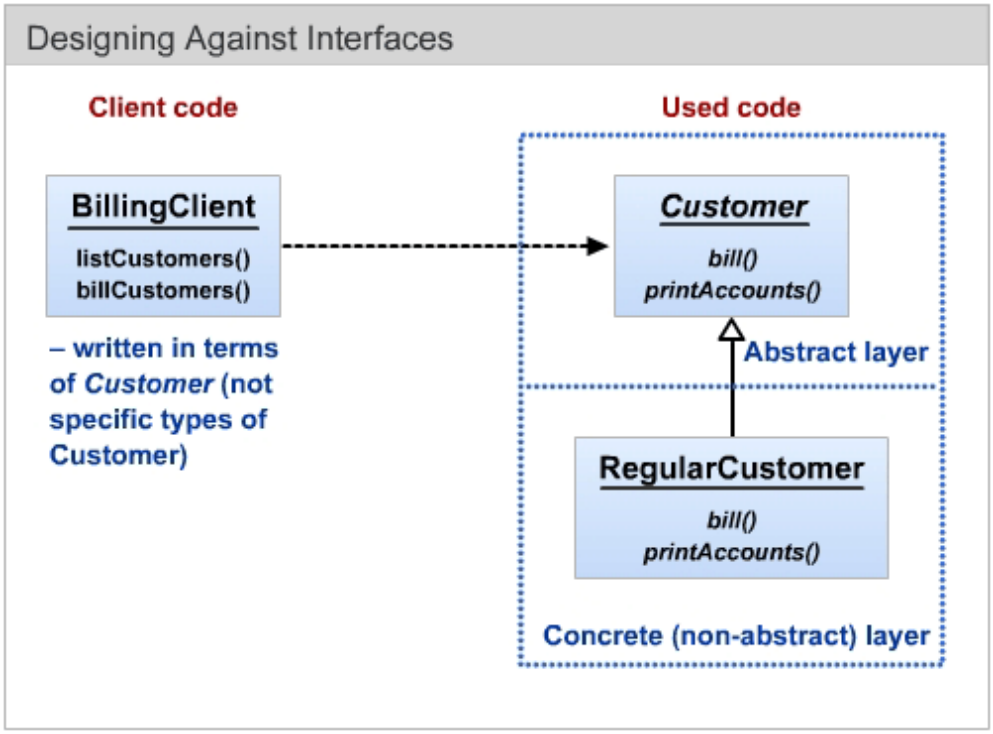
Designing against interfaces is an important design principle. It says, for example, that the programmer who calls methods *bill()* and *printAccount()* need not learn the details of billing or printing as long as he knows that these operations would be performed correctly if the preconditions specified for these methods are satisfied. Learning internal details of these methods costs money, distracts attention, crowds out other knowledge, and slows down the development process. In addition, using details in the calling methods increases coupling and makes maintenance more difficult.

Designing against interfaces takes many forms. One is the use of abstraction. For example, if code is to be written about the way *Mammal, Fish, and Bird* objects move, then we try to write it to mention only *Animal*objects where the Animal class is a superclass of classes Mammal, Fish, and Bird. In other words, we try to use the *Animal*interface (set of methods) even though the program will in fact use the objects of implemented subclasses of class Animal (Mammal, Fish, and Bird). When the program calls the method *move()* for class Animal, it in fact calls a method *move()* from one of concrete classes (a polymorphic call). This allows greater applicability and greater flexibility for our design.

As a further example, suppose that we are writing code about customers. This can be understood as writing against the Customer interface. We can consider using an abstract class Customer, with a non-abstract (concrete) subclass such as RegularCustomer, as shown in the following example.

This design is more flexible than writing against a concrete (non-abstract) class Customer, because to the abstract superclass Customer we can easily add other types of customers, such as SavingsCustomer. These new classes would have specialized versions of method *bill()*, each designed for its own type of customers. Then there would be no need to change the code that uses Customer objects. If the initial design had the concrete class Customer instead, adding new types of customers would require adding to class Customer a new data field that would specify what type of customer this specific object is. Then we would change the *bill()* method, implementing different algorithms depending on the type of the customer. That is, to ADD new capabilities, we would have to MODIFY existing code. This is always difficult and error-prone. The design with an abstract class depicted below allows us to ADD new types of customers by ADDING new code. This is much better.

The division into an abstract and a concrete layer is characteristic of many design patterns.



Reusing Components

Most engineering disciplines (electrical, mechanical, etc.) rely on the use of components that can be procured separately. Bridge designers, for example, try to use standard I-beams. Until recently, software developers had no components to reuse. Applications were crafted totally by hand. Of course, experienced designers relied on their previous experience and reused designs, at least partially, but implementation had to be repeated from scratch because functions written in older (“procedure”) languages cannot be used outside of the initial context.

The widespread adoption of object-oriented and other component paradigms has helped to promote software reuse. Because of the large number of methods packaged with each class, the functionality that we need is often included and is relatively convenient to locate. The use of Microsoft libraries, Visual Basic controls, Microsoft Assemblies, JavaBeans, and other Java Application Programming Interface (API) classes are examples of code reuse. The Object Management Group's Common Object Request Broker Architecture (CORBA) is a standard for distributed reuse.

Frameworks, discussed in Module 5, are packages of components designed for reuse. We developed frameworks to support application architectures, and so they are effectively reusable. JavaBeans provide reusable components for Java applications. They include graphics beans and “enterprise” beans, which encapsulate corporate tasks such as database access. In addition to the advantages afforded by being classes, beans obey standards that make them capable of manipulation within development environments. Web-based programs (i.e., not components), such as JavaScript and CGI scripts, are often reused.

At a different level, the C++ Standard Template Library (STL) provides mix-and-match capability of standard algorithms, such as sorting and searching. STL is applicable to a variety of data structures and to objects of virtually any class. Its components are thoroughly tested and their performance is optimized.

In summary, a component marketplace has emerged and is growing continually. Actually, there are so many reusable components in different libraries today, that it becomes a serious problem for a developer to (a) review existing components, (b) find those that might fit into a new application, (c) check to what extent they fit, and (d) determine how they should be modified. Sometimes, it is easier to write a component anew than to spend time reviewing the properties of huge libraries of available components.

Having found a component that could possibly be used in an application, should it be used? The following factors are typical in making this decision.

Considering a Component for Reuse

- Is the component documented thoroughly?
    - If not, can it be?
- How much customization of the application is required?
- Has the component been tested thoroughly?
    - If not, can it be?

Sequence and Data Flow Diagrams for Detailed Design

Some detailed designs are best communicated via detailed sequence diagrams or detailed data flow diagrams. We used sequence diagrams and data flow diagrams in specifying requirements - especially as an aid to selecting business objects - but it was not necessary to select actual function names at that stage. When we come to detailed design, however, function (method) names must be identified.

The two outlines below provide guidance on what needs to be done with sequence diagrams and data flow diagrams to carry out detailed design. The sections following this one provide details and examples.

Refining Models for Detailed Design: *Sequence Diagrams*

1. Begin with the sequence diagrams (system sequence diagrams) constructed for detailed requirements and/or architecture (if any) corresponding to the use cases.
2. Introduce additional use cases, if necessary, to describe how parts of the design typically interact with the rest of the application.
3. Provide sequence diagrams (detailed sequence diagrams) with complete details.
    1. Be sure that the exact objects and their classes are specified.

2. Select specific function names in place of natural language
   (calls of one object to another to perform an operation which the first object needs).

Refining Models for Detailed Design: *Data Flow Diagrams*

1. Gather data flow diagrams (DFDs) constructed for detailed requirements and/or architecture (if any).
2. Introduce additional DFDs, if necessary, to explain data and processing flows.
3. Indicate what part(s) of the other models the DFDs correspond to.
    1. E.g., "The following DFD is for each *Account* object."
4. Provide all details on the DFDs.
    1. Indicate clearly the nature of the processing at each node.
    2. Indicate clearly the kind of data transmitted.
    3. Expand processing nodes into DFDs if the processing description requires more detail.

Detailed Sequence Diagrams

Recall that use cases can be utilized to express requirements, and that we also use them to determine the key business objects for the application. For the detailed design phase, we provide classes with the methods referenced in the sequence diagrams.

Let's take as an example the *Encounter* video game case study discussed throughout this course. The sequence diagram for the "Encounter Foreign Character" use case in the case study is shown below, except that instead of the verbal descriptions of functionality called, which we used for requirements, we designate specific functions. A refinement is required because we introduced the *Façade* class *EncounterCast,* through which all external references to the game characters must pass. The reasoning behind the functions chosen is as follows:

1. *ForeignCharacter* is to have a display function. We will implement this with a method *display()*. Since all characters will need to be displayed we can actually ensure this requirement by giving the base (abstract) class *GameCharacter* a *display()* method. Other classes for game characters inherit from this class. The sequence diagram shows *EncounterGame* creating the foreign character (step 1.2) and also an Engagement object, and then calling *display()*.

2. This step in the use case indicates that we need an *execute()* method in *Engagement*.

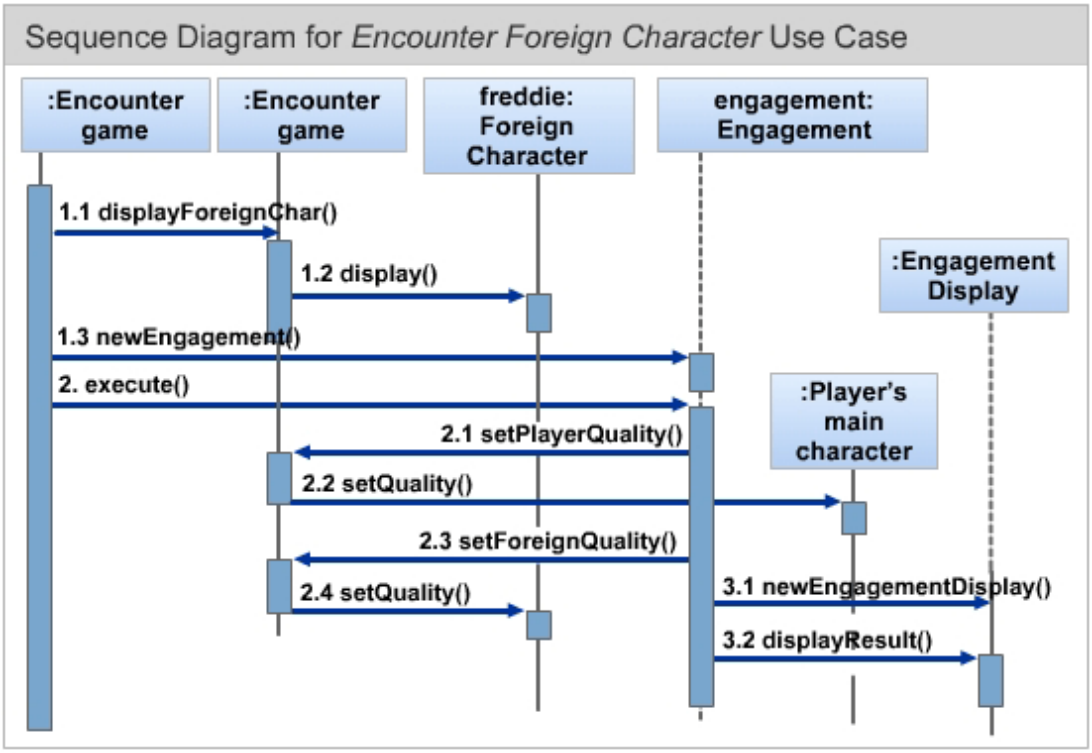
   2.1. This step requires that Freddie and the main player character be able to change their quality values. Since this capability is common to all *Encounter* characters we provide the base *EncounterCharacter* class

with a *setQuality()* method.

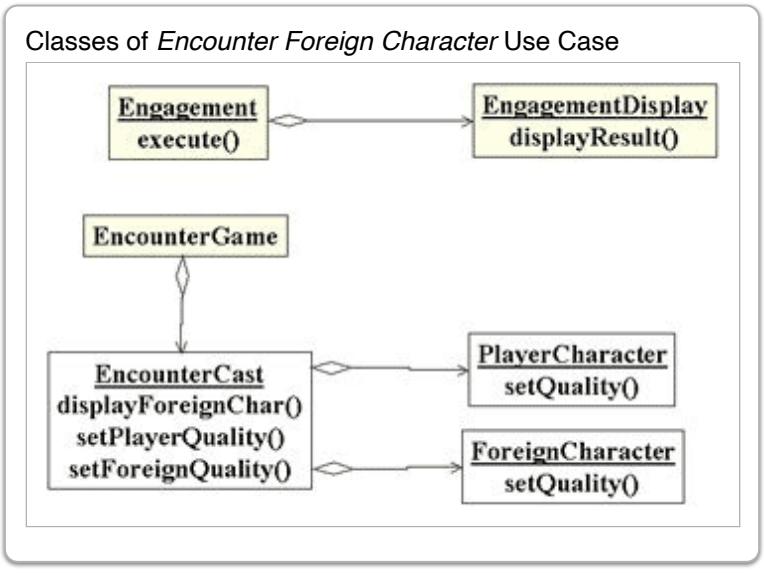
3. This step requires that the result of the Engagement be shown. The following two sub-steps constitute one way to do this.

   3.1. First Engagement creates an *EngagementDisplay* object.

   3.2. Now we show the Engagement display by calling its *displayResult()* method.



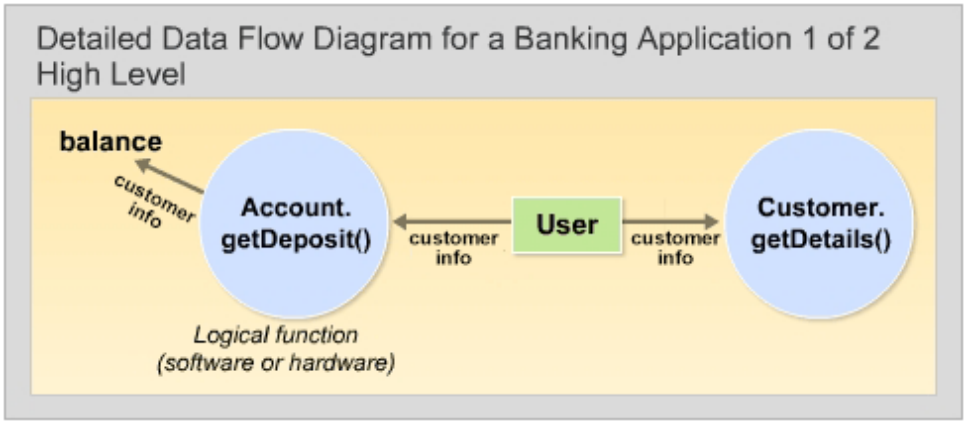
Since the names of the methods required to execute this use case are now identified, we can indicate them on the object model, as shown below. Continuing this process, the class model and the use case model (in the form of sequence diagrams) are completed in detail, as shown in the case study. The state model (if applicable) must also be completed in detail. A data flow diagram is yet another possibly useful model, and is discussed next.



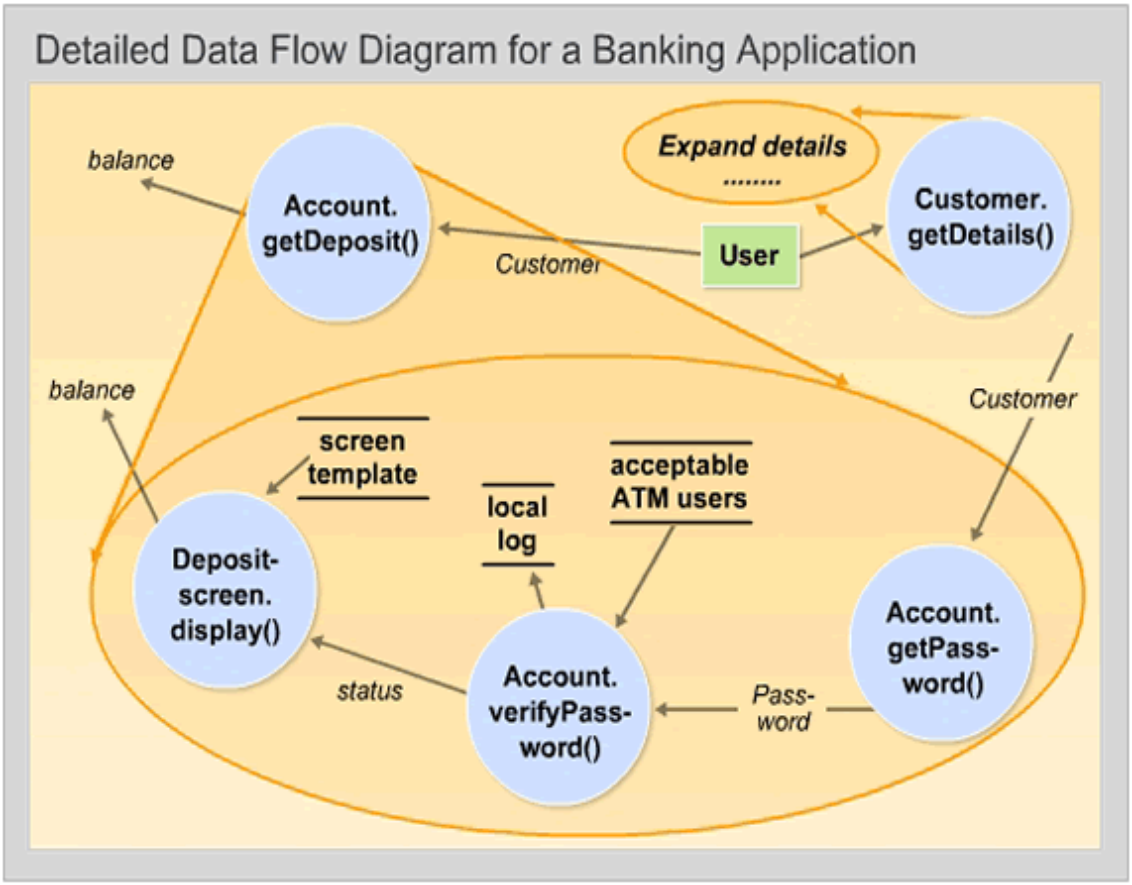
Detailed Data Flow Diagrams

To relate data flow models to classes, we map each processing element to a method of a class, as shown in the second example below.

Data Flow models can be *telescoped*. Note how the processing elements from the DFD in the first example below are expanded in the second example.



This useful property allows us to show a high-level view, followed by successive stages containing as much detail as we wish. This avoids overwhelming the viewer. Each processing element is expanded into a more detailed DFD, and this expansion process is continued until the lowest level processing elements are reached. The latter are typically individual functions, possibly of different classes. For example, the *getDeposit()* function is expanded into three functions (getting the password, verifying it, and making a display). Two of these interact with data stores (a local log of transactions, a list of problem users, and a template for screen displays) that were not shown in the high-level DFD. Note that the data entrances and exits from the detailed expansions match those in the versions from which they are expanded.



DFDs are not helpful for all applications, although they do add much to the *Encounter* case study.

Summary of Part I: Detailed Design Principles

Summary

- Business objects and architectures are used as input to the detailed design process that specifies detailed design classes
- Detailed designs should be sufficient to code from: for each class, detailed design specifies its data fields, methods, and class relationships
- Specify the methods in sequence diagrams
- Expand data flow diagrams to show all details

Part II: Specifying Detailed Designs

Introduction

met_cs682_10_fa2_ebraude_w6l2 video cannot be displayed here

The goal of detailed design is to provide a complete blueprint from which a system can be constructed. A good house blueprint leaves the builder with as few doubts as possible about the intentions of the designer, and the same is true for detailed system design.

Specifying Platforms, Classes, Functions, and Algorithms

A design must specify in complete detail the platforms on which the system is intended to run. This includes client platforms and software, server platforms and software, communications platforms, and software, and database platforms and software. A platform can be specified by make, model, etc. Sometimes we want to account for upgrades in hardware that can occur while the design is being finalized.

The goal of this lecture is to cover the following topics:

- Techniques for specifying platforms and classes in detail
- Techniques for specifying methods and algorithms in detail
- The IEEE Standard for Detailed Design
- Techniques for changing the project plans using results of design

Specifying Classes in Detail

UML allows us to specify classes to an extent. The class diagram defines classes, their relationships (inheritance, aggregation and association), names and types of data fields (attributes), and names of the methods. Still, it does not entirely define each class.

The outline below provides typical steps in carrying out detailed design for each class, and the succeeding text explains the steps in detail.

Specifying a Class

1. Gather the attributes listed in the SRS
    - This can be done more easily if the SRS is organized by class
2. Add additional attributes required for the design
3. Name a method corresponding to each of the requirements for this class
    - Again, this is easy if the SRS is organized by class
4. Name additional methods required for the design
5. Show the attributes and methods on the object model
6. State class invariants and method preconditions and postconditions (see next section)

The detailed class diagrams should include all attribute and operation names, signatures, visibility, return types, etc. Some required attributes may be left to the discretion of the implementers. It is also customary to omit accessor functions (e.g., *getSize()* and *setSize()*), since these can be inferred from the presence of the corresponding attributes (in this case, the name of the attribute is size).

UML tools have the benefit of allowing designers to suppress (i.e., to not show) certain elements of the figure—e.g., the “responsibilities” section or the variable types. Many tools allow designers to view only the classes and their relationships in order to get the big picture.

One useful manner in which to specify classes is through the CORBA Interface Definition Language (IDL). This is a standard, textual format for specifying the interfaces provided by collections of classes, their attributes, and their functions. For the specification of IDL, see The Object Management Group.

In some organizations, detailed designs are specified by providing code without function bodies. This is sometimes true of agile programmers. The advantage of this procedure is that there is no need to translate detailed specifications into code. A disadvantage is that a code form (even when it consists almost exclusively of comments) is somewhat less readable than straight prose. The functions in code form are then filled out at implementation time by programmers. The design form can be regenerated (reverse engineered) from the code. You should try this way of specifying detailed designs.

Specifying Methods and Algorithms in Detail

One way to control the behavior of functions is to use *function invariants*. These are assertions about relationships among variables that functions are guaranteed to obey. An example from *Encounter* is the following possible invariant for the *adjustQuality()* method.

```
Invariant: the sum of the values of the qualities.
```

In other words, when the value of one quality is changed with *adjustQuality(),* the values of the remaining qualities change in a way that leaves their sum unchanged.

An effective way to specify functions is by means of preconditions and postconditions. Preconditions specify the relationships among variables and constants that are assumed to exist prior to the function's execution; postconditions specify these relationships after the function's execution.

This means that enforcing preconditions is the job of the calling function, and enforcing the postconditions is the responsibility of the function being called by the caller. For example, the function *doWithdrawal(int withdrawalAmountP)* of an *Account* class could be specified as shown in the example below. Note that the bank's definition of “available funds” does not include the customer's overdraft privilege. Every invariant can be replaced by repeating the invariant assertion among both the preconditions and the postconditions.

Specifying Functions: *withdraw() in Account*

Invariant of *withdraw()*:

availableFundsI = max(0, *balanceI*)

Precondition*:

withdrawalAmountP >= 0 AND

balanceI - *withdrawalAmountP*

>= *OVERDRAFT_MAX*

Postcondition*:

balanceI' = *balanceI* - *withdrawalAmountP*

Conventions used:
xI denotes an attribute;
xP denotes a function parameter;
x' is the value of *x* after execution;
X denotes a class constant

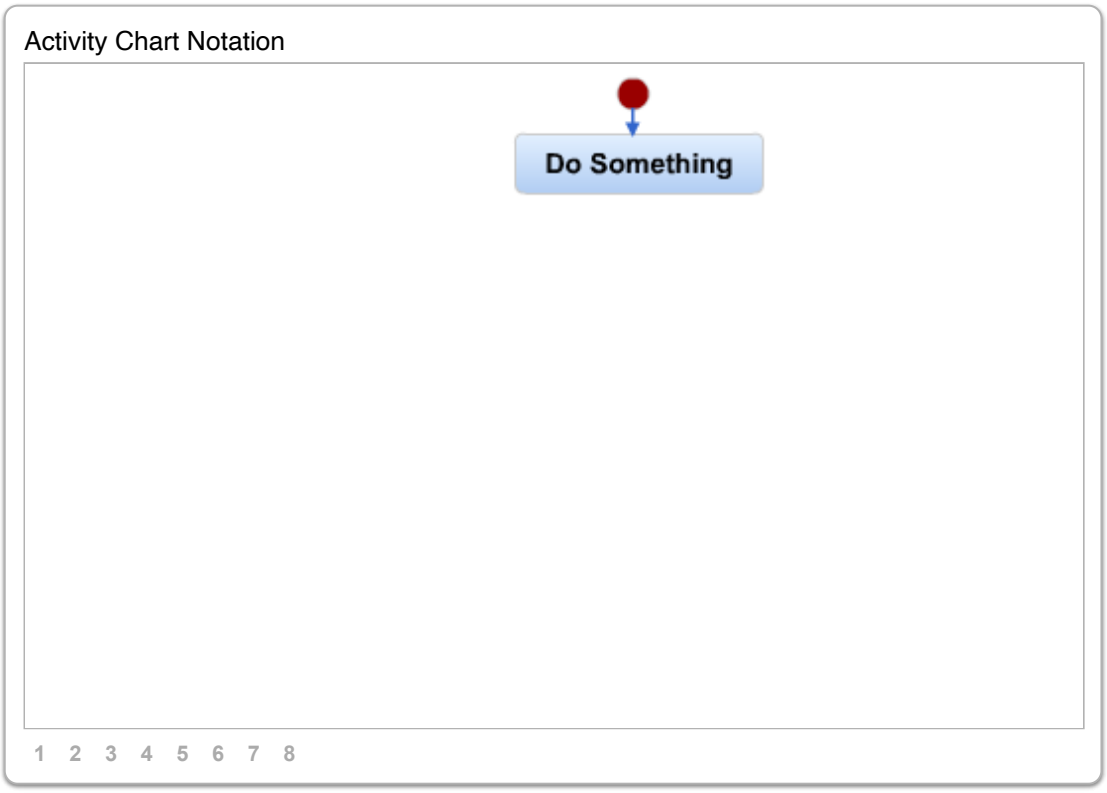
*The function invariant is an additional pre- and post-condition

We can specify non-trivial *algorithms* at detailed design time. Activity diagrams (“advanced flowcharts”) are one way to do this. They can encompass one or more methods in one or more classes.

Activity Diagrams

Flowcharts are among the oldest graphical methods for depicting algorithms. The UML uses an extended form of flowcharts called **Activity Diagrams**.

The notation for activity diagrams is shown below. This example includes parallelism, showing that the activities *Do a Task* and *Do Another Task* operating in parallel. Control is not passed to *Do Even More* until both have completed.



The following example uses Visio to show the flowchart for a social network application that updates the network when certain text is posted.

Pseudocode

Pseudocode is a means of expressing an algorithm textually without having to specify programming language details. As an example, pseudocode for a hypothetical automated X-ray controller is shown below. An advantage of pseudocode is that it is easy to understand, but it can also be made precise enough to express algorithms. Another advantage is that algorithms can be inspected for correctness independently of the clutter of language. A third advantage is that defect rates in pseudocode can be collected and used as a predictor for defect rates in the product, using historical defect data.

The rules for writing pseudocode are flexible. We outline the algorithm as a sequence of statements. These statements might include arithmetic expressions or verbal descriptions of operations. When an action should be performed repeatedly while some condition holds true, terms like "while," "for," or "for each" are used to express looping. When an action should be performed if a certain condition holds true, terms like "if" or "else" could be used. The example above uses the terms "ENDIF" and "ENDFOR" to denote the end of the scope of these control constructs. This is a good practice that makes pseudocode easier to understand. However, it is not mandatory. Many developers rely on appropriate indentation instead.

When part of the algorithm is performed by calling another method that implements this part, the name of the method should be used. For object-oriented systems, where the job is performed by cooperating objects, it is also important to indicate to what object this method belongs. We do this by indicating the name of the object (or the class) and the name of the method separated by the dot. For the example above, let us assume that this pseudocode describes method *runXrayMachine()* that belongs to class Controller. The public constants that are used to check the safety of the setup are called CRITICAL_NUM_MICROSECS and CRITICAL_POWER and belong to class XRayPolicies. We also assume that method *getApprovalForLongExposure()* belongs to class Supervisor and method *applyPower()* belongs to class XRayMachine, and that objects of these classes are components of class Controller. Then the pseudocode above can be expressed in the following way:

```
Controller.runXRayMachine(double numMicrosecs, double powerLevel)
  if (numMicrosecs >  XRayPolicies.CRITICAL_NUM_MICROSECS) then
     approved = supervisor.getApprovalForLongExposure(numMicrosecs)
     if approval is false then
          throw exception("no supervisor approval")
  if (powerLevel > XRayPolicies.CRITICAL_POWER)
     throw exception ("power level exceeded")
  time = 0
  for each microsecond, while (time < numMicrosecs)
     xRayMachine.applyPower(powerLevel)
     increment time by 1 microsecond
```

When computational algorithms are rather simple, pseudocode mostly describes cooperation among objects similar to a detailed sequence diagram. For example, the *encounterForeignChar()* method described in Module 6 part II, that calls methods *displayForeignChar()* and *execute()* can be represented by the following pseudocode:

```
EncounterGame.encounterForeignChar()
   EncounterCase.displayForeignChar()
   engagement = new Engagement()
   engagement.execute()
```

Similarly, method *execute()* of class Engagement that calls methods *setPlayerQuality()*, *setForeignQuality(),* and *displayResult()* can be described by the following pseudocode:

```
Engagement.execute()
   EncounterCast.setPlayerQuality()
   EncounterCast.setForeignQuality()
   engagementDisplay = new EngagementDisplay()
   engagementDisplay.displayResult()
```

Pseudocode for methods *setPlayerQuality()* and *setForeignQuality()* would be needed if these operations are the least bit complex.

When algorithms are more computationally intensive, pseudocode might look quite similar to real code. For example, method *balanceProjection()* that belongs to class Projections computes the accumulated balance using the values of initial balance, yearly rate (simple interest), and the number of years of accumulation. Its pseudocode might look like this:

```
double Projection.balanceProjection(double initBalance,
                              double rate, int years)
   int year = 0
   while year < years
      interest = balance * rate
      add interest to balance
      increment year by 1
   return balance
```

This method might be called by its client code using the following statement:

projectedBalance = Projection.balanceProjection(1000000.0,0.03,10)

Pseudocode Review

1. Begin with the description of the algorithm in the requirements or in the detailed sequence diagram
2. Introduce additional details if necessary
3. Express the algorithm in terms of
    a. Sequence of operations
    b. Loop statements, e.g., doing something while some condition is true
    c. Conditional statements, e.g., doing something if some condition is true
    d. Method calls to methods of the same or other classes

Using a system that controls inventory of items for a factory that makes tablet computers, as described earlier, the following pseudocode describes a function that determines in real time (i.e., without interrupting the process) what parts need to be reordered. Assume that the pseudocode is for a function *usePart()* in the class *TabletPart* that is executed every time a part is taken for a tablet assembly. Assume that the program performs other operations while the reorder process is executed. This is *parallel programming* and is sometimes indicated with “Thread.” One interprets this as *adjustInventory()* being started and the process continuing with the next line (“WHILE noResponse”) without waiting for *adjustInventory()* to complete. In non-parallel (i.e., conventional) programming, it is assumed that a line can execute only when the previous line has completed.

```
getPartType()
adjustInventory() (Thread)
WHILE noResponse
    continueProcessing()
    IF timeout
        alertOrderingStaff()
IF reorderingNeeded
    orderParts()

<next step>
```

Many organizations use inspected pseudocode as annotated comments in the source code listing. Tools are then able to extract the pseudocode from the source.

The example below contains a flowchart for a setName() method, showing the most commonly used flowchart constructs: Decisions (diamonds) and processes (rounded rectangles).

Activity Chart Example

Deciding Whether to Use Flowcharts and Pseudocode

Flowcharts and pseudocode each have the advantages and disadvantages listed below. The decision whether or not to use pseudocode and/or flowcharts depends on factors particular to the application. Some developers shun flowcharts as old-fashioned, but flowcharts and pseudocode can be worth the trouble, at least for selected parts of applications where they help to produce better quality products.

Advantages of Pseudocode and Flowcharts

- Provide documentation that helps clarify complex algorithms
- Impose increased discipline on the process of documenting detailed design
- Provide additional level at which inspection can be performed
    - Help to trap defects before they become code
    - Increase product reliability
- May decrease overall costs

Disadvantages of Pseudocode and Flowcharts

- When design and/or code changes, they have to be maintained to be consistent
- Introduce error possibilities in translating to code
- May require a tool to extract pseudocode and facilitate drawing flowcharts

IEEE Standard 890 for Detailed Design

Recall IEEE standard 890 for Software Design Documents that was shown in Module 5 on software architecture and below (see Module 5 topic IEEE Standards for Expressing Designs). This format for the detailed design section of this document consists of specifying a description of each module (package) and specifying a detailed description of each data part. For OO designs, data part can be replaced with a detailed description of each class.

Effects of Detailed Designs on Projects

Once a detailed design is in hand, the project plan can be made more specific in several respects. In particular, cost estimation can be made much more precise, schedules can be broken down into tasks, and tasks can be allocated to individuals. The following lists most of the important updates to be performed once detailed design is complete.

Bring the Project Up-To-Date After Completing Detailed Design

1. Make sure the SDD reflects latest version of detailed design, as settled on after inspections
2. Provide details needed to complete the schedule (SPMP)
3. Allocate precise tasks to team members (SPMP)
4. Improve project cost and time estimates (see below)
5. Update the SCMP to reflect the new parts
6. Review process by which the detailed design was created, and determine improvements. Include...
    1. Time taken; broken down to include:
        - Preparation of the designs
        - Inspection
        - Change
    2. Defect summary
        - Number remaining open, found at detailed design, closed at detailed design
        - Where injected, include previous phases and detailed design stage

As an example, consider the effects of completing the detailed design of a system that tracks the parts for tablet assembly outlined previously.

1. We would ensure that the Software Design Documents are up-to-date.
2. We can specify when each part (e.g., the inventory alert function) should be complete.

3. We can allocate particular tasks to team members (SPMP) For example " John Jones to implement and test inventory alert function by January 1."
4. The accuracy of estimation is always improved when design details are known because some uncertainties have been removed. For example, we can better estimate when the tablet assembly system will be completed since we know its parts.
5. The design parts are entered into the configuration management system, which tracks version numbers. Configuration management systems are usually thought of in connection with code but they are also useful for documentation because designs evolve over time, and their sections need to be coordinated and tracked.
6. It is ideal if the team can spend some time reviewing the process with an eye to improving it the next time around. We'd ask, for example, how successfully the detailed design for "Tablet Assembly" turned out and how we could better organize the detailed design process. In particular, we would look at the time taken to perform design, often broken down to include the following.
    1. Preparation
    2. Inspection of the designs
    3. Change in designs
    4. What phase in the process introduced the defect

Since we can estimate the number and size of the methods involved in the application using detailed designs, a more precise estimation of the job size is possible. Job costs can then be inferred from the size. The list below shows steps for carrying this out.

Estimating Size and Time From Detailed Designs

1. Start with the list of the methods of all classes on all platforms.
2. Ensure completeness, otherwise underestimate will result.
3. Estimate the lines of code (LoC) for each:
    - Classify as *very small, small, medium, large, very large*
        - *Normally in +/- 7% / 24% / 39% / 24% / 7% proportions.*
    - Use personal data to convert to LoC
        - Otherwise use Humphrey's table below
4. Sum the LoC
5. Convert LoC to person-hours
    - Use personal conversion factor if possible
        - Otherwise use published factor
6. Ensure that your estimates of method sizes and time will be compared and saved at project end

This algorithm assumes that you can compute the size of the project in source lines of code (LoC) by reviewing each method in your detailed design and classifying these methods as very small, small, medium, large, and very large. This classification is, of course, quite subjective, but hopefully the errors will cancel each other out.

The next step is to decide how many lines of code each method would require to implement. Again, this is subjective, and the accuracy of this estimate depends on experience. This is why the algorithm above says, “use personal data.“ If personal data is not accumulated yet, you can use the table developed by Humphrey (below) that predicts the required lines of code for methods of different sizes (very small, small, etc.) and of different nature (whether the primary goal of the method is calculation, data processing, input/output, etc.—see the next section for a more detailed discussion and an example).

The next step is to convert the total number of lines of code in the application into the number of person-hours (or person-months) required to implement and debug the application. Here also the best guide is personal experience. For many projects, the so called COCOMO model can now be used again to refine the estimate of job duration.

The COCOMO model (Constructive Cost Model) was developed by Barry Boehm of TRW in 1981 as the result of studying more than 60 projects of different sizes (from 2,000 to 100,000 lines of code). He developed a statistical formula that would give an estimate of effort required to implement the given number of lines of code.

In 2001, Boehm revised his formula to reflect the projects that deviate from the waterfall development model and also to reflect the use of personal computers rather than mainframe computers. He also introduced five so-called scale drivers, which modified the estimate to reflect the team experience with similar project, team cohesion, development process maturity, and other factors.

The accuracy of these estimates is limited, especially because the projects used for deriving the model were run by DOD contractors working for the government. However, they are often used to evaluate the effort required. It is best to use personal data to estimate the LOC using size descriptors for methods as “very small,” “small,” etc., jobs. In the absence of such data, department, division, or corporate data can be used. The Humphrey's table below (Humphrey, 1995) applies to C++ LOC per method. The table is organized according to the job performed by the method as (a) calculation, (b) data, (c) I/O, (d) logic, (e) set-up, and (f) text. More about these method types in the next section.