Object-Oriented Analysis & Design Using The Unified Modeling Language

Student Workbook

itcourseware
Training Materials For IT Professionals

OBJECT-ORIENTED ANALYSIS & DESIGN USING THE UNIFIED MODELING LANGUAGE

Lynwood Wilson

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Contributing Authors: John McAlister, Jamie Romero, Rick Sussenbach, and Rob Seitz.

Editors: Danielle Hopkins and Jan Waleri

Editorial Assistant: Dana Howell

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, offering comments, suggestions, criticisms, and insights.

CONTENTS

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Apply the principals and practices of Object-Oriented Programming.
- Use modeling in analysis and design, particularly in visual modeling.
- Use the Unified Modeling Language to create visual models of business problems and software solutions.
- Design programs with objects.
- Create more flexible and more maintainable software systems at lower costs.

COURSE OVERVIEW

- **Audience**: Programmers, analysts and software designers.

- **Prerequisites**: Some exposure to the problems of analysis and design. Experience with structured analysis and design, as well as object-oriented programming, would be helpful.

- Note: This course is based on UML Version 2.0, and occasionally mentions features from previous versions.

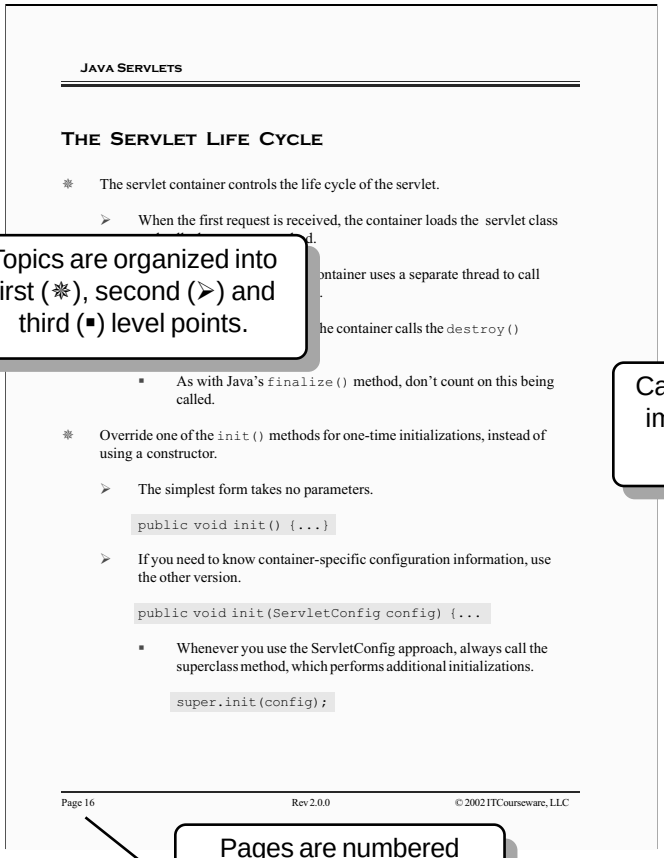
USING THE WORKBOOK

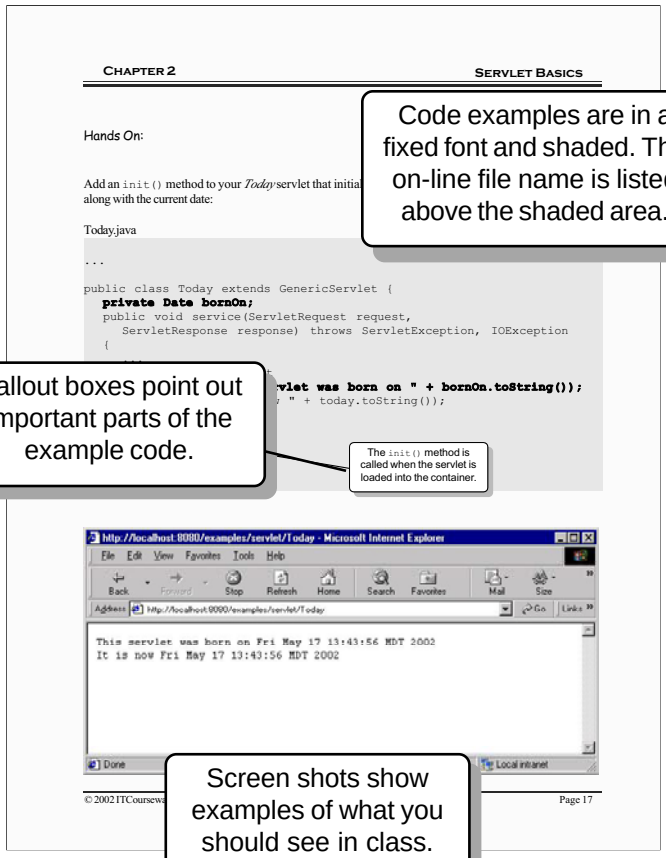
This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as online if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.





Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Topics are organized into first (❋), second (➢) and third (▪) level points.

Callout boxes point out important parts of the example code.

Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

SUGGESTED REFERENCES

Ambler, Scott W. 2002. ***Agile Modeling: Effective Practices for Extreme Programming and the Unified Process***. John Wiley & Sons, New York, NY. ISBN 0471202827.

Beck, Kent and Cynthia Andres. 2004. ***Extreme Programming Explained: Embrace Change***. Addison-Wesley, Reading, MA. ISBN 0321278658.

Bellin, David and Susan Simone. 1997. ***The CRC Card Book***. Addison-Wesley, Reading, MA. ISBN 0201895358.

Bentley, Jon. 1999. ***Programming Pearls***. Addison-Wesley, Reading, MA. ISBN 0201657880.

Booch, Grady, James Rumbaugh and Ivar Jacobson. 2005. ***The Unified Modeling Language User Guide, Second Edition***. Addison-Wesley, Reading, MA. ISBN 0321267974.

Buschmann, Frank, et al. 1996. ***Pattern-Oriented Software Architecture, Volume 1: A System of Patterns***. John Wiley & Sons, New York, NY. ISBN 0471958697.

Coad, Peter and Edward Yourdon. 1990. ***Object-Oriented Analysis, Second Edition***. Yourdon Press/ Prentice Hall, Englewood Cliffs, NJ. ISBN 0136299814.

Coad, Peter and Edward Yourdon. 1991. ***Object-Oriented Design***. Yourdon Press/Prentice Hall, Englewood Cliffs, NJ. ISBN 0136300707.

Cockburn, Alistair. 2001. ***Agile Software Development***. Addison-Wesley, Reading, MA. ISBN 0201699699.

Cockburn, Alistair. 2000. ***Writing Effective Use Cases***. Addison-Wesley, Reading, MA. ISBN 0201702258.

Demarco, Tom and P. J. Plauger. 1979. ***Structured Analysis and System Specification***. Prentice Hall, Englewood Cliffs, NJ. ISBN 0138543801.

Fowler, Martin, et al. 1999. ***Refactoring: Improving the Design of Existing Code***. Addison-Wesley, Reading, MA. ISBN 0201485672.

Fowler, Martin. 2003. ***UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition***. Addison-Wesley, Reading, MA. ISBN 0321193687.

Freedman, Daniel P. and Gerald M. Weinberg. 1990. ***Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products.*** Dorset House Publishing Company, New York, NY. ISBN 0932633196.

Gamma, Erich, et al. 1995. ***Design Patterns: Elements of Reusable Object-Oriented Software***. Addison-Wesley, Reading, MA. ISBN 0201633612.

Highsmith III, James A. 1999. ***Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.*** Dorset House, New York, NY. ISBN 0932633404.

Hunt, Andrew and David Thomas. 1999. ***The Pragmatic Programmer: From Journeyman to Master***. Addison-Wesley, Reading, MA. ISBN 020161622X.

Jacobson, Ivar, Grady Booch and James Rumbaugh. 1999. ***The Unified Software Development Process***. Addison-Wesley, Reading, MA. ISBN 0201571692.

Jones, T. Capers. 1994. ***Assessment and Control of Software Risks***. Prentice Hall PTR, Englewood Cliffs, NJ. ISBN 0137414064.

Kernighan, Brian W. and Rob Pike. 1999. ***The Practice of Programming***. Addison-Wesley, Reading, MA. ISBN 020161586X.

Kruchten, Philippe. 2003. ***The Rational Unified Process: An Introduction, Third Edition***. Addison-Wesley, Reading, MA. ISBN 0321197704.

Larman, Craig. 2004. ***Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0131489062.

McConnell, Steve. 2004. ***Code Complete: A Practical Handbook of Software Construction, Second Edition***. Microsoft Press, Redmond, WA. ISBN 0735619670.

McConnell, Steve. 1997. ***Software Project Survival Guide***. Microsoft Press, Redmond, WA. ISBN 1572316217.

McLaughlin, Brett D., Gary Pollice and David West. 2006. ***Head First Object Oriented Analysis and Design***. O'Reilly Media. Sebastopol, CA. ISBN 0596008678.

Meyer, Bertrand. 2000. ***Object-Oriented Software Construction, Second Edition***. Prentice-Hall, Englewood Cliffs, NJ. ISBN 0136291554.

Miles, Russell and Kim Hamilton. 2006. ***Learning UML 2.0***. O'Reilly Media, Sebastopol, CA. ISBN 0596009828.

Page-Jones, Meilir. 1988. ***The Practical Guide to Structured System Design, Second Edition***. Prentice Hall PTR, Englewood Cliffs, NJ. ISBN 0136907695.

SUGGESTED REFERENCES (CONT'D)

Page-Jones, Meilir. 1999. ***Fundamentals of Object-Oriented Design in UML***. Addison-Wesley, Reading, MA. ISBN 020169946X.

Pilone, Dan and Neil Pitman. 2005. ***UML 2.0 in a Nutshell***. O'Reilly Media, Sebastopol, CA. ISBN 0596007957.

Rumbaugh, James, Ivar Jacobson and Grady Booch. 2004. ***The Unified Modeling Language Reference Manual, Second Edition.*** Addison-Wesley, Reading, MA. ISBN 0321245628.

Shlaer, Sally and Stephen J. Mellor. 1991. ***Object Lifecycles: Modeling the World in States***. Yourdon Press/Prentice Hall, Englewood Cliffs, NJ. ISBN 0136299407.

Shlaer, Sally and Stephen J. Mellor. 1988. ***Object-Oriented Systems Analysis: Modeling the World in Data***. Yourdon Press/Prentice Hall, Englewood Cliffs, NJ. ISBN 013629023X.

Wirfs-Brock, Rebecca, Brian Wilkerson and Lauren Wiener. 1990. ***Designing Object-Oriented Software***. Prentice Hall, Englewood Cliffs, NJ. ISBN 0136298257.

http://alistair.cockburn.us/index.php/Resources_for_writing_use_cases
http://www.agilealliance.org
http://www.junit.org
http://www.rational.com

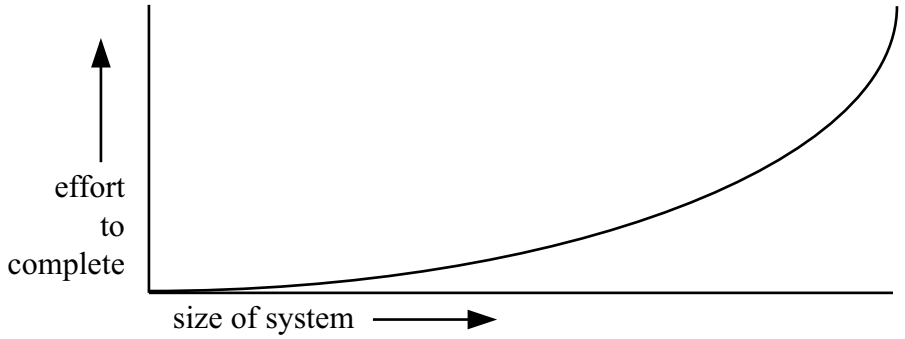
CHAPTER 2 - INTRODUCTION TO ANALYSIS AND DESIGN

OBJECTIVES

- Identify essential problems and tasks of software development.
- Describe basic concepts of modularity and abstraction.
- Outline the concepts of Objects and Object-Oriented Programming.

WHY IS PROGRAMMING HARD?

- It's complicated.
  - Computer programs are among the most complex things people build.
  - Most people can only think of 7 ± 2 things at a time.
- It gets more complicated as the system gets bigger.
  - Why is the curve of *effort vs. size* exponential?
    - More communication links among programmers, designers, analysts, clients, etc.
    - More communication links among modules in the system.
  - The most efficient software project is a single programmer working on a program no one else will use. There's no communication.



- Modularity helps to flatten this curve.
  - With good design and abstraction you can work on a module — a part of the program — as though it were a single small program, and thus stay toward the left end of the above graph.
  - This works even when it's a high-level module that uses several low-level modules, if you properly define and constrain the interfaces.

Miller, G. A., "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *Psychological Review*, Vol. 63, March 1956, pp. 81 - 97. Miller's research showed that most people can only hold seven plus or minus two things in working memory at a time. Think about it when you are designing menus. Most of our models and drawings should contain no more than nine different top-level artifacts.

Blaise Pascal, they say, once closed a letter by saying, "I'm sorry this letter is so long. I didn't have time to make it shorter." Most of what we create will benefit from taking the time to make it shorter. Text, models, and code alike are clearer and communicate better if we take the time to make them concise, precise, and elegant. Take the time.

You have a client even if you aren't a consultant or a contractor. Your client is the person who manages the group that needs the system you are working on. Often the client is the person who asked for the work, or the person who pays for it or whose division pays for it. Usually the client is the person who knows best what the top-level requirements are. The users are important, and your system must satisfy them, but they often do not know all the needs of the business.

THE TASKS OF SOFTWARE DEVELOPMENT

- Figure out what problem to solve or what system to build.
  - Analysis
- Build the system to solve the problem.
  - Design
  - Implementation / Programming
- Analysis is harder.
  - Most of the problem is communication: communication with the computer and communication with people. The computer is easier, in spite of (or perhaps because of) being so literal and requiring perfection in each detail.
- What tools do we use to manage complexity and help with communication? (Not just in software, but everywhere.)
  - Modules — Break a job into simpler components such that if we complete the components the job will be done.
  - Models — Represent the problem, the solution, and their component parts in such a way as to enable us to work with the important aspects (for a particular task) and ignore the rest.
  - Formal Process — Organize the work so that we do everything important with a minimum of non-productive effort.

We divide the analysis into two parts: Domain Analysis, and Requirements or Specification Analysis.

Domain Analysis is finding out about the business and its processes. Building a common vocabulary with the domain people, users, clients. Understanding the context within which our proposed system must operate. And if there is an existing system that ours is to replace, we should understand that as well.

Requirements or *Specification Analysis* describes the system that will solve the client's problems, characterizing it in such a way and to such a depth that if we meet the specification we satisfy the client.

It's not possible to perform either of these perfectly or completely. This is one reason we must deal with change throughout the process, as we discover missing, incomplete, inconsistent, or erroneous specifications.

MODULES

- A *module* is a part of a program or model that can be considered as an entity separate from the rest.
  - A module has a purpose.
  - A module has a specified interface through which it interacts with the rest of the system.
  - A module is abstract. It hides its implementation, the details of its operation, from the rest of the system.
- A module should have high cohesion.
  - It should do one thing, have a single responsibility, at its level of abstraction.
- A module should have low coupling.
  - It should be a black box. The modules that use it need not understand its internal operation.
  - Its external interface should be simple, narrow, and elegant.
- The kind of module we will be most interested in during this course is the object.
  - We will also see higher-level modules that contain multiple objects.
  - Objects contain attributes (data) and methods (functions), and these are modules, too.

MODELS

- The kind of model we will talk about most is an abstraction made up of modules and relationships between them.

  - *Abstraction* is ignoring those aspects of something that do not contribute to your task in order to focus on those aspects that do.
  - A model displays a few aspects and hides all the others.
  - We try to use between nine and five important modules in each model.
  - These modules are all at about the same level of abstraction.
  - The parts we choose contribute to some particular understanding of the thing modeled.

- For example, the mathematics operations in our programs are abstractions. When we add two simple integers in a program we don't know (or care) what's going on at the level of the memory and the CPU registers. Instead, we are free to concentrate on the logic of the program and on the way this addition will contribute to some result.

  - If we had to deal with addition at the byte level every time we added two integers it would add considerable complexity to our task and divert our attention from the higher-level logic.

- This abstraction allows us to focus on just a few things at a time, and also allows us to work at a constant level.

  - People seem to perform better if they can work at a fairly constant level of abstraction. This seems particularly valuable in communication.
  - And don't forget the magic number 7 ± 2.

- One of the most important attributes of a model, regardless of its perspective, level, or purpose, is that it is clear, and can be easily read and understood.

  - This is as important as accuracy. If you cannot understand it you will never know if it is accurate.
  - Part of this is the artistic quality that we call "elegance." Spend a little extra effort to make it clear and clean and pretty and simple. Elegant.

When Ford proposes to build a new car, they first build models of it. In the old days they'd build a wooden frame, cover it with clay, and carve it into the shape of the proposed body. First in miniature, later full size. They'd probably model several variations. Finally they would paint the final version and photograph it with pretty people standing around it. Was this a car? Of course not. What was it? An abstract representation of a single aspect of a car: the appearance. In other words, a model. Today most of this is done with computers, but the ideas are the same.

This isn't the only model of a new car, although it often seems like the most important one. (As with computer programs the visual impression is crucial.) Ford will also build a computer model of the suspension with all its parts and characteristics. This will be a working model (although today it's a computer simulation with variables for spring rates and pivot locations). Then they can exercise it by subjecting it to various simulated bumps at various simulated speeds. They can calculate the forces on the parts to see if they are strong enough, and the forces on the rest of the car to see if the passengers will like it. Perhaps there will also be wind tunnel models, and coupled with the mathematical models of the proposed engine they can calculate the performance and fuel mileage. Thus Ford can find out a lot about their design and can make certain kinds of decisions fairly cheaply, before making a commitment to metal and tooling.

Note that each of these models has a purpose and each is an abstraction, each represents only one part of the car.

Some models become central to your project and will be maintained and used forever. Others will be built only to solve an immediate problem and subsequently discarded. Maintaining a model can be more expensive than building it (like a program). Don't maintain it unless you make a profit on the effort. If you choose not to maintain it, get rid of it immediately. The model is often less valuable than the thought and planning that goes into it.

The ultimate model of software is the code. In the beginning we design and model with diagrams and text. Then we write code. We find problems and shortcuts, test and debug, find better ways to do things, and do a lot of low-level design as we go. The result does not exactly follow our models. Whether we update our models, they are never exactly like the code. The code rules.

MODELING

- Our choice of models to build is driven by our perception of the problem as much as it drives our understanding of the problem.
  - One of our many challenges will be to understand what the models are telling us and change our ideas as we go forward.
  - We have to start somewhere, guided by our best guesses based on our experience, but then we must leave them behind and follow our models.
  - As we learn, we find it easier to decide what we need to model, which of our existing models may no longer be relevant, and what we need to do next.

- Determine the purpose of the model before building it.
  - Not the purpose of the thing modeled, but the purpose of this particular model in the development process.
  - Put in the model only that which contributes to the purpose of the model.
  - Don't try to tell everything you know in a single model.
  - Don't try to model everything.

- For each model we must choose the level of abstraction.
  - This too will drive our understanding and be driven by it.
  - This too will become easier, more apparent, as we proceed.

- It's useful for our models to be well grounded in reality.
  - Models derived from the domain communicate better with our clients, users, and domain experts.
  - Modules derived from the real world are more likely to be reusable.
  - Models and modules derived from the domain have a characteristic kind of unity and integrity and reality that is hard to create from scratch. They model something that exists and works rather than something we imagine might work if we could build it.

- The UML is not complete. Explore other kinds of models: Data Flow Diagrams, Entity Relationship Diagrams, etc.

What Makes a Model: Text and Graphics

In our world there are many kinds of models, although we hardly ever build wooden forms and cover them with clay. Most of our models are text or graphic representations. Sometimes we build a software model of an algorithm or a process to test its speed or storage requirements. Sometimes we model a user interface to get the user's opinion. But mostly we stick to text and graphics. Some prefer one, some the other. Most learn best with both: text with plenty of illustrations.

Static vs. Dynamic

Some of the things we model will be static and others dynamic. A static model can represent a static thing. It can also represent a dynamic thing, but incompletely. A static model of a dynamic thing is a snapshot, change frozen at a particular moment, a particular state. A dynamic model can represent a dynamic thing more completely.

Analysis vs. Synthesis

Analysis is describing something that exists and *synthesis* (design) is predicting something in the process of creation. See Herbert Simon's *Sciences of the Artificial* for a fascinating discussion of the difference between synthesis and analysis, the difference between understanding something natural and creating something artificial.

Iteration

The first model you build may be perfect. It may not. Don't just build a model and go on. Test your models, consider other possible ways to do things, try modeling the subject in another form, see if someone else thinks of it in a different way. As your experience with the problem grows, the chance of an insight that yields a better model grows too. However, beware of analysis paralysis.

Feedback

Get as much feedback as you can, soon and often. Usually this will be from the users and the client. If you are working without it you are kidding somebody. The classic mistake here is to build what we think they need. That trick never works.

PERSPECTIVE

- In software engineering we view our work and build our models from three different perspectives.

- *Domain* perspective (also called *Conceptual* or *Essential* perspective) is concerned with the domain, the context of the problem.

  - The existing business, its organization and operation.

  - An existing system that our proposed system is to replace (if any).

- *Specification* or *Requirements* perspective is about what the proposed system should do.

  - The proposed system as seen from outside, by the users and the client.

  - The requirements.

- *Design* or *Implementation* perspective describes the internal organization and operation of the system we propose to build.

  - The modules that will make up our proposed system and the way they will work together to accomplish the tasks.

- Record the perspective of your diagrams and models.

  - It may be obvious to you now, but not to someone else later (and you may be that someone else on a later project).

OBJECTS

- Objects are better modules.
- Objects give us more abstraction, better modularity, more flexibility.
  - Now it's modularity of both function and data.
  - We can encapsulate more in a module and enforce the encapsulation in ways that we could not before.
  - The modules are even further from the machine, closer to human thinking.
  - The modules can represent artifacts from the problem and the problem domain, from the world of the people rather than the world of the computer.
- Before OOP we thought first of the operation: What's it do?
  - After we worked that part out, sometimes a long time later, we thought about the data: What's it do it to?
  - In OOP the data gets at least equal attention.
- Each of these advances in abstraction, modeling, and modularity gave us the power to build larger systems with less effort by managing communication problems.
- Objects help us to program the way we actually work in the real world instead of the way we worked in school.

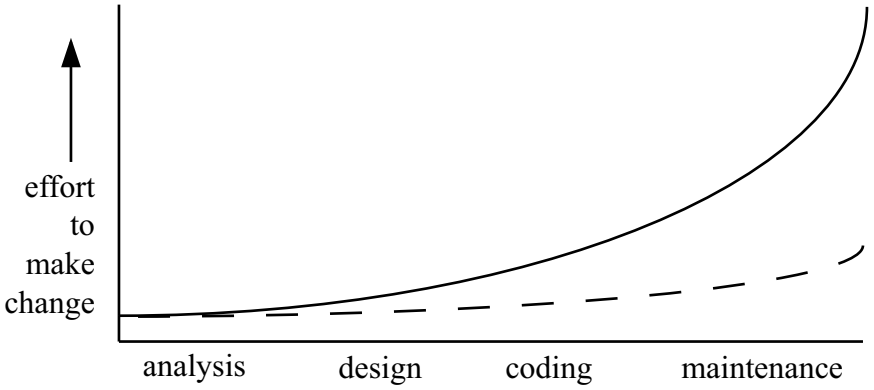
Much of this jargon comes from the Smalltalk world, once the center of OO. A Smalltalk programmer says "Object A sends message foo to object C" when a C++ programmer would say "Object A calls function foo on object C". They both mean the same thing. What a Smalltalk programmer would call a *method* a C++ programmer calls a *member function*. Physically they are functions, so physically some code in one object calls a function that belongs to another object, no matter how you say it.

Anthropomorphism in designing OO models and programs is encouraged. In fact, it's one of the advantages of objects. We all know that our programs don't have tiny people inside doing the work, but it's often useful to think of modules as having desires, responsibilities, mental state similar to that of a human.

Up until now we've talked mostly about software objects. It turns out that objects are also quite useful as modules in models for all the same reasons. One important difference, however, is that in software objects we enforce rules about information hiding and abstraction, whereas in objects used in models of businesses and such we don't try so hard to enforce these rules. It's the difference between analysis and synthesis. In software design we want to build something good. In analysis we want to model something that exists, be it good or bad. Sometimes this results in objects you wouldn't want to use in a program, but they are just right for modeling a business.

CHANGE

- Specifications change.
  - This can cause poor communication between us and them.
  - They don't know what they need until we give them what they say they want.
  - Specifications change as we (and they) learn more.
  - Specifications change as the business domain changes.
  - The later we change, the more it costs.



- The solid line above is the classical curve of effort to change the system vs. the point in the development cycle at which we begin. (The same graph describes the cost of fixing a bug vs. the length of time it went undiscovered.)
  - The solid graph is based on data from before OOP.
  - If we design and build good objects and maintain the structure of the system every time we touch it we can flatten this curve into something more like the dashed line.
  - It will always cost more to make a change as time passes, but we can keep the curve from getting so steep.

- OOP and OOD help us to do a better job in the real world, working with change instead of fighting it or pretending we can control it.

We have to handle vague and changing requirements. We cannot force our clients to work or think the way we want them to. And if we could it still wouldn't be the right thing for their businesses or ours. The world is vague and constantly changing, and the rate of change is increasing.

OO helps. If you do a good job at the object level, it's much easier to change the program later when you know more. We want to flatten that curve.

NEW PARADIGMS

- Adopting new technology such as OOP is difficult, confusing, expensive, and can frustrate both managers and programmers. Ignoring new technology is even more expensive.

- Training — More projects have foundered from too little training than from too much.

- Mentoring

  - Some shops that have a training organization often send a teacher out as mentor to a team about to begin working with objects.
  - Smaller organizations may find one or two developers who have the experience, the knowledge, and the teaching ability to season a team just getting started. Be very good to them — you don't want to lose them.
  - If there's no one available in your organization who can take the mentor role, you might consider bringing in a consultant from outside: full time, if the project is big enough, or part time to sit in on reviews and offer ideas.
  - A good mentor can make the difference between success and failure. There are insights that seem to come only from experience. And it's time well spent to look for just the right person: someone with both the knowledge and the ability to communicate it.

- Learn something new with each project, even if there is a cost. (There's always a cost.)

  - It should pay off later.
  - Sometimes what you learn is that a particular thing won't pay off later, and that is valuable too.
  - Sometimes you are better off to let someone else make that discovery and tell you about it, but it's hard to see that ahead of time.
  - Read the literature, the books and the magazines and the websites. You are crippled if you don't keep up.

LABS

❶ Have you ever participated in a software project that failed? What caused it?

❷ Some gurus and their books consider models and requirements documents to be deliverable products of the development process. What do you think?

CHAPTER 9 - ACTIVITY DIAGRAMS

OBJECTIVES

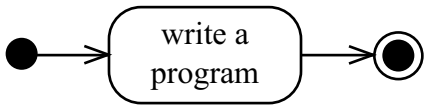
- Model activities, such as complicated algorithms and multi-branched user interfaces.
- Model parallelism.

ACTIVITY NOTATION

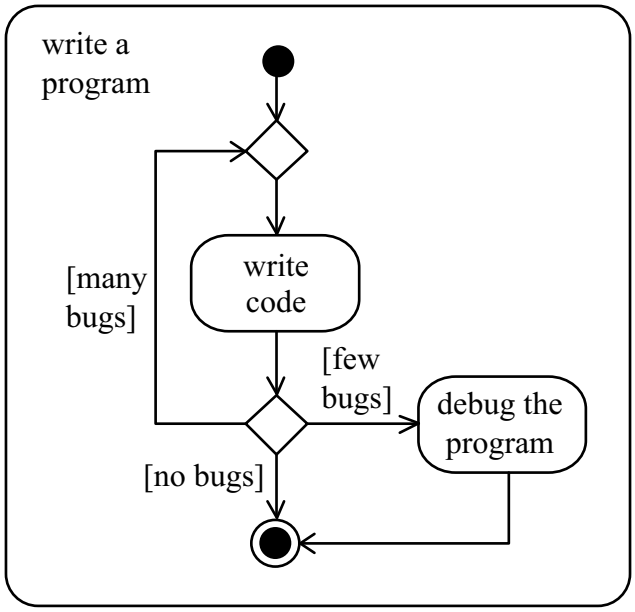
- Activity diagrams are similar to flow charts.

- The notation of activity diagrams is much the same as that of state diagrams.

  - The main difference is that most of the nodes in a state model are states (although some may represent activities) while most of the nodes in an activity model represent activities (although some of them may represent states).



- You can add details for the activity inside a rounded rectangle.



- Partly because most nodes are activities, most transitions are triggerless. When the activity of the node is finished, the transition occurs.

- There can be guard conditions like those in state charts.

- Activity diagrams, like state charts, usually flow from left-to-right or from top-to-bottom.

The guard conditions should not allow ambiguity about what will happen in any situation.

A guard condition **else** will be true if all other guards for transitions from the activity are false.

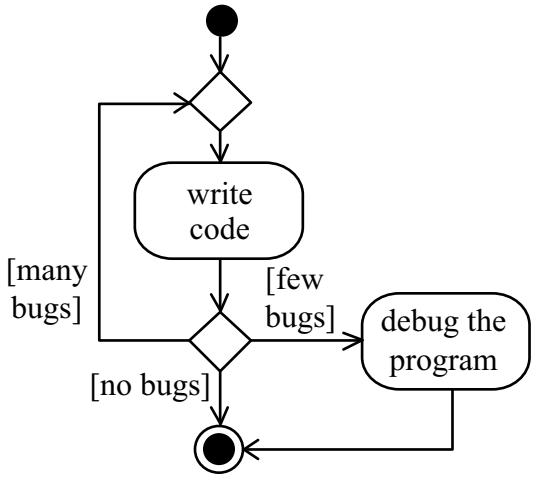
Transitions may have actions on them (**/action**) like transitions in state diagrams, but it is unusual.

An activity takes up time; a transition (even if it has an action) is atomic and takes no time.

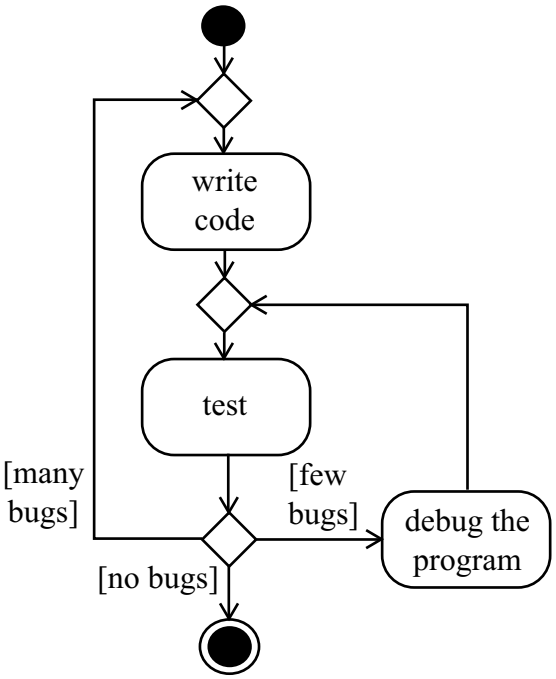
Activities have no internal transitions, or entry or exit actions.

DECISIONS AND MERGES

- You can express testing of a program using a decision diamond.



- If you believe the program needs to be checked again after debugging (whatever are you thinking?) you can run the path back around through another diamond, called a *merge*.



  - If it's got one in and more than one out, it's a decision and it'll have guard conditions on the transitions from it.
  - If it's got one out and more than one in, it's a merge and it won't have guard conditions on the transition from it.

Forks and Joins

- One of the strengths of activity diagrams is their ability to model parallel processes.

- When a single transition encounters a *fork* (heavy vertical or horizontal line) it continues along the paths on the other side in parallel and asynchronously.

  - This used to be called a *synchronization bar*.

  - Parallel paths may be traversed any way at all.

  - In the example on the facing page, you can first reach for a cup with your right hand and then start pouring coffee into the filter with your left while you set the cup on the counter and reach for the water. Or you can complete the paths one at a time from left-to-right. Or right-to-left.

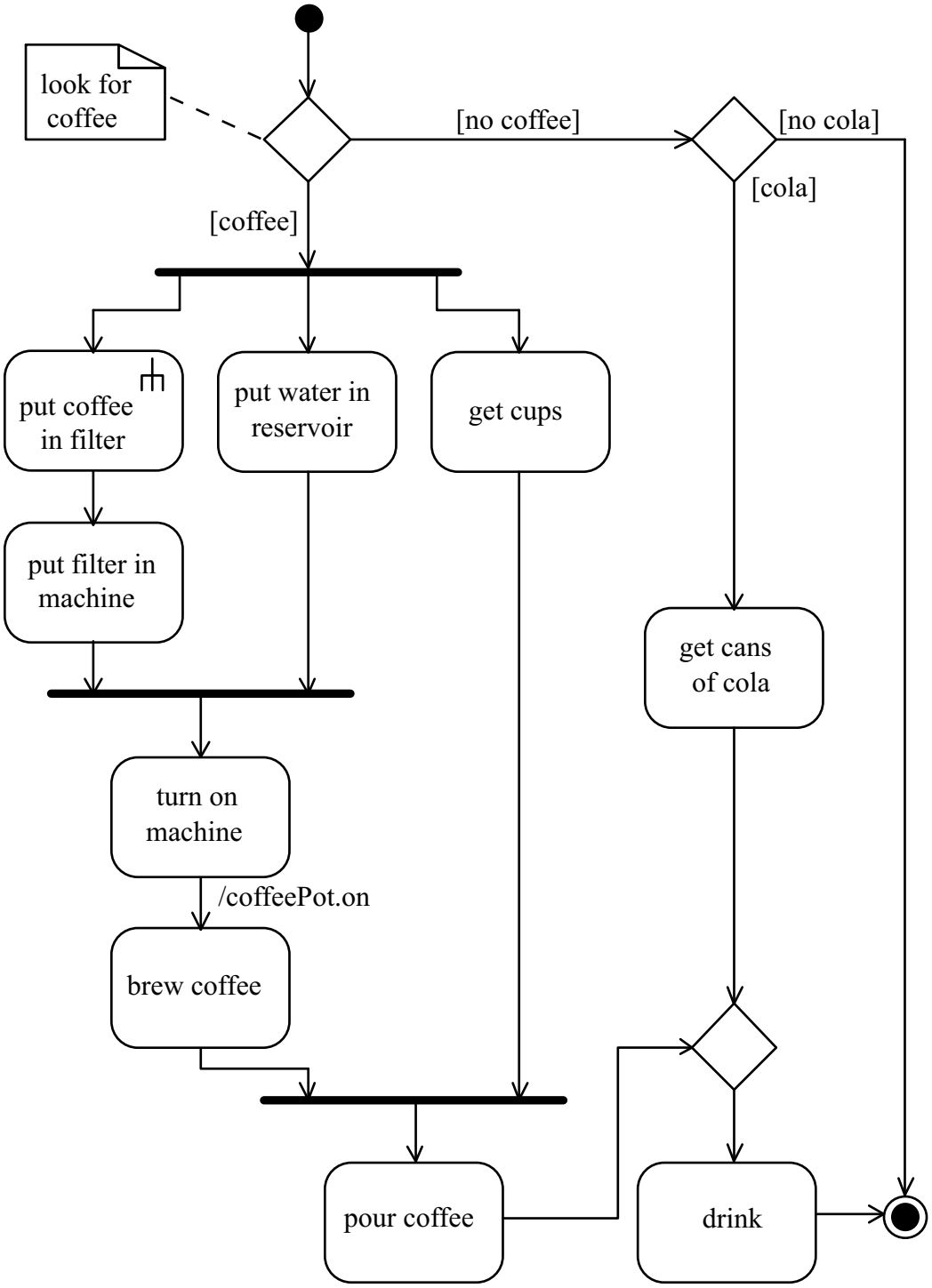
  - Three of you can work on it at the same time.

  - The only restriction (in the first parallel section) is that you'd better put the coffee in the filter before you put the filter in the machine.

- A *join* (multiple in, single out) means that all the incoming transitions must arrive before the outgoing transition may leave. It synchronizes the process.

  - Don't turn on the machine before putting both the coffee filter (with coffee) and the water in it.

- A join and fork can be combined and have multiple arrows arriving on one side, and multiple arrows leaving on the other. All the incoming transitions must arrive before any of the outgoing transitions may leave.

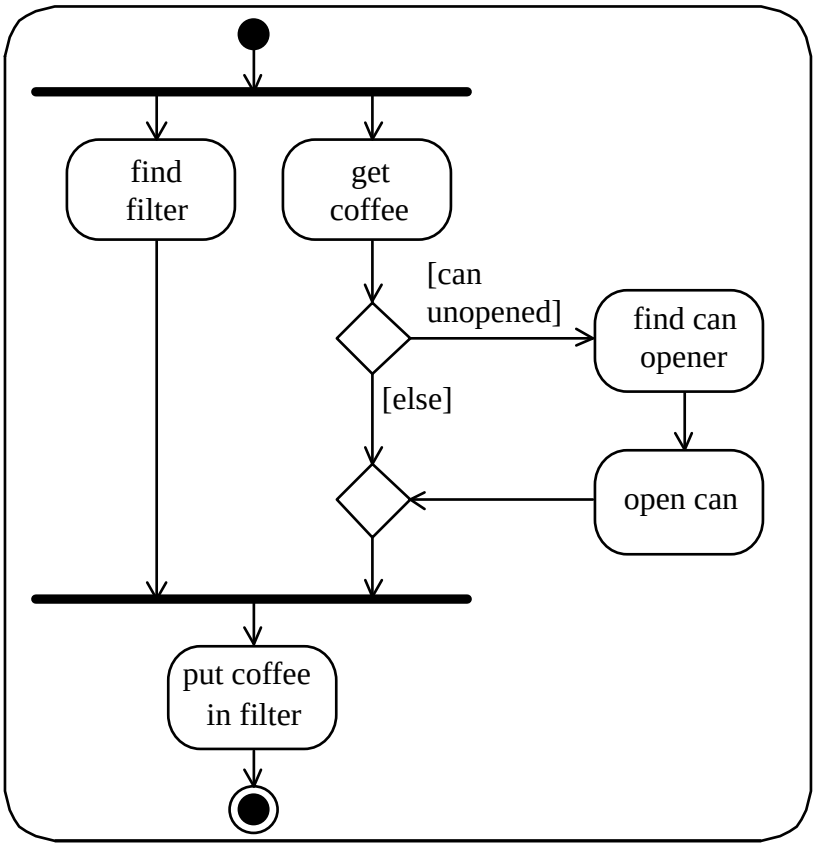


Note the action associated with one transition. This diagram was adapted from the UML standard, V1.4, sec. 3.84.3.

DRILLING DOWN

- The activity model on the previous page is on the complicated side of our magic number rule, but activity diagrams are usually easier to understand than, say, communication diagrams with the same number of nodes.

- If we want to add detail to a section of this model we would probably do it on another diagram, both to avoid making this one more complicated, and to avoid adding material at a different level of abstraction. This is called *drilling down*.



- We expanded the **put coffee in filter** node to create this new model with more detail and mark it with a rake.

  - We could expand any of these nodes if we needed to. Finding the can opener, for example, could turn into an adventure.

- We probably should make notes on both of these diagrams that tie them together. The **CASE** tools can help organize them for us.

Here the **else** is quite useful, as the alternative to an unopened can could be an open can, an open bag, a closed jar with a lid, or a heap of coffee lying out on the counter. The **else** covers all of these possibilities and anything we didn't think of.

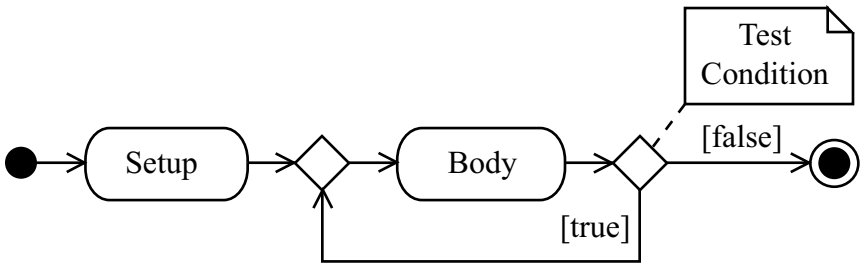
This model fits in place of the previous **put coffee in filter** node. The entry and exit of that node correspond to the start and endpoints of this model that expands and explains it.

A lot of business and software processes could be much more parallel than they are. Often this is because we couldn't represent the parallelism, and things that were represented sequentially became sequential. Now that we can represent it, we should be on the lookout for places to go parallel. Many user interfaces, for example, force the user to do things in a particular order, no matter that another might be more convenient. Activity diagrams are very good for modeling the back and forth of a user interface.

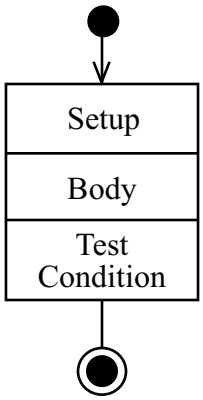
Investigate:
Why did we need the merge here? Couldn't we go down from **open can** to the join?

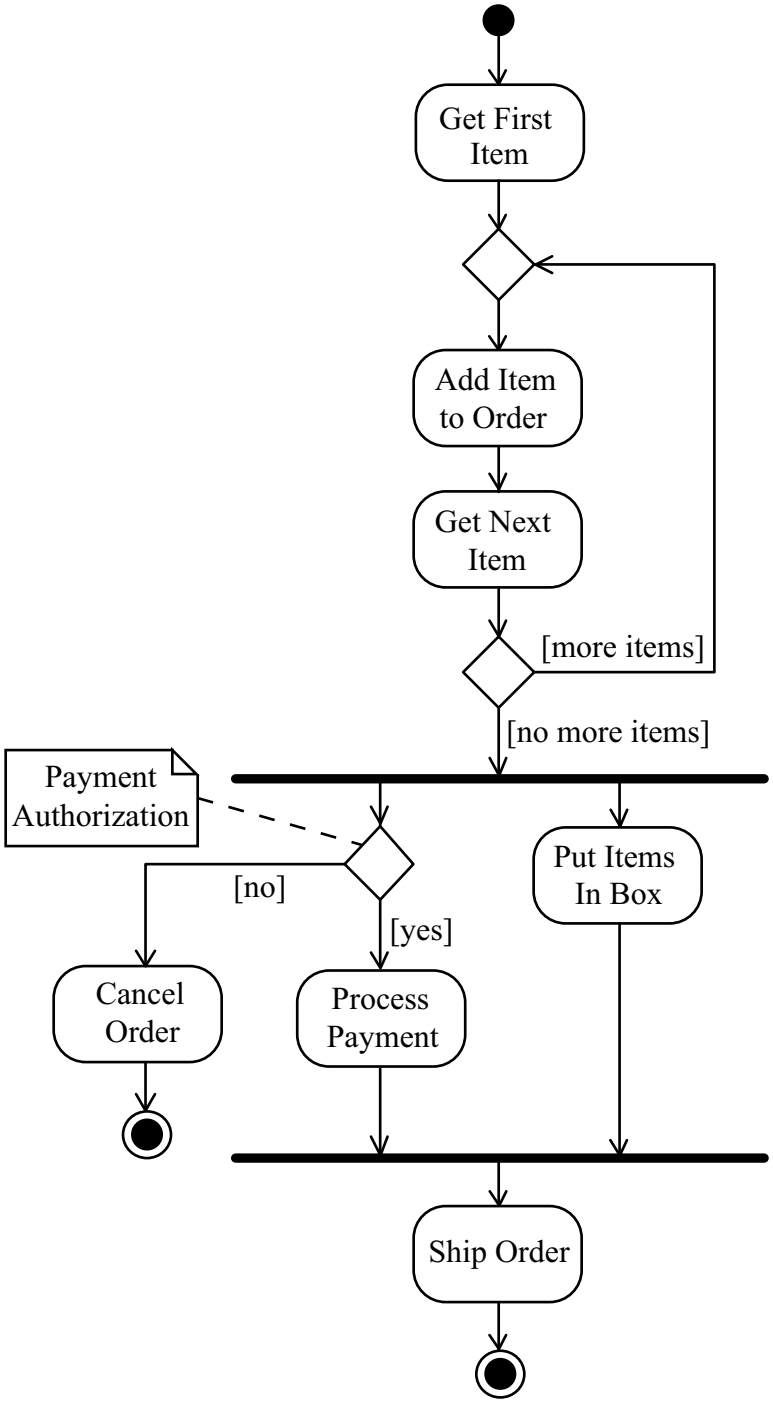
ITERATION

- We can model iteration on an activity diagram as we do on other dynamic models.
  - Note that the decisions and branches of control are clearer than they were in sequence and communication models.
- In the diagram on the facing page, the activity **add item to order** is done for each item.
- UML 2.0 provides a notation for modeling looping in activity diagrams.
  - A typical loop has three parts: the setup, the body, and the test.



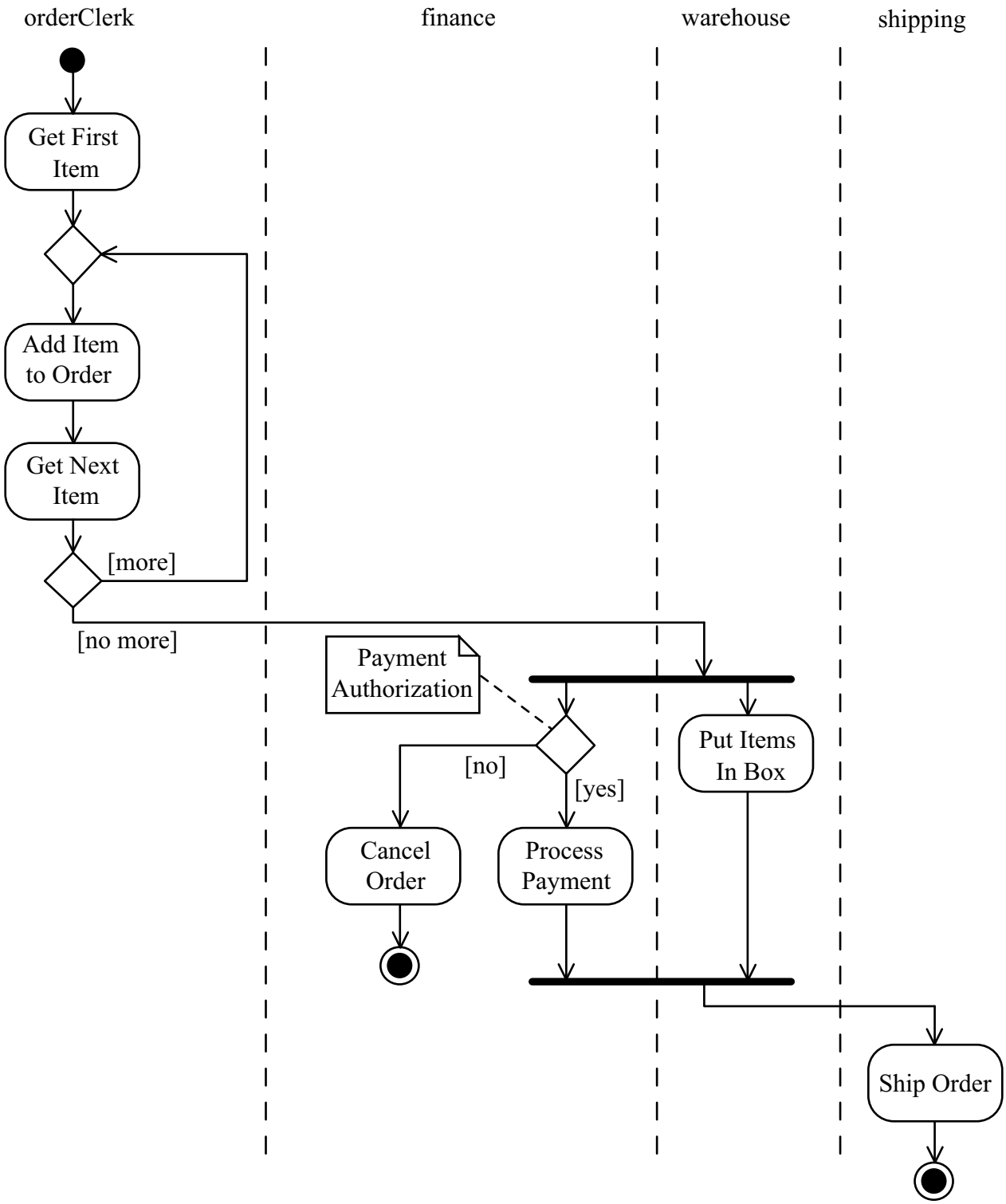
  - You can remodel the previous diagram using activity partitions in a single node.




Get First Item
Add Item to Order
Get Next Item
[more items]
[no more items]
Payment Authorization
[no]
[yes]
Cancel Order
Process Payment
Put Items In Box
Ship Order

PARTITIONS

- This activity model illustrates *partitions* (also known as *swimlanes*). The labeled vertical bands show who is responsible for each of the activities.
  - Use partitions when you need the activity model to show complicated flow of control and who does what.


orderClerk
finance
warehouse
shipping
Get First Item
Add Item to Order
Get Next Item
[more]
[no more]
Payment Authorization
[no]
[yes]
Cancel Order
Process Payment
Put Items In Box
Ship Order

SIGNALS

- A simple activity diagram has clearly-defined starting and ending points that determine that program's runtime. Some applications need to respond to signals generated externally from the application.

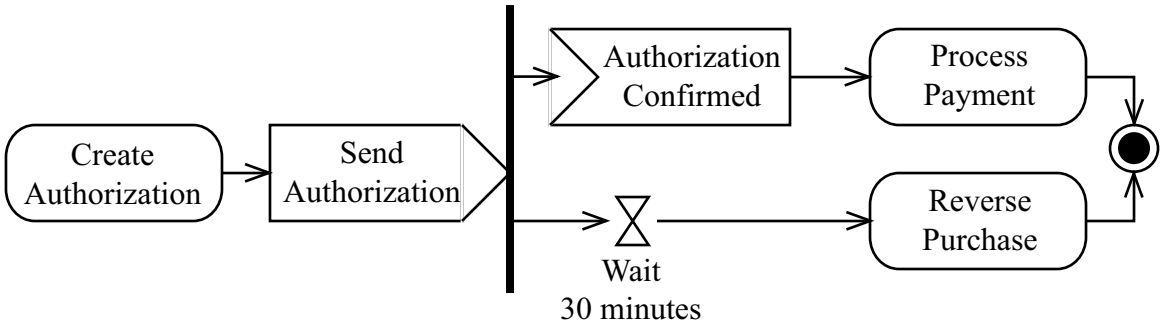
- UML 2.0 provides for three types of signals: time signals, input signals, and output signals.

  - *Time signals* specify an interval of time before a signal is produced.

  - *Input signals* come from an outside source or process.

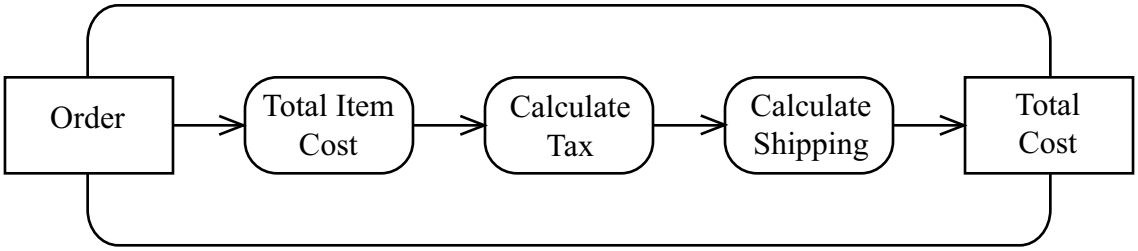
  - *Output signals* are signals sent from your system to some outside system.

- The following diagram depicts a system that sends a signal to authorize a payment and then waits 30 minutes for a reply. If a reply is not received, then the authorization will assume to be declined.



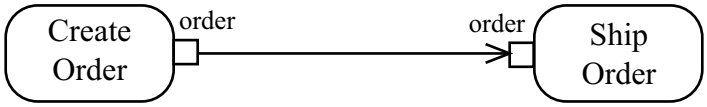
PARAMETERS AND PINS

- You can specify input into an activity and output from an activity using parameter nodes.
  - *Parameter nodes* are rectangular boxes on the edges of an activity.
  - The input parameter node must have an edge to the first action and the output parameter node must have an edge from the final action.

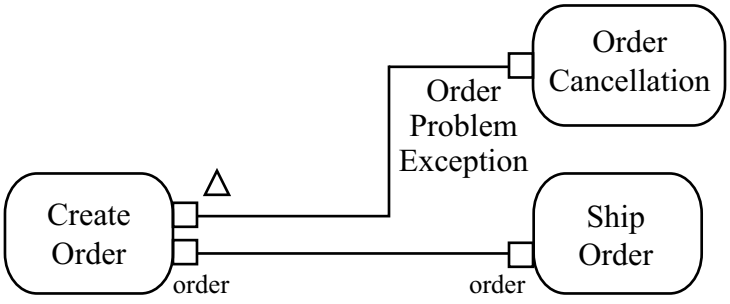


- *Pins* are a notation in activity diagrams to depict input and output parameters.

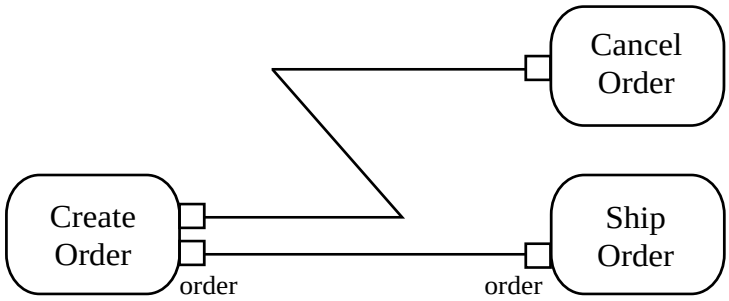
- In the following diagram, the activity **Ship Order** requires an **order** object from **Create Order**.



- If the output of an activity is an exception, then flag that pin with a small arrow near the pin.



UML 2.0 provides an alternate notation for exception handles using a "zig-zag" or "lightning bolt" instead of marking a pin with a triangle:



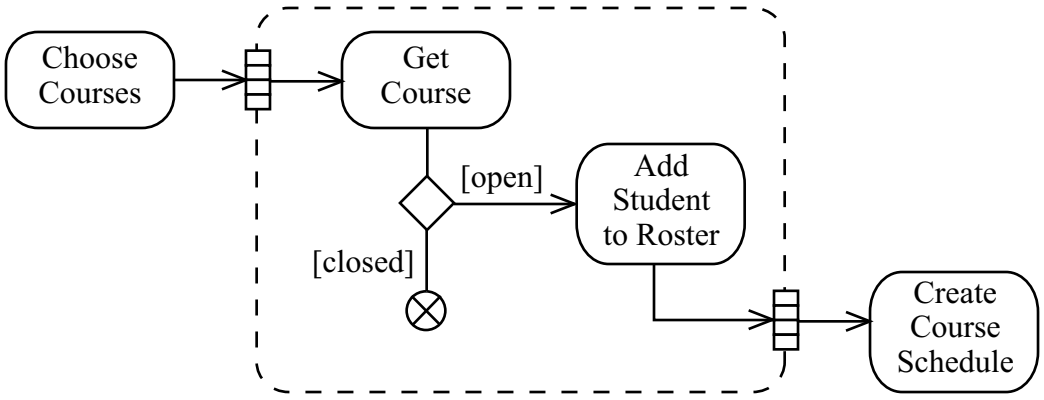
EXPANSION REGIONS

- You can use *expansion regions* to show that an action or series of actions occur on a collection of items.

- The following diagram depicts an activity that produces a collection of courses that need to be processed.

- Sometimes when processing a collection of items, one of the items may not be able to be processed. In the previous example, one of the courses chosen may have been closed. Just because some of the courses are closed does not mean the student should not be registered for the rest.

- *Flow finals* indicate that one of the items in the collection may terminate, but the rest should be processed.



USING ACTIVITY DIAGRAMS

- Activity diagrams are mostly about the flow of control. The activity model is particularly good at showing complicated branching and parallelism in control, but there isn't much information moving here.

- Activity diagrams are good for modeling the operation of an existing paper system that you intend to replace, if it is more about control than information.

  - If the information flow and manipulation is the main thing, try data flow diagrams.

- Activity models are useful for modeling user interfaces.

- Activity models are often used to illustrate branching in the operation of use cases.

- You can model complex algorithms you are going to use in the code for your methods.

- Activity models are at home in all three perspectives: domain, requirements, and design.

LABS

❶ Create an activity diagram for the user interface to a simple Automatic Teller Machine performing the withdrawal operation. Include as much parallelism as possible.

