Table of Contents

Purpose

The purpose of this OOAD/UML Instructor Best Practices document is to provide some guidance, suggestions, hints and helpful ideas that will enable an instructor to deliver an interesting and value-added course. It should be used in conjunction with the *Rational University Instruction Guide* and the instructor notes included with the OOAD/UML course material.

General Guidance

"The best teacher … is not the one who knows most, but the one who is most capable of reducing knowledge to that simple compound of the obvious and the wonderful." H.L. Mencken

From "Inside Technology Training" may 1998:
10 commandments for trainers:
http://www.ittrain.com/archive/May_98_26.html

1. thou shalt thoroughly prepare.
2. thou shalt check logistics.
3. thou shalt take responsibility.
4. thou shalt involve the learners.
5. thou shalt emphasize comprehension.
6. thou shalt apply material to the trainee's job.
7. thou shalt honor the time schedule.
8. thou shalt solicit feedback.
9. thou shalt view training as a process.
10. thou shalt have fun.

10 commandments for trainees:
http://www.ittrain.com/archive/May_98_27.html

1. thou shalt come prepared.
2. thou shalt meet with your managers.
3. thou shalt take responsibility.
4. thou shalt participate.
5. thou shalt learn from mistakes.
6. thou shalt seek to apply the training to your job needs.
7. thou shalt honor the time schedule.
8. thou shalt have a positive attitude.
9. thou shalt give feedback.
10. thou shalt complete the course evaluation.

- The instructor should emphasize his/her experience and include lessons learned, real life examples, "war stories", etc.

Typical Schedule

The schedule presented below is a recommendation. The course can be delivered in many different ways, and lengths. The length of the exercises may vary depending on how they are conducted (if you let the students present the results; how much you guide and give hints during the exercise, etc). You may also choose to let the students use tools (e.g., Rose) for the exercises. This does however require more time.

The schedule/timeline has been organized so that Book 1 is taught in the first day and a half, Book 2 for one day, and Book 3 for the last day and a half. It is important to note that the student is expected to be familiar with many of the concepts that are discussed in Use-Case Analysis; therefore the instructors should

NOT allow themselves to be bogged down in Book 1. Rational University now offers an Analysis and Design curriculum instead of a single course. This course assumes that the student has taken DEV 275 Essentials of Visual Modeling or else has equivalent experience.

Note: There is simply too much material for students to digest in a one week time period (especially the last two days). Even though you may cover all the material, continually reinforce the basics and key concepts. Repeat, repeat, repeat.

This schedule assumes that class starts at 8:30AM, ends at 5PM, with an hour for lunch, as well as frequent breaks throughout the day.

	Total	Module	Slides	Time	Notes
Day 1 Total	**8.00**				**There is buffer built in here for breaks and reviews which are unaccounted for.**
		About this Course	10	0.50	Expectation setting, Introductions, logistics, how to use materials. See 00about.ppt for instructor guidelines
		USP Best Practices	45	1.5	The Six Best Practices and RUP Overview have been condensed into one module now.
		Concepts of Object Orientation	50	1.25	Just define key concepts and UML notation. Much of this will be review for students who attended Principles of Object Technology.
AM Total	3.25				
		Requirements Overview	31	1.00	
		Exercise: Requirements Overview		0.50	
		Analysis and Design Overview	16	0.50	
		Arch Analysis	38	1.00	
		Exercise: Arch. Analysis		0.75	
		Use-Case Analysis (through p.35)	35	1.00	It is very important that you at least begin the lecture on this module on the first day. This will allow plenty of time for the exercise and allow you to begin design on Tuesday afternoon.
PM Total	4.75				
Day 2 Total	**7.75**				**There is buffer built in here for breaks and reviews which are unaccounted for.**
		Complete Use-Case Analysis	27	1.00	
		Exercise: Use-Case Analysis		3.00	A lot of time is allocated for this exercise so that the students can create their models and you will

					have plenty of time to review their work. Remember, that the students should already know how to create an analysis level use-case realization before attending the class.
AM Total	4.00				
		Identify Design Elements	54	2.00	
		Exercise: Identify Design Elements		1.00	This time includes time to review the students results.
		Identify Design Mechanisms	23	.75	There is no exercise for this module. It is possible that this module may not begin until Day 3.
PM Total	3.75				
Day 3 Total	**7.50**				
		Describe the Run-time Architecture	33	1.25	
		Exercise: Describe the Run-time Architecture		.75	This exercise should go pretty quickly since you are only drawing what is given.
		Describe Distribution	31	1.5	Make sure the students understand the distribution mechanism since they will need to model it in Use-Case Design.
		Exercise: Describe Distribution		.75	This exercise should go pretty quickly since you are only drawing what is given.
AM Total	3.25				
		Use-Case Design	40	1.5	This module now includes the distribution mechanism. Be sure to take your time so your students understand how it is implemented.
		Exercise: Use-Case Design		2.00	This exercise will take a long time since the students are now incorporating subsystem interfaces and the distribution mechanism. Review time is built into the exercise time.
		Subsystem Design	29	.75	The RDBMS persistence mechanism is discussed in this module.
PM Total	4.25				
Day 4 Total	**8.00**				

		Exercise: Subsystem Design		1.25	Exercise may run longer if the students are still having some trouble with identifying classes and allocating responsibilities. That's OK as those concepts are the key things you want your students to learn (and practice). Just don't let the students get hung up on the nitty-gritty details of a particular subsystem.
		Class Design: through Define Attributes	40	1.25	
		Exercise: Define Attributes/Operations/Statecharts		1.0	
AM Total	3.50				
		Class Design: Define Dependencies to end of module	62	2.00	
		Exercise: Define Relationships		1.75	
		Database Design	23	.75	There is no exercise for this module.
PM Total	4.50				
Overall Total	**31.25**			**31.25**	

Instructor Preparation Tips

New Instructor

The following is a recommended strategy for someone that has never taught the OOAD or the Rose course before that would like to prepare to teach OOADv2000 (some are general, and others are specific to the courses):

- Review the data sheet for the OOAD and Rose courses
- Review the OOAD/Rose sales guide

OOAD/UML

- Review the existing course slides, student notes, and instructor notes
- Work all of the exercises.
- Review the instructor best practices document (this document)
- Review the following chapters from the Rational Unified Process
    - Graphs: Overview
    - Manuals: Introduction
    - Graphs: Workflow in Analysis and Design
    - Process: Analysis and Design
    - Artifacts: Analysis and Design
    - Modeling Guidelines: Modeling Guidelines for Analysis and Design
    - Process: Iteration Workflows, concentrating on analysis and design

activities
- Read "Unified Modeling Language User's Guide" by Grady Booch
- Take a look at the OOAD Instructor web site to become familiar with the information that is available (this is not currently available to non-Rational employees)

Rose

- Install Rose
- Review the existing course slides, student notes, and instructor notes.
- Work all of the exercises.
- Walk through the online UML Tutorial (installed with Rose)
- Review the Rational Unified Process Rose Tool Mentors
- Browse the RBU Beginner's Guide to Rational Rose web site (this is not currently available to non-Rational employees). Start at http://midnight.rational.com/rbu/products/beginner.htm. Here you can view presentations that talk about Rose and Visual Modeling in general, order the book Visual Modeling with Rational Rose and UML (part number 4100-09007), see a list of other recommended books, and order the Inside the UML CD (reference number G-062).
- Browse the Rose web site (this is not currently available to non-Rational employees). Start at http://midnight.rational.com/rbu/products/rose98i.htm . Here you will find many interesting topics from an introduction to rollout activities. Pay careful attention to the FAQ section—there are many questions already answered for you at this site. Become familiar with the information that is available (it's impossible to read it all ;-))

OOADv3.6 Instructor

The following is a recommended strategy for someone that has taught the OOADv3.6 course before that would like to prepare to teach OOADv2000:
- Review the OOADv2000 data sheet.
- Review the OOAD/Rose sales guide (provided).

The above are important as the scope and focus of the course has changed. Consistent with suites and the new Rational tag line to make teams successful, the OOAD course is the course for the designer, NOT the analyst (suite role definitions). As a result, use cases are NOT part of the OOAD course (they are in RMUC). The scope of the course is the Rational Unified Process analysis and design workflow.

- Review the instructor best practices document (this document)

This is important as it provides hints and tips on what to concentrate on and how to guide the students through the large amount of information now contained within the OOAD course.

The following are the technical concepts that are new or whose emphasis has changed in OOADv2000, as compared to OOADv3.6. They are listed in order of importance (most important first). If you are an experienced OOADv3.6 instructor, just becoming familiar with these concepts and how they are presented and used in OOADv2000 should provide you with a good base for teaching OOADv2000.
1. Analysis and Design Workflow (serves as the framework for the course)
2. Use-Case Realizations
3. Interfaces and subsystems
4. Subsystems vs. packages
5. USP Unit (Best Practices of Software Engineering and Introduction to the Rational Unified Process; see the **Delivery of the Unified Software Best Practices Modules** section in this document for more information)
6. Modeling processes and the mapping of design elements to processes
7. Mechanisms (Identify Design Mechanisms) and their realization (use-case design, subsystem design): Security, Distribution, Persistence

If you are short on time, you can concentrate on the above areas, but the ideal situation is for you to do everything described in the preparation for a new instructor, moving quickly through those items you are familiar with.

OOADv4.2 Instructor

The following is a recommended strategy for someone that has taught the OOADv4.2 course before that would like to prepare to teach OOADv2000:

- Review the OOADv2000 data sheet
- Review the OOAD/Rose sales guide
- Review the OOADv2000 release notes and review those sections of the course that have changed, especially the new architectural mechanisms
- Review the course materials and how to use them (see the **How to Use the Course Materials** section of this document). The supporting documentation has completely changed from what was provided in v4.2.

How to Use the Course Materials

Instructor Manual, Books 1,2, and 3

Review these manuals very carefully. They contain both student notes, as well as instructor notes. All the technical information you need to know to teach the base course is in these manuals.

Instructor Best Practices Document

That is this document. It contains a lot of valuable information regarding the delivery of the OOAD and Rose courses. When mistakes are discovered in the course, or additional hints and techniques are defined between releases of the course, they will be defined in this document. You can think of this IBP document as a holding tank for additions to the course. Updates to this document will be mailed to each certified instructor. As part of each course release, any applicable notes will be included in the main course and removed from this document.

Course Registration Requirements Document

This document contains the requirements that will drive the development of the Course Registration System course example (e.g., the Use-Case Model main diagram, Problem Statement, Glossary, and Supplemental Specification).

Payroll Requirements Document

This document contains the requirements that will drive the development of the Payroll System course exercise (e.g., the Use-Case Model main diagram, Problem Statement, Glossary, and Supplemental Specification).

Payroll Architecture Handbook

This document contains all of the architectural information that is “given” to the students to complete the exercises. This includes textual descriptions of the architecture, as well as documentation for the architectural mechanisms.

Payroll Solution Appendix

This appendix contains hard copy of the solutions for each of the exercises. There is chapter per module, each with it’s own table of contents.

Course Registration Models and Payroll Solutions Models CD

This CD contains the soft-copy of the Rose models for the Course Registration System course example and the Payroll System course exercise. It is included in a pocket inside the back cover of the **Payroll Solution Appendix**.

The soft-copy of the Payroll System models are "incremental". There is a model file for each exercise that reflects what the model would look like AFTER the exercise was completed. This is helpful if OOAD is taught "hands-on" with Rose. See the **Intermingling the OOAD and Rose Courses** section for more information.

Incremental models were not provided for the Course Registration System as the ROI for such an effort was not warranted (the Course Registration System is the course example is not developed by the students).

Additional Information Appendix

This appendix contains a set of modules that may or may not be included in the course delivery, depending on the students' experience level, background, and interest, as well as the instructor's knowledge and comfort level.

There are basically three types of additional information modules:

- Architectural Mechanism Modules
- UML-to-Language Map Modules
- Patterns Supplement

Each of these is described in the following sections.

Note: The course schedule/timeline that is provided in this document DOES NOT include the incorporation of these appendices. Thus, if the instructor chooses to include modules from the Additional Information Appendix, the timeline will need to be adjusted accordingly.

Architectural Mechanism Modules

The architectural mechanism modules contain the details on the architectural mechanisms referenced in the base course. There is one module per architectural mechanism. Each module has two parts. These parts are separated by colored sheets of paper and a title page (they do not have separate tabs). The first part contains those slides that should be included when discussing the Identify Design Mechanisms module of the base course. The second part contains those slides that should be included when discussing the Use-Case Design module of the base course. The base course includes references to these sections from the appropriate slides.

Note: The Course Registration and Payroll Rose models incorporate all of the architectural mechanisms. However, there is a separate use-case realization for each mechanism for each use case, so it should be straightforward for an instructor to selectively cover specific architectural mechanisms

UML-to-Language Map Modules

These modules contain a mapping from the UML to specific implementation language constructs. These language-specific examples may be used in the course to articulate certain concepts or as a reference for the student after the course. The instructor may include them in the base course discussion or just point the students to them for later reference. Instructors may use this appendix to connect with students that will not believe anything the instructor says until the instructor "shows them the code".

Patterns Supplement

This module contains an overview of the following Design Patterns:

- Command
- Proxy

- Singleton
- Factory
- Observer
- Mediator

If students in the course mention an interest in learning more about patterns, this section provides further information and can aid in the understanding of the subject before delving into the GoF's book (or some other resource).

Delivery of the Unified Software Best Practices Modules

The first course module (following the general course information module), the Six Best Practices of Software Engineering, are called the USP Unit. They are included from the one-day Unified Software Best Practices course. They will be included in each professional education course (i.e., non-tool-focused course) offered by Rational University.

The goal of these modules is to set the context for the course and provide an overall consistent message for all professional education courses. For this reason, it is possible that some of the students may have seen the material before. Thus, it is important that the delivery of these modules be tailored to reflect the focus of this course, analysis and design. Provide forward references into the OOAD course topics and modules, as appropriate.

I have included a few ideas on how to deliver these modules in this document (see the **Frequently Asked Questions / Points to Emphasize** section for the Six Best Practices of Software Engineering module for specific information on what to emphasize). The bottom line is to make the modules interesting, and eliminate the potential for "death by slides". These modules do not contain any exercises, so it's up to the instructor to make it interesting. Remember, tie everything back to the topic the students have came here to see – analysis and design! You can position this with the students by saying that they "have the full set of slides for their own interest, but that this course concentrates on solving these particular problems with these best practices and this workflow".

See the **Typical Schedule** section for the amount of time to be spent on these modules.

Note: These modules do add value; however, instructor's have the option to discuss this type of material later in the class when the class is ready to stop thinking tactics and step back and look at some of the foundational technology. The instructor may delay the delivery of these modules until later in the course. Some instructors have found that this material can cause the class to get kicked-off slow (e.g., on the first day, people are ready, attentive, eager - they want to get started and learn. We in turn, spend 3-4 hours with overview material). To address this concern, you can jump in to some real, meaty material they can get their arms around and do some exercises with as soon as possible - because that's what they are there for and they're eager for it.
It is sometimes inappropriate to assume that everyone is interested in the background and theory stuff immediately. They may not be. The instructor often has to resolve their immediate need (learn some hands on technology) first then later step back and look at the overall picture of what we're doing.

Use of the Architectural Mechanisms

The overall goal is to provide a course that can be taught to an introductory audience "right out of the box". The inclusion of the mechanisms in the course can make this difficult, however, some mechanism discussion is valuable. Thus, the course includes the following "solution":

In general, general information on the mechanisms is discussed in the architectural modules, specifically the Identify Design Mechanisms and the Describe Distribution module, and the incorporation of the mechanisms is described in the detailed design modules (Use-Case Design and Subsystem Design). Thus, there are two sets of mechanism slides in the appendix, one set to be discussed during the architectural mechanisms section, and one set to be discussed during the detailed design modules. These "slide sets" are separated distinguished in the appendix with a title slide for each (but they both appear under the same mechanism tab).

Note: As discussed later in the **Exercise Solution** section, the Payroll exercise solution contains separate use-case realizations that demonstrate the incorporation of the different architectural mechanisms. There is one use-case realization for each use case for each mechanism, plus a use-case realization that incorporates all the mechanisms. This was done to make it easier for instructors to teach some of the mechanisms.

Frequently Asked Questions / Points to Emphasize

"In my experience, the single question most often asked … "Do you write with a pen, a typewriter, or what?" I suspect the question is more important than it seems on the surface. It brings up…" "On Becoming a Novelist", John Gardner

The OOAD course contains a significant amount of information, especially for students new to OO. The students cannot be expected to learn it all. It is imperative that the instructor stays focused on the key points, and de-emphasizes the nitty-gritty details. This document is meant to assist the instructor in determining what those key points should be.

The instructor manual contains frequently asked questions and points to emphasize in the Instructor Notes. The following are those that are more general in nature, or those that have not made it into the Instructor Notes yet. Be sure to review the instructor notes for each section for more information.

This section describes the key points of each module. In addition to reviewing this section, be sure to review the **Course Flow Options** section if you feel that you need to tailor the course delivery based on your audience's skill level. For hints and techniques for the exercises of each module, see the **Course Exercises** section**.**

General

Use of the checkpoint slides: The majority of the modules contain "checkpoint" slides – slides that contain some checkpoints that designers can use to assess their progress. There is no need for the instructor to read every checkpoint. Just pick a couple and elaborate on those. It's important that the student knows that the checklists exist for later reference during the exercises.

Another way to discuss the checkpoints is to ask the students why some of them are good checkpoints (e.g., why do we want each class to do one thing and do it well?).

About this Course

Frequently Asked Questions

- How much will we cover each day? Sometimes a student has to miss a few hours for a meeting or appointment so they would like a rough idea of what they will miss.
  You can white board the timeline included in this document, but emphasize that every class is different and you can't guarantee exactly where in the course you will be.
  Revise estimates at the end of each day. If you give a firm schedule, students get anxious if they are "behind" where they "should" be or bored if they are too far ahead of where you told them they "should" be.

Points to Emphasize

- The course objectives: the development of a robust design model in the context of a use-case-driven, architecture-centric, iterative process.
- The course is NOT an architecture or an implementation course, it is an analysis and design course (for the Designer).
- How to use the course materials.

Six Best Practices of Software Engineering

Frequently Asked Questions

- How much time do you spend in each phase?
- What do I do First, Next?
- Spiral - How do I know when I'm done?
- When are you done with analysis? When are you done with design?
- When are requirements done?
- That's not what are process is, or We can't convince our management to do this, can we still do OO?
- My company doesn't do an iterative development process or software architecture like you describe, can we still get benefits from OO?
- What do you mean by architecture, two-tier?
- What is a conceptual prototype?

Points to Emphasize

The following best practices are those that you should concentrate on as they relate directly to the OOAD/UML course, listed in priority order (highest to lowest):

1. Practice 4: Visually Model Software
2. Practice 1: Develop software iteratively
3. Practice 3: Use Component-Based Architectures (though, keep in mind, OOAD is NOT an architecture course)

Present the others briefly, so the students are aware of their existence.

Visually Model Software

- Consider leading a discussion on why we visually model.
- Emphasize using the UML as the language of expression – it manages complexity and facilitates communication.
- Emphasize the hump back chart

Develop Software Iteratively

- Emphasize reduced risk. Software developers and managers must consider business risks as well as technical risks. Give the students examples of each kind of risk.
- Project managers typically have concerns applying an iterative process to their software development projects. The two most common concerns are: 1) "When am I done?" and 2) "How do I know iteration 6 won't break all the iterations developed before it?"
- Discuss with the class how they can use use cases to help measure progress and determine when they are done.
- Discuss with the class how architecture helps reduce the risk of future iterations breaking previous iterations.
- Emphasize that these are legitimate risks and they must use good engineering judgement to mitigate them.

- Also, the fact that testing needs to occur throughout the lifecycle makes some managers uncomfortable. Some are not used to worrying about testing, including allocating resources for testing, until later in the lifecycle.

Use Component-Based Architectures

- Possibly lead a discussion on what is architecture.

Be careful with this module if you are presenting to an audience that is not Rational Unified Process (RUP)-friendly or they have their own process. In such a case, just use this section to justify the organization of the OOAD course NOT to push RUP. RUP is just the process framework that we use to structure the course.

Emphasize that knowing the UML is not enough – you need a process. The UML provides/defines many different types of diagrams, but really does not provide any guidance on where in a process they should be used. Just learning the UML to learn object-oriented analysis and design would be like learning the English dictionary to learn the English language. That is why the OOAD course is organized around the Rational Unified Process (though the concepts can be applied in ANY process framework).

The most important topics to cover in this module are the following, in priority order (highest to lowest):

1. *Use-Case Driven*. Use cases drive the development of all of the other models. Use cases are covered in a little more detail in the Requirements Overview module.
2. *Architecture-Centric*. Early development and validation of the architecture reduces overall project risk. The architecture is the primary artifact for conceptualizing, developing, and managing the system under development.
3. Lifecycle phases and iterations (e.g., the "hump chart")

For more technical background on the rational Unified Process, refer to "The Rational Unified Process: An Introduction", by Philippe Kruchten (Addison-Wesley, 1999).

Points NOT to Emphasize

Do not spend any time on the UML diagrams in this section, as these diagrams will be presented, in detail, throughout the OOAD course. This section is really for the other professional education courses where UML is not such a prevalent topic.

Don't spend too much time on a detailed look at the 4+1 views of software architecture. These will be covered in detail in the OOAD course. Just familiarize the students with Rational's concept of there being different architectural perspectives.

Do not spend any time discussing the workflows, especially the analysis and design workflow, as the analysis and design workflow is the focus of OOAD/UML (e.g., it will be covered in detail in the core OOAD/UML course). Simply provide a high-level description of the purpose of each workflow. Do not discuss each step (unless the topic is of interest to the majority of the students). It is important to let the students know that the RUP provides information on all these areas. It is not important that they be covered in detail (that's what the other Rational courses are for ;-)). Use the student's interests, background, and expectations that you "gleamed" from the introductions to decide how much time to spend on the other RUP workflows. For example, if you have a room full of analysts, you may want to spend a little more time on the Requirements workflow and very little time on the Implementation and Environment workflows.

Concepts of Object Orientation

Points to Emphasize

- Discussion of basic OO principles
- Introduction of basic UML terms and notation

Points NOT to Emphasize

- The UML details, as these will be addressed in later course modules.

Requirements Overview

Frequently Asked Questions

Why is Login a separate use case?

Login has been documented as a separate use case for the following reasons:

- The Login use case is a complete and meaningful flow of events – the user can log in, then log out with nothing in between, and it is complete and meaningful. The observable result of value is that the user is authenticated to the system and can now perform other duties - in other words, you have satisfied a pre-condition for the other use cases. In fact, some use-case pre-conditions will say: "User has authenticated at Administrator level", to indicate that this is not just an everyday thing that everyone should be able to do.
- Authentication and authorization are issues separate from the rest of the "interesting" system behavior.
- Authentication and authorization are preconditions for all other system functionality.
  Since a user can perform multiple use cases once logged in, the prerequisite approach seems to be the clearest representation of reality.
  Just about everything requires authentication - this would make the use-case diagrams unnecessarily cluttered as you <<include>> Login in every use case. You could use the Login use case as an example of how preconditions can be used very effectively to simplify a use-case's flow of events and why that is a good thing.
- Login has the potential of becoming complex if a security mechanism is applied. Keeping it separate reduces the duplication of information and the resulting maintenance effort

Another option that was discussed as a possibility was to have every use case <<include>> the Login use case. However, because just about everything requires authentication, such an approach would make the use-case diagrams unnecessarily cluttered with an <<include>> Login in every use case. We also did not really want to introduce/review use case includes and extends in the OOAD course, as they can be considered advanced use-case modeling techniques. For more information on why we chose not to include use-case includes and extends, see **Why don't we cover use-case includes and extends** section below.

The bottom line is that whether or not you model Login as a separate use case depends on your domain and your user. For this domain, we have chosen to model it separately (for the reasons given above).

While a dialog about why Login is a separate use case may be quite interesting, be careful not to spend too much time here, as the focus in the OOAD course is on developing object models, not use-case models.

Why don't we cover use-case includes and extends?

The old uses and extends have, in UML 1.3 as well as in the RUP, been redefined into the new use-case relationships: include, extend, and use-case generalization. Since not only the relationships, but also the change in itself, probably will require clarification in class, we have left them out of the OOAD course to avoid confusion. However, you may want to look at the new specifications of these relationships (in the RUP or in Jim Rumbaugh's UML specification book). The world is not quite as simple as uses=include, uses=use-case generalization, and extends=extend. The specification of the relationships have become a lot clearer, and leaves less room to one's own interpretation (which has been a problem in the past). There won't be time in the OOAD class to explain them thoroughly anyway. That's what the RMUC course is for ;-)

While a discussion on use-cases includes and extends may be quite interesting, be careful not to spend too much time here, as the focus in the OOAD course is on developing object models, not use-case models.

Why is Printer an actor?

If anyone questions the printer actor, ask what he or she thinks, and then discuss how there are no "wrong" designs. Some arguments for and against including the printer as an actor are provided below.

Arguments FOR the printer actor:

- The ultimate destination/recipient, as far as the payroll system is concerned, is to the printer. If some picks those paychecks off the printer and hands them to an employee (or puts them in his/her box), or mails the checks to the employee, is not within the scope of this system. Such business rules are better represented in the business processes in which this payroll system runs (i.e,, in the business model). The capturing of the problem just seems incomplete without the printer actor
- Position the printer as an external system and not simply a device. It is very rare that you ever print directly to a printer these days, and you will almost always be constrained by some API, therefore the Printer actor is a necessary part of the use-case model (it is an external system that provides an interface). This also helps in the estimation process. What if the company decided to outsource the paycheck printing to a security firm or bank? Then the Printer would still be an external system that provided an API, and would definitely need to be an actor.
  Not all devices are actors. For example, the mouse and keyboard are not actors but **in this particular instance**, the printer is an actor. I also think that sensors may be actors at times too, especially if they initiate use cases. I guess what it boils down to is whether the device is providing significant external constraints on your system - a mouse or keyboard almost never does (unless you are writing device drivers), but printing is still a classic 'hard problem' despite driver libraries available.
- "What extra work does a developer have to do (say in a Windows environment) to handle the mouse or keyboard? When we want to print something, we have to decide on the layout etc, and the algorithms for doing this can be reasonably complex (another example: if you're printing checks on a laser printer, you will have to use MICR or E-13B fonts at the bottom of the check to print the ABA number, checking account number, etc that can be read by magnetic character strip readers. This is something that most printers don't do automatically by themselves... you have to tell them to do it). Such formatting and layout activities should be handled by a boundary class. Also, at requirements specification time, it is helpful to represent that hard copies are required."
- You reduce risk of misunderstanding by adding detail. The cost is the added time to build and maintain the model, and the slight increment to the complexity of the model.

Arguments AGAINST the printer actor:

- Things like Printers are a means for the system to produce some result to someone; show that 'someone' as the actor. If you know to whom you are presenting the 'result', you can better understand what to include in it. What means to use for producing the result is a design decision (although it might quite possibly be listed as a requirement). It could be a fax, a monitor, audio, ... As a comparison, you don't model the keyboard or the mouse as an actor, we model the Clerk, or the Customer (using the keyboard and the mouse) as an actor. Use cases and actors should show what the system does, not just the software

As the students will see by the discussion, people will model it differently. It is more a matter of modeling taste. As long as the definition of the boundary of the system is clear, that's what counts. One of the more valuable aspects of Use-Case modeling is the proper identification of the system boundary. What needs to be developed and what needs to be accommodated? This can and should be answered in the definition of the actors.

Ultimately, it is that the development team and the customer define the level of detail and completeness needed in a model. It has been said that models can never be made "complete", so we shouldn't lose too much sleep over it. Hit the high points and move on. The discussion of whether or not the printer is an actor should be kept short. In a class focused on use cases, then I would talk about the issue because it is important in that context.

Points to Emphasize

How to read the requirements artifacts that drive analysis and design.

Points NOT to Emphasize

How to develop the requirements artifacts (that's the focus of the Requirements Management with Use Cases course -- RMUC).

Analysis and Design Overview

Frequently Asked Questions

Should you maintain a separate analysis and design model?

A separate analysis model is not always a good thing, nor is it even needed. This may be because it's almost impossible to create a clear distinction between what should be in the analysis model versus what should be in the design model. People end up doing analysis using the design model and design using the analysis model, and it's impossible to keep them from doing so. What's more, it's impossible to separate analysis and design so completely that they can be segregated into two different models. The term analysis model also means vastly different things to different people - some use it in the same way as a 'domain model', and some as an 'ideal design' from which all "implementation details" have been expunged, and so forth. Since we cannot really tell people how they should use it and why, you may want to avoid telling people to use it at all.

Why is the workflow used in this course different that the Analysis and Design core workflow presented in RUP?

The workflow that we use in the course is a tailored version of the Analysis and Design core workflow of the Rational Unified Process. This course will be presented using the above workflow as a framework. This is to aid in presentation, and does not hinder the concepts from being applicable in other process contexts. It can be difficult to present an iterative process in a sequential order, and the amount of information in the analysis and design workflow is more than can reasonably be covered in four days. Thus, we decided to present a tailored and scoped version of the workflow that hits the high points and provides context and flow for the analysis and design activities.

Points to Emphasize

OOAD Course Scope: Be sure to stress what they will learn HOW to do in the course (the Designer activities), what they will be TOLD ABOUT, but not expected to derive (the Architect activities), and what will NOT BE COVERED AT ALL (the Database Designer activities). It is here that you set expectations. If you are planning on skipping or re-ordering any modules, you may want to mention that here when you are walking through the analysis and design workflow.

Points not to Emphasize

Separate Analysis Model: Maintaining a separate analysis model is definitely an option a project has. The Analysis Model is a sketch of ***some*** of the important concepts of the design. The investment in its development needs to be bounded by the value you expect to receive from it, and the risks it helps to mitigate. Maintaining a separate Analysis Model is a resource-intensive process that requires additional steps. Such steps are not addressed within this course.

Analysis and Design Activity Details: Do not explain each of the activities of the analysis and design workflow. Just hit the high points of each one. Tell them what you would expect them to remember about each activity at the end of the week. I would use the key concepts listed in the **OOAD/UML Course Roadmap** section as a starting point. You can then refer back to these descriptions when setting the context for each module (each module starts with the analysis and design workflow picture with a star on the current activity).

Architectural Analysis

Points to Emphasize

- *Identification of upper-level architectural layers*: Provides an initial structure in which to work. May be driven by a chosen architectural framework or analysis pattern.
  Introductory students don't need to know how to define layers, just what they are, how they are modeled, and how they affect the rest of the design process.
- *Identification of key abstractions*: Provides a jump-start to Use-Case Analysis, so that every use-case team does not spend time defining the same key abstractions. This will reduce the amount of homogenization that must occur when the individual use-case teams compare and consolidate their Use-Case Analysis results.
- *Analysis mechanisms*: Be sure the students understand the concept of mechanisms or else they will have a very difficult time in the design portions of the course. At this point, all we're identifying are shorthand key words for non-functional requirements (e.g., persistency, security, etc.).

If the system being built is a completely new system for which no one has any experience, Architectural Analysis can be skipped since you really don't know enough to identify anything.

Points NOT to Emphasize

Detailed class modeling: It is enough to just identify and define the key abstractions at this point.

Architectural patterns and frameworks: Just talk about patterns and frameworks briefly as these may drive the identification of the layers.

Use-Case Analysis

Points to Emphasize

Use-Case Realizations (analysis classes, allocation of responsibilities): The focus in Use-Case Analysis is the development of the analysis use-case realizations. This module has been divided into two parts, each with it's own focus and it's own exercise:

Part 1 focuses on the analysis class interactions -- the identification of the analysis classes and the allocation of use-case responsibilities to the analysis classes. You are looking across the entire use case to make sure you haven't missed anything. Be sure to walk through the interaction diagrams, emphasizing the responsibility allocation. Part 2 focuses on the analysis classes themselves – making sure that the

necessary relationships are defined to support the collaborations that implement the use case, and to make sure that no one class does too much or too little. In Part 1, you develop the interaction diagrams of the use-case realization. In Part 2, you develop the class diagram for the use-case realization (the VOPC), using the interaction diagrams derived in Part 1 to drive the relationship definition.
If you consider use cases and scenarios to be black box descriptions of the system, then the use-case realizations are the associated white box descriptions.

Attributes: If attributes are defined during Use-Case Analysis it is because there is a concept in the use case that must be captured, but does not warrant a full-blown analysis class. Don't spend any time filling out the attribute signatures.

Responsibility naming conventions: The use of "//" in the analysis operation name is a convention, not a requirement. It signifies a responsibility that may or may not be refined into one or more actual operations in design.

Relationship Identification: In Part 2, emphasize that for every link on the use-case realization collaboration diagram, you need a relationship on the VOPC.

Another way to focus the students on finding classes is to go to the whiteboard and list potential classes, using the stereotypes to guide you. For the entity classes, you can demonstrate a filtering exercise in the following way:

- Project one of the use-case flows of events onto a whiteboard, and then go through it word for word underlining the nouns on the whiteboard - and have the students do the same in their books.
- Together, go through and "filter" the nouns and discussed the why's and why not's, as well as documenting the reasons for "filtering in/out".
- Produce an interaction diagram on the whiteboard using just the objects/classes identified, demonstrating the allocation of responsibility.

If you have time, or feel that the students aren't "quite there yet", do another flow of events.
If there isn't a whiteboard to project on, one can project onto the screen and just make sure the students underline in their books, and then having an already "underlined" copy of the Word document to project as the result of the underlining activity to use for the filtering.
Rose can be used to draw the simple interaction diagram if one doesn't have a large enough whiteboard. This approach may "connect" the students with the problem at hand, maybe more so than displaying slides.

Identify Design Elements and Identify Design Mechanisms (used to be Architectural Design)

Points to Emphasize

Subsystems and Interfaces: Subsystems are the logical representation of components – completely replaceable units that satisfy/conform to some interface. Subsystems originate from "superman" analysis classes (classes with more stuff to do than can be implemented by a single class). In Architectural Design, analysis elements (analysis classes) morph into design elements (subsystems and design classes) – maintaining the mapping from analysis elements to the design elements that actually implement them is critical to effective requirements-to-model traceability.
If students ask about how interfaces are implemented, the following explanation from Kurt Bittner may help:

> "Here's a sketch of approximately what will happen:
> The 'client' has dependency relations on a set of interfaces, so that any element which realizes the interface should be substitutable. At code-generation time, the code generator replaces the interface dependency with a dependency on a 'smart stub' which allows run-time substitution of elements, which realize the specified interface. On the server side (the element which realizes the

interface), the code generator also creates the code necessary to do the dynamic linking at run time (essentially the complement of the 'smart stub'). If you're running in a CORBA or COM environment, IDL is generated, too. Note that static linking could be done as well, in which case the bindings occur at code-generation time.

Basically what's happening is operation indirection (if you've ever worked with C, there's an obscure ability to execute a pointer to a function; this is sort of similar). Let's say ClassA needs opA(parm1,parm2) on InterfaceI. Let's also say that ClassC and ClassD both realize InterfaceI, and that opC(parm1,parm2) on ClassC and opD(parm1,parm2) on ClassD both realize opA() on InterfaceI. So if we want to statically bind ClassC to InterfaceI, at code generation time calls to InterfaceI.opA(parm1,parm2) will be bound to ClassC.opC(parm1,parm2). If you were to look at the literal code generated, you would no longer see InterfaceI, you'd only see ClassC. Smart stubs simply do this magic through indirection, sacrificing a bit of performance for the flexibility of run-time binding.

There are a couple of books that describe for CORBA and COM the specifics if you're interested in the details (I've oversimplified a bit), but basically the code generator inserts some code on both the client and server sides to realize the interfaces."

Subsystems vs. Packages: Subsystems have distinct, "abstractable", and extractable responsibilities (their responsibilities are more than the sum of the responsibilities of the classes contained within the subsystem). Packages are just for grouping things together – no hidden meaning or architectural control. The traditional way of using packages with a limited number of public classes has been replaced with subsystems and interfaces. You can think of subsystems as a special kind of package, a package with a limited, well-controlled interface. Packages demonstrate modularity and subsystems demonstrate encapsulation. Remind the students that there are certain criteria that warrant the use of subsystems (e.g., replaceable/substitutable, reusable, etc.). Not everything is a subsystem ;-).

Subsystem Modeling Convention: Stress to the students that in this course we will be following the convention described in Architectural Design for modeling subsystems. The convention is that for each subsystem, you should define:

- A package with a stereotype of <<subsystem>>
- A class within the <<subsystem>> package with the same name as the package and a stereotype of <<subsystem proxy>>. The <<subsystem proxy>> class realizes the interfaces that the subsystem is to realize.

Some students may complain that we use this convention because Rose does not directly support subsystems. While this may be true, it is not the only reason we use this convention. See the **OOAD and Rose Gaps** section for more information on positioning the lack of subsystem support (and other concepts) in Rose.

Architectural Mechanisms (general concept): For whatever mechanisms are chosen, the architect must provide a static picture (class diagram) and a dynamic picture (interaction diagrams) of how they work. That way the designer will know how to apply the mechanisms during design. The architect is responsible for developing a "Designer Handbook" that guides the designer in how to perform the detailed design of his design elements so that they are consistent with other design elements and will fit into the overall architecture. Architectural mechanisms and how to use them are documented in that "handbook", also known as the Modeling Guidelines.
The analysis mechanisms identified in Architectural Analysis are refined into Design and Implementation mechanisms in Architectural Design.
Also emphasize that the use of the specific implementation mechanisms (e.g., ObjectStore, RMI, Java, etc.) are just examples, not recommendations or endorsements.

Points NOT to Emphasize

Architectural mechanism details (e.g., persistency, security, etc.). The goal is not to teach to students how to design these mechanisms, or to discuss the pros and cons of the different choices. They just need to understand how the choices affect their design. After all, when they all go back to their projects, their architects will probably choose a different set of mechanisms. The student (i.e., designer) just needs to know what the mechanism means to him and how it affects what he does.

Describe the Run-time Architecture

Points to Emphasize

Process View: The focus of Describe the Run-time Architecture is the development of the Process View of the architecture.

UML notation for the Process View: What UML elements can be used to represent the Process View. Two options: Logical/Design Model elements (active classes, dependencies, compositions) or Component/Implementation Model elements (modules/components and dependencies). The course chooses the former representation.

Points NOT to Emphasize

How to come up with the Process View. This is not an architecture course, so we won't teach the students HOW to come up with the Process View, only what one looks like (e.g., the UML notation).

Describe Distribution

Points to Emphasize

Deployment model: The focus of Describe Distribution is the development of the Deployment Model.

UML notation for the Deployment Model: What UML elements can be used to represent the Deployment Model.

Points NOT to Emphasize

How to come up with the Deployment Model. This is not an architecture course, so we won't teach the students HOW to come up with the Deployment Model, only what one looks like (e.g., the UML notation).

Use-Case Design

Frequently Asked Questions

In the RUP Analysis and Design workflow, Use-Case Design follows Subsystem and Class Design. In the OOAD course, Use-Case Design precedes Subsystem Design and Class Design. Why?
Note: This question does not come up that often now that I have used a tailored version of the workflow in the course slides. However, someone may still ask this if they are familiar with the RUP Analysis and Design workflow.
Once you have morphed your analysis elements (defined in Use-Case Analysis) into design elements (defined in Architecture Design) you must refine the analysis use-case realizations (written in terms of analysis elements) into design use-case realizations (written in terms of design elements). This is the purpose of Use-Case Design. In Use-Case Design, you also incorporate the architectural mechanisms.

Thus, the (very simple) analysis use-case realizations, morph into (more complicated) design use-case realizations. Use-Case Design is where the responsibilities are re-allocated to the design elements. Only after you have re-allocated those responsibilities and made sure that nothing has been missed, should you dive into the detailed design of the design elements, which is what you do in Subsystem and Class Design. Subsystem and Class Design are essentially "peers" – each performing the detailed design of a design element – a subsystem and a class, respectively. Of course, there is a lot of iteration amongst these detailed design activities – when doing Use-Case Design, you may find additional subsystems and classes, so additional instances of Subsystem and Class Design must occur. During Subsystem Design, you may find additional subsystems and classes, so additional instances of Subsystem and Class Design must occur. During Subsystem and Class Design you may need to alter their interactions with other design elements, so you may need to re-visit Use-Case Design to make sure everything still flows, etc. I felt the order implemented in the class flowed a little better than in RUP.

Points to Emphasize

Use-Case Realization Refinement: Use-Case Design is where the analysis use case realizations meet the identified interfaces and the design/implementation mechanisms. Use-Case Design is where the use-case realizations initially defined in Use-Case Analysis are refined to incorporate the design elements and the architectural mechanisms defined in Identify Design Mechanisms, Describe the Run-time Architecture, and Describe Distribution (e.g., following the "Design Handbook"). The analysis classes that appeared in the use-case realizations during Use-Case Analysis, need to be replaced with the design elements (subsystems and design classes) that the analysis classes morphed into in Identify Design Elements. Subsystems should be represented by their interfaces on the interaction diagrams. At this level, Use-Case Design becomes a substitution exercise. This becomes more interesting when you then apply the architectural mechanisms documented in the "Design Handbook" mentioned above. The use-case realizations (i.e., interaction and class diagrams) are then updated to include the architectural mechanisms.
Be sure the emphasize how the mechanisms are incorporated (not developed). Demonstrate to the students how the mechanisms chosen/defined in Identify Design Mechanisms and Describe Distribution are applied.

Responsibility Allocation: There's not really anything different from Use-Case Analysis happening in Use-Case Design. The focus is still on making sure that all the use case responsibilities have been allocated appropriately, but now you are allocating to design elements (design classes and subsystems), rather than analysis classes. You need to do this to make sure that you have a good starting point to kick off Subsystem Design and Class Design. By doing a good job during Use-Case Design, you can have multiple instances of Class and Subsystem Design occurring in parallel. In Use-Case Design, you decide what responsibilities are allocated to the subsystems (representing these as operations on interfaces, with the interfaces appearing on the interaction diagrams). The details of how a specific subsystem implements those responsibilities are deferred until Subsystem Design. The same is true of the design classes. In Use-Case Design, you decide what responsibilities are allocated to the design classes (representing these as operations on the design classes, with the design classes appearing on the interaction diagrams). The details of how the design class implements those responsibilities are deferred until Class Design.

Points NOT to Emphasize

Architectural mechanism details: The goal is not to teach to students how to design these mechanisms, or to discuss the pros and cons of the different choices. They just need to understand how the choices affect their design. After all, when they all go back to their projects, their architects will probably choose a different set of mechanisms. The student (i.e., designer) just needs to know what the mechanism means to him and how it affects what he does.

Subsystem and class details: As described above, the details of how a specific subsystem or class implements a responsibility are deferred until Subsystem and Class Design, respectively.

Subsystem Design

Points to Emphasize

Interface Realization Development: The focus in Subsystem Design is the detailed design of a subsystem. This involves looking at the responsibilities that have been allocated to the subsystem in Use-Case Design, and identifying what design elements are needed to implement those responsibilities. Remember, for subsystems, the subsystem's responsibilities are captured in the interfaces the subsystem realizes. The good news here is that we are not doing anything new. The identification of design elements, allocating responsibilities, and representing these decisions on class diagrams and interaction diagrams is exactly what we did in Use-Case Analysis and Use-Case Design. The only difference is that we are focussing on subsystem responsibilities (i.e., interface operations) rather than use cases. Thus, we produce "interface realizations" – at least one interaction diagram per interface operation, and then a subsystem main class diagram with relationships to support the collaborations (a VOPC for the subsystem, if you will). In Subsystem Design, the designer concentrates on the structure (class diagram) and the dynamics (interaction diagrams) of the subsystem elements.

In Rose, interface realizations can be documented using a variant of the use-case realization (i.e., the Rose version of a UML collaboration) idea. Essentially, a <<interface realization>> stereotyped use case is created inside the <<subsystem>> package with the same name as the interface it models. Thus, there is one <<interface realization>> use case (e.g., UML collaboration) for each interface the subsystem realizes. The interface realization contains the static and dynamic models that describe how the subsystem realizes the interfaces (e.g., a class diagram and an interaction diagram – at least one interaction diagram per interface operation. As with use-case realizations, the interface realization does not contain any classes (they "live" in the <<subsystem>> package or in other design element packages).
This fits well with the large-scale 'systems-of-systems' development idea where you would have subsystem use cases and their corresponding realizations, and also keeps the diagrams neat.
This convention is not part of the tool mentors yet, bit it was used in the sample models provided with the course materials.

Modeling Conventions: Always start the subsystem interaction diagrams with a subsystem client sending a message to the subsystem proxy class (NOT the interface) and the subsystem proxy class then sending messages to other objects to perform the operations. The subsystem proxy class orchestrates the interactions of the design elements to carry out the subsystem responsibilities. The subsystem client does not have to mapped to a specific class. Remind the students that interfaces cannot send messages – they have no implementation. It is the proxy class that truly realizes the interface and delegates the work to the other design elements.

Points NOT to Emphasize

Architectural mechanism details: The students can use the interaction diagrams given in the Subsystem Design module to drive what they develop. Alternatively, the instructor can skip the application of architectural mechanisms completely if he/she thinks that they will only confuse the students and cause them to lose sight of the goal of the module – interface operation realization development.

Class Design

Points to Emphasize

Focus is on fleshing out the class internals.

Relationship Design: Make sure the students understand what makes a relationship structural (association or aggregation) or not (dependency). Any time you define a dependency, you must be able to describe the visibility that drives it (e.g., local, parameter, or global).

Incorporation of Generalization: Generalization/inheritance is a very important and powerful concept in OO. Thus, it is important that the students understand all the ways that inheritance can be applied to the model during design. One important use is the use of generalization to support metamorphosis. Such use of generalization is the purpose of the exercise in that section.

Points NOT to Emphasize

The language specifics inherent in the class operation and attribute signatures. It is important for the students to know that implementation language rules affect what the operation and attribute signatures look like, but don't dwell on it. This is not a programming course.

Course Exercise Hints

Architectural Exercises

The architectural exercises are the exercises for the architectural modules: Architectural Analysis, Identify Design Elements, Describe the Run-time Architecture, and Describe Distribution (there is no exercise for Identify Design Mechanisms).

Some instructors and students question the value of the architecture exercises (some feel they are just a drawing exercise). The following is my rationale for including them:

> The students should have to draw the UML diagrams that represent the various architectural concepts. The OOAD/UML course is not an architecture course, but it is important that the students get an idea of what UML architectural diagrams look like. Having some not so hard exercises that focus more on notation than actually having to figure out analysis and design issues is a nice way to get things going as we approach a lot of detail in the hard-core design activities.

Exercise Solution

The Payroll Solution appendix contains a "chapter" per module. Except where noted, each module only contains diagrams explicitly asked for in the exercise description. However, the soft-copy of the Payroll System Rose model is more complete (there is much more in the solution than can be completed in four days). It is a fairly complete model of the Payroll System. It also contains additional diagrams that show different aspects of the payroll solution. The students need to understand that they will only be producing part of the solution throughout the four days. A fairly complete model is provided in soft-copy form to demonstrate to the students how things end up, and to show the students that the exercise is based on a realistic problem and has a consistent and well thought out solution.

The Payroll Exercise Solution contains a relatively complete analysis and design of the main flow of the following use cases:

- Login
- Maintain Timecard
- Run Payroll

This includes the Use-Case Model, Analysis Model, Design Model, Process View and Deployment View.

For the remaining use cases, only the Use-Case Model is provided.

The Payroll exercise solution contains separate use-case realizations that demonstrate the incorporation of the different architectural mechanisms. There is one use-case realization for each use case for each mechanism, plus a use-case realization that incorporates all the mechanisms. This was done to make it easier for instructors to teach some of the mechanisms.

During Class Design, the introduction of the Payment Method class to support metamorphosis, as well as the ability of that class to know how to execute itself (including communication with the external printer and bank system) introduces a circular dependency between the Payroll Artifacts and External system Interfaces packages. This was left and the design trade-off was documented in the Payroll Solution appendix.

Exercise Facilitation Hints

This section provides exercise facilitation ideas for the module exercises. In addition to reviewing this section, be sure to have a look at the **Course Flow Options** section if you feel that you need to tailor the course delivery based on your audience's skill level. For a description of the key points of each module, see the **Frequently Asked Questions / Points to Emphasize** section**.**

Points to Emphasize

There is NO SINGLE RIGHT answer
Some answers are better than others are. All answers evolve and improve over time, but there is NO single right answer. Be sure to weigh the pros and cons of the student answers, emphasizing the trade-offs that are made. Have the students present the rationale for their decisions.
The exercise solutions provided in the material are one set of possible answers to the problems – more specifically the course author's answers to the problems. They are NOT THE ONLY answers. They can be used just like any student's answers. They can be presented and pros and cons discussed. They can be compared and contrasted to the other answers presented.
It's all right to suggest that a portion of the model needs improvement. You want to make sure that if a student presents something that is way off base from how it was described in class, that the other students don't walk away and think that it is OK. "Really wrong" answers need to be fixed and the students need to understand what was "really wrong". I use "needs improvement" rather than "really wrong" when discussing this with the student ;-). You don't want to leave a blatant error unmentioned. It's important not to (directly) lead, nor mislead, your students.
Try not to change student answers for them. Discuss some improvements and implicitly lead them to the conclusion that their model needs adjusting. In this way, they understand what was wrong with it and how it can be fixed – a valuable skill.
Be careful about changing just part of an answer without talking it through. There is always the potential that you will leave something inconsistent or an issue unresolved.

Guidance on How to Lead

The following sections describe some ways to handle the course exercises.

Exercise Set-Up

When setting up the exercise, the instructor should:

- Convey the purpose of the exercise, emphasizing the key concepts that are being exercised
- Make sure that the students can locate all of the inputs (having the students mark the pages, if necessary).
  It may help to write actual page numbers of the givens on the white board (these are not included in the student and instructor notes due for maintenance reasons)
- Review the process the students should use to apply the concepts that result in the expected deliverables. It is important that the instructor describe how the exercise inputs/givens are used to develop the requested diagrams.

- Make sure the students understand what the deliverables are and what they should look like.

To make the exercise set-up clearer, all of the exercises have the same general format. There are three sections in the exercise description: The exercise inputs (or givens), what needs to be done, and what needs to be produced (the deliverables, if you will). The Student Notes include references to where the exercise inputs/givens can be found, brief explanation of what needs to happen and how the exercise inputs are used, as well as references to examples of what is to be produced. The instructor notes also contain references to the exercise solution. Be sure to use the exercise description slides to guide the exercise set-up.

Remember that the exercises are where the students learn to apply the concepts they just learned in the associated course presentation. Thus, if you emphasized some key concepts during the course presentation, you want to concentrate on those same concepts in the exercise.

Team Exercises vs. Individual Exercises

There are always trade-offs as to whether to have the students do the exercise individually or have them work in teams. Some exercises are more suited for brainstorming amongst team members (identifying classes and allocating responsibilities), while others are more suited individual work (e.g., generating a state diagram). A good heuristic to use is to have the students work as they would on a real project. (Note: The general consensus among instructors is that students get more out of working with someone else rather than working alone).

The optimum team size is 3 students – do a 2 or 4-student team to round out the teams when necessary. Typically a class is 12 students, so 4 teams of 3 students usually works fine.

It also may be necessary to do some gerrymandering of the teams. Make sure the teams have a balance of skills as best as possible – for example a domain expert, an analyst, and an implementer. It's a judgement call in each class as to whether to allow the students to group themselves or not. Don't worry about this too much because you can always regroup the students after a couple exercises and still make them continue with the models they have been working on.

Another option is to just go around the room assigning team numbers to the students (the number you assign them is their team number). This random assignment is generally acceptable. If there are problems with the resulting teams, you can adjust them later.

Team Roles

Tell the students that there are a couple roles on their team that have to be played. The primary role is that of a Scribe. One person stands at the easel pad and captures salient ideas as they are discussed amongst the group. The other team members actively participate in the discussion. Specifically state that the Scribe role must be rotated – either during a long exercise or between different exercises – everyone takes turns being the Scribe.

Another role is the person who will present the teams answer to the class in the design review. This role should be rotated amongst team members, as well.

Instructor Involvement

The instructor must be able to lead a class through exercises and help teams develop a solution.

While the teams are working on the exercises, the instructor should walk around and listen to how all the teams are doing. The instructor should observe how the students apply the concepts, what problems they are encountering, what concepts they don't quite understand, as well as any interesting ways they have chosen to apply the concepts.

When the instructor sits and listens to the teams, the team's natural tendency is to look to the instructor for answers while they're discussing their work. The instructor should be careful not to tell them what their model should look like. Instead, the instructor should ask what they are doing and why they are doing it and then make suggestions to evolve their thinking (Socratic method), or ask the appropriate questions to help them come up with the answer on their own. Focus on evolving their thinking, not telling them what they her should do. Refuse to take the pen and lead them through their exercise.

Keep the teams focused on the concepts being exercised. Recognize when the teams stop making progress and bring them together to discuss their answers.

Model Reviews

One of the best ways to learn is to see how everyone did their exercises and be able to compare and contrast the different answers and discuss the pros and cons as a class. This serves as an example for the class on how to handle model reviews. It also serves to show them that they just need to get some modeling captured so they can communicate it and start discussing it – then all sorts of good ideas flow forward!
This is contrasted to where the students may try to hide in a room until they can justify all the elements of their model which gives them time to become very attached to their model. You want to create an open atmosphere where students are willing to accept constructive criticism and let their models evolve.
That being said, the instructor still must maintain some control of the class during exercise facilitation. It's important that the instructor provide guidance, focus and synthesis of the class discussions. Otherwise, class chaos can be the result, there is no closure for the exercise, and the students become frustrated. Discussions on different ideas are good but the instructor needs to make sure that the class understands what is under debate and what the pros/cons of the different opinions/solutions are.

Each Team/Student Presents Their Results

Each team/student is responsible for presenting to the class the results of their exercise. Then providing an open forum for discussing their model and the trade-offs they made. Students should not be defensive but articulating their model clearly and looking for good ideas for improvement.
The teams should present in whatever form is convenient for the class. A few ways to do this include white boards, easel pages, or transparencies.

Critique One Another

Everyone is responsible for critiquing one another. Keep the entire class involved. When a team/student presents their exercise results, the first thing the instructor should do is have the class provide feedback and ask questions about the model. The instructor should let the class drive this; and then corral the discussion into interesting salient points. Initially, until the class gets used to the style, they will naturally look to the instructor for what he/she thinks of their answer to the exercise. Don't give it to them right away – make the class provide feedback. My experience is if you do this, most of the things the instructor would have said come out in the discussions the other students bring up; and often times, other interesting things are brought up by the other students that the instructor may not have noticed or thought of.

Right vs. Wrong Answers

Get people away from saying that an answer is right or wrong. One technique to do this is to ask specific questions: Did the team use the notation correctly? Did this team meet the requirements of the exercise? Are there advantages to the way this team modeled their problem? Compare and contrast different answers, keeping in mind the rationale and thought process behind the design decisions. This is excellent practice for the real world where generally there is not a single solution to fall back on, but a set of solutions, all of which may have their strong points.

Instructor Feedback

The instructor must be able to critique, in class, the answers to exercises and discuss student solutions and NOT just present the book answer – the class should be able to come up with better answers than are in the book.
The instructor should add commentary to the design reviews as well, particularly synthesizing what the class is saying as feedback. The instructor can also provide observations based on what they notice as they watch the teams work on the exercises.
There is sometimes value in discussing, not just the OO characteristics of the model, but also the process and techniques that the teams are using to work together.
The instructor should always prompt the students to describe their thought process and the rationale behind their design choices, the tradeoffs they made, etc.

The instructor should be sure to provide positive feedback when at all possible. This helps the students to gain confidence in their ability to apply the concepts.

Key Exercise Issues

It is always a good idea to discuss any key issues after the exercise (i.e., discuss with the class the concepts, topics, etc. that many of the teams seemed to have trouble with).

Baselining vs. Building upon Students' Answer

After each exercise, you can either have the teams build upon their previous answer or you can baseline them with the either the book answer or homogenize the individual team answers into a class solution. There are trade-offs with either approach.
In either case, the exercise needs to result in a solution that the students understand and can describe, rationalize, defend. Such a solution can be the class solution or the team/student's solution. The solution doesn't have to be complete in every respect (there isn't enough time in class for such a task); however, it should at least be consistent.

Non-Baseling

Some instructors have the students/team build upon their individual answers – the classes answers start to diverge and it causes interesting discussions during model reviews.

Baselining

Other instructors baseline the answers in order to keep the teams at a similar level of work and pace. You can use baselining as a means to demonstrate the homogenization of different ideas/solutions. Some instructors prefer the baselining method because they feel it models what happens in real life where you may have differences that need to be resolved.
When baselining, you can have the teams start from the solutions provided in the exercise solution appendix, or start from the baselined class solution to the previous exercise. In either case, it is very important that the students have something consistent on which to start each exercise. Using the book answer makes this a little easier because the instructor can just say "start with what is on page x". However, if you baseline to a class answer, the instructor has to do a little more work. The instructor can record the class solution for an exercise or can ask a student to (either on paper or in Rose). Be sure to include the exercise name at the top of the diagrams, so they are easy to identify later. The instructor then makes copies of the solution and distributes it to each of the students in time for the next exercise. While this may seem like more work, it does allow the students to take home another solution to the exercise – one they had a part in developing.

Note: When baselining to a class solution, the references to the givens for each of the exercises will need to be replaced with references to the baselined class solution.

Showing Book Answer

Whether the book answer is shown or not is up to the instructor. However, even if the instructor baselines to a class solution, it is recommended that the instructor discuss the provided solution with the students, comparing and contrasting the solution with the presented solutions. In this way, the students are exposed to what is meant to be a "good OO solution", and the instructor can increase the chances that the students understand the contents of the course material that they will be taking with them when they leave the course.

Additional facilitation hints are provided in the instructor notes for each of the exercises in the Instructor Manual. The exercise notes include details on how to run the course exercises, as well as references to sample diagrams (examples of what the solution should look like), and the actual solutions (both in the model and in the hard copy).

Course Review

It is important to hold review at the end of each day, as well at the end of the course, so there should be no surprises for you or the class that some topic was not covered or discussed. See the **Review the Material** section in the *Rational University Instruction Guide*.

There is not a formal review module in the OOAD course materials. It is up to the instructor to provide a comprehensive review of the course, using the information provided in this section.

Some ideas include:

- Use the material developed during the class kick-off related to what the student expectations for the course were. Review what has been covered and how and where the topics they identified were covered.
- Use a course roadmap, with a list of results/deliverables (see **OOAD/UML Course Roadmap** section). . It is important to make sure the class understands the steps for applying the OOAD concepts on a project. The course roadmap outlines these steps.
- Use a model element summary, with a list of the model contents and their organization (see **OOAD/UML Model Element Summary** section).
- Use the review slides provided at the end of each module.

Course Flow Options

OOADv2000 contains a significant amount of information. The instructor may find it necessary to tailor the course delivery based on the audience's skill level, objectives, or expectations. This section describes some possible "paths" through the course.

Note: Whenever you alter the flow of the course, forward and dangling references to skipped and re-ordered modules are always possible. It also means that some examples may refer to topics that were skipped. Examples that don't include the skipped topics may need to be "white-boarded" by the instructor. It is CRITICAL that you walk through the complete course in the order/organization you plan on delivering it to make sure it flows OK. You also need to make sure that the exercises still work, given your adjustments.

In addition to reviewing this section, be sure to have a look at the **Frequently Asked Questions / Points to Emphasize** section for a description of the key points of each module, and the **Course Exercises** section for hints and techniques on the exercise facilitation for each of the module exercises**.**

Always start simple. You can always expand and customize, as you become more comfortable with the course material and the included concepts.

Intermingling the OOAD and Rose Courses

There are certain trade-offs when considering whether to use a tool during technology training (e.g., "should you teach Rose at the same time you are teaching OOAD?"). Keep the following in mind when deciding whether or not to use a tool during technology training:

- Remember the objectives of the course – to convey knowledge of the technology. The tool, if used, must not get in the way of learning the concepts. If the tool does get in the way, you are defeating one of the prime purposes of the class.
- Early on in the course, brainstorming is important. For small teams, brainstorming from easel pages is often easier than brainstorming in front of a computer terminal.

There isn't a black and white answer. Tools and technology are not mutually exclusive. The following are some options you can consider:

- Start the first day or so without utilizing any of the tools. Once the class has started to learn and apply some of the concepts, introduce the tool.
- Use the scribe method. Use the in-class time to work in teams on the exercises; then after class capture the current state of the model in Rose and bring it to class for review. This allows both the brainstorming on easel pads as well as utilizing the tool.
- Demo the tool using the class exercise during breaks or lunch.

Note: Whether you actually use the tool during the technology training, or will be teaching it after, references to the tool during the technology training are not a bad idea. It allows the students to see what areas of the technology can be automated. However, be careful about mentioning Rose explicitly if the students are sensitive to being "sold to".

It is possible to teach OOAD "hands-on", doing the exercises in Rose, rather than on paper. Such intermingling may offer some pacing of the amount of material that is presented and applied, thereby giving the students some time to absorb what they have learned. It also provides a context to apply the information in the tool, which is what Rose customers are really looking for.

The following describes guidelines for intermingling the OOAD and Introduction to Rose courses.

If you intermingle OOAD and Rose, the recommended order of the intermingled modules is as follows (the corresponding OOAD and Rose modules are listed in the table below):
1. Present the OOAD module, up to the exercise
2. Present the Rose module, up to the exercise
3. Do the OOAD exercise using Rose
4. Skip the Rose exercise
5. Do the OOAD Review
6. Do the Rose Review

Note: The OOAD instructor notes also include Rose hints and tips, where applicable, and the OOAD Rose models for the exercise (the Payroll problem) are "incremental". There is a model file for each exercise that reflects what the model would look like AFTER the exercise was completed. In that way, the students can use the model from the previous exercise as a starting point for the next exercise. This can be used to keep everyone in sync in case some students fall behind and don't complete a previous exercise.

Note: Where the table entry is empty, there is no associated module mapping.

OOAD v2000 Module Title	**Introduction to Rose 98i v5.2 Module Title**
About this Course	- About this Course
Best Practices of Software Engineering	- None
Concepts of Object Orientation	- Introduction
Requirements Overview	- Use-Case Diagram - Activity Diagram
Analysis and Design Overview	- None
Architectural Analysis	- Class Diagram (emphasizing packages, package relationships, allocating classes to packages)
Use-Case Analysis	- Team Development (Only required if you want to use

	Rose controlled units right away to control what the different use-case teams develop; otherwise, Rose Team Development can be postponed to Architectural Design) - Collaboration Diagram - Sequence Diagram (The Rational Unified Process (RUP) recommends the use of Collaboration Diagrams in analysis and Sequence Diagrams in design. If you adhere to this, then you really don't need to cover sequence diagrams until Use-Case Design.)
Identify Design Elements and Identify Design Mechanisms	- Team Development (We really don't introduce any new diagrams in these architectural modules, but this is a good point to discuss parallel development in Rose, because parallel development is something that is enabled by a strong architecture. Note: If you want to use Rose controlled units right away to control what the different use-case teams develop, you will need to introduce Rose Team Development in Use-Case Analysis.)
Describe the Run-time Architecture	- Component and Deployment Diagrams (component diagrams only) - (We don't discuss the creation of components in the course anywhere really. This is really an Implementation activity (Structure the Implementation Model). However, the Component View can be used to model the Process View. Thus, if this option is to be used, the Rose Component Diagram module must be presented. Otherwise, it is not really necessary.)
Describe Distribution	- Component and Deployment Diagrams (deployment diagrams only)

Use-Case Design	- Sequence Diagram (if not already covered after Use-Case Analysis)
Subsystem Design	- None
Class Design	- State Diagram
	- RoseScript
	- Printing and Reporting

General Tips and Hints

Color on the Slides

Many of the slides use yellow or blue to highlight important concepts. Most notably, yellow or blue is used to distinguish what changes have been made to the design to incorporate the architectural mechanisms. Such use of color is not visible in black and white manuals. To make up for this, the slides also contain arrows pointing to the changes. However, the instructor can suggest that students use the yellow highlighter found on one end of the Rational University pen that is included with the course materials to highlight in their manuals what is colored on the slide.

Easel Pads / White Boards

If you wish the team exercises to be done on easels, try to get 4-5 easel pads (or as many as you can get) so that the students can use these during their exercises and the instructor can use them to write on during the lecture. The value of easel pages over white boards is that the easel pages don't get erased, the easel pages can be hung up on the wall where everyone can refer back to them. Make sure there are sufficient pens for both the easel pads and white boards. If using easel pads, make sure there is masking tape so you can hang some easel pages on the wall for reference (unless the easel pages are self-adhesive).
If there are not going to be enough easel pads and/or white boards to do model reviews, you may need to be sure there are a lot of blank transparencies and pens so they can transcribe their answers to be projected on an overhead projector (if one is available).

Notation Page

If you are using easel pages and hanging them on the walls for students to reference, it may be valuable to have a page dedicated to the elements of the notation. As you learn a new piece of notation, you can add it to the notation page. This page can then be referenced often and can be used as a review of what has been covered in class.

OOAD/UML Course Roadmap

Understanding and articulating how all the course sections fit together and why they are being presented in the order in which they are, is very important if the students are going to be able to apply the concepts on real-life projects. Being able to "tie it all together" effectively comes from understanding the overall process framework, a definite requirement for all instructors.
The instructor needs to understand which of the course concepts are important, which ones to stress for each section and emphasize those, possibly using a course roadmap to guide them (this can be developed ahead of time and built incrementally in class). The instructor should give an overall picture of where the class is going and continue to explain how all the course modules they fit together, reviewing and previewing sections, and review and reiterating important concepts.
An effective way to teach a technology is to give the class a simple recipe for what they must do. While learning, the class often needs this and often asks for it. While teaching the course material, you can record the steps on an easel page as they are being introduced and taught in the material. The result is, in the

midst of the second day you have an easel page with a set of bullets on it. With this on an easel page hanging in the room, it serves as a good reference throughout the course and a refresher to them regarding what they should be doing during their exercises and a good review form.
Emphasize and re-emphasize that, in reality, the process is never this sequential. Simply serialize it here to put a stake in the ground and give the students an example to learn from – in reality all this will be meshed together and happening in some fashion concurrently.

The course roadmap is conceptually the same thing as the cookbook mentioned in the *Rational University Instruction Guide*. Please see that document for more general information.

The following describes the basic steps in the process described in the OOAD/UML course, as well as the results/deliverables. These topics can be abbreviated and written on an easel chart as the class progresses. They can then be used as a guide for reviews, and exercises.

Note: I usually build this course roadmap incrementally on a flip chat after I complete a module, asking for class input to their content, rather than just handing "my version" out. This serves as an excellent review of the module, and encourages class participation. You can then choose whether or not you want to hand out "the formal version" at the end of class.

Activity	**Results/Deliverables**
Inputs to Analysis and Design (outputs of Requirements Workflow)	- Use-Case Model (actors, use cases, and their relationships) - Supplementary Specification - Glossary
Architectural Analysis ✓ Define modeling conventions ✓ Define analysis mechanisms ✓ Define initial upper-level architectural layers and their dependencies ✓ Define key abstractions	Analysis/Design Model - Upper-level architectural layers and their dependencies - Analysis mechanisms - Key abstractions
Use-Case Analysis ✓ Supplement use case descriptions ✓ Find classes and distribute behavior ✓ Describe responsibilities, attributes and associations, and qualify analysis mechanisms ✓ Unify analysis classes ✓ In summary, develop analysis use-case realizations	Analysis/Design Model - Analysis classes (boundary, control, entity), their responsibilities, and their analysis mechanisms - Use-Case Realizations (interaction diagrams for subflow(s), VOPCs)
Identify Design Elements and Identify Design Mechanisms ✓ Identify and define design and implementation mechanisms ✓ Define interfaces and subsystems ✓ Identify reuse opportunities ✓ Refine architecture layers (concentrating on the lower level layers)	Design Model - Architectural layers, packages, their relationships, and their contents (now includes lower-level layers and packages) - Subsystems, Interfaces, and their relationships - Design classes (some analysis classes may be combined, split, turned into subsystems, etc.) - Analysis class to design element mapping - Architectural mechanisms map (analysis mechanisms to design mechanisms to implementation mechanisms) - Patterns of use for the key mechanisms (e.g., persistency, security), including any necessary support classes

Describe the Run-time Architecture ✓ Identify concurrency requirements ✓ Model processes ✓ Map the processes to the implementation environment ✓ Map model elements to processes	Process View - Processes and their relationships - What design elements run in what processes
Describe Distribution ✓ Analyze distribution patterns and network configuration ✓ Allocate processes to nodes	Deployment Model - Nodes and their connections - What processes run on what nodes - Patterns of use for distribution, including any necessary support classes
Use-Case Design ✓ Define interactions between design objects, incorporating any architectural mechanisms ✓ Encapsulate Common Subflows ✓ Refine Flow of Events description ✓ Unify classes and subsystems ✓ In summary, refine the analysis use-case realizations developed in Use-Case Analysis into design use-case realizations	Design Model - Refined Use-Case Realizations (interaction diagrams for subflow(s), VOPC), including any new/refined design classes. - Incorporation of patterns of use for the architectural mechanisms (interaction diagrams should show necessary collaborations and class diagrams should reflect the supporting relationships) - Refined class, subsystem, package and layer dependencies to support design element dependencies (may require architectural approval)
Subsystem Design ✓ Distribute subsystem behavior to subsystem elements ✓ Document subsystem elements ✓ Describe subsystem dependencies ✓ In summary, develop interface realizations	Design Model - Interaction diagrams for each interface operation or public class operation - All subsystem elements and their relationships - Refined subsystem, package and layer dependencies to support design element dependencies (may require architectural approval)
Class Design ✓ Create initial design classes ✓ Define operations ✓ Define methods ✓ Define states ✓ Define attributes ✓ Define dependencies ✓ Define associations ✓ Define generalizations	Design Model - Complete class definitions, including complete signatures for all attributes and operations - State charts for classes with significant state - Updated/refined class relationships (composition, dependency relationships, more navigability, etc.) - Refined subsystem, package and layer dependencies to support design element dependencies (may require architectural approval)

Model Element Summary

The following outline lays out the model organization that is followed in the OOAD course. It reflects the basic layout of the models for the course example and exercise that are delivered with the course.

This Model Element Summary can be used instead of, or in addition to, the OOAD/UML Course Roadmap provided earlier. It is generally more applicable if your students are “Rose-friendly”. It is especially beneficial if you are intermingling OOAD and Rose (see the **Intermingling the OOAD and Rose Courses** section).

Note: The models may contain additional diagrams. This outline just includes the key ones.

Note: I usually build this model element summary incrementally on a flip chat after I complete a module, asking for class input to their content, rather than just handing "my version" out. This serves as an excellent review of the module, and encourages class participation. You can then choose whether or not you want to hand out "the formal version" at the end of class.

Use-Case View
- Main diagram (all actors, use cases, and their relationships)

- Actors
- Use Cases
  - Attached use-case specification document

- Attached problem statement document
- Attached glossary document
- Attached supplementary specification document

Logical View
- Design Model
  - Main diagram (contains layers)

  - Use-Case Realizations package
    - Traceabilities class diagram (shows relationships from use case realizations to use cases in the Use Case Model)

    - Specific use-case realization packages
      - Use-Case Realizations (<<realization>> use cases)
        - View of participating classes (VOPC) class diagram
        - Interaction diagram(s) for the use-case flows of events

  - Architectural Mechanisms package
    - Specific architectural mechanism packages
      - Architectural Mechanisms (<<mechanism>> use cases)
        - Class diagrams
        - Interaction diagram(s)

  - Layers (<<layer>> packages)
    - Main diagram (contains design elements in layer)

    - Packages
      - Main diagram (contains design elements in package and their relationships)
      - Classes
        - Operations
        - Attributes
        - State diagram
      - Packages
      - Subsystems (<<subsystem>> packages)
      - Relationships
      - …

- Interfaces (<<interface>> classes)
- Subsystems (<<subsystem>> packages)
    - Main diagram (contains design elements in subsystem and their relationships)
    - Classes
        - Operations
        - Attributes
        - State diagram
    - Packages
    - Subsystems (<<subsystem>> packages)
    - Relationships
    - Interface Realizations (<<interface realization>> use cases)
        - View of participating classes (VOPC) class diagram
        - Interaction diagram(s) for the subsystem responsibilities/operations

Process View

Main diagram (contains all processes and their relationships, as well as what design elements have been allocated to the processes)

Processes and threads (<<process>> and <<thread>> classes)

Deployment View

Deployment diagram (contains all nodes and their connections, along with the processes that have been mapped to the nodes)

Nodes

The processes that have been mapped to the nodes (the processes should have the same name as those processes in the Process View)

Additional Exercises

A Different Exercise

Many instructors bring in additional problem statements for exercises; they either use these problems as a replacement exercise to the one in the course or in addition to the one in the course.

Iteration 2

Many instructors take the time to make the students refine the model they have been working on based on the critiques; thus, creating a second iteration of the model.

Requirements Change

Give the students a requirement change that causes them to rework parts of their model. Discuss the impact of the requirements change and what they learned they could have done to be better able to handle that type of requirements change.

Object Game

Some instructors have the students do a CRC card exercise. The way that this has been successful is if they do it in the same fashion that Peter Coad does it. Take an easel page paper and crumple it up into a ball; have the team set around a desk; and they throw the ball around to each other; whoever has the ball is the current active object. They have to pass it to whomever they are invoking and then pass it correctly down the return path as well. This can make the exercise a little more fun and interactive.

Momentum Killers

Wednesday – It's Uphill Now

By mid-day Wednesday, the instructor starts to get tired – because you have been on constantly during this course and the students start to say "my brain is full": it is just a long week. From this point forward it is crucial for the instructor to work even harder at keeping the class energetic and interactive. Bring in different exercises, ask a lot of questions of the class, inoffensive jokes can be useful at this point, Dilbert cartoons on view-graphs help, etc.

OOAD and Rose Gaps

I made a conscious decision NOT to limit the concepts discussed in the OOAD course to those that Rose supports, so that the student's don't complain that we ONLY include modeling concepts that Rose supports. There are not that many places in the OOAD course where we discuss concepts that Rose does not support, and where they occur, the Rose workarounds are included in the instructor notes.

Some of the concepts that Rose does not support include the following:
- relationships other than dependencies to/from packages, etc.
- dependencies between use cases,

Positioning the Gaps

The following are some notes on the positioning of the gaps between the UML/RUP and Rose:
- When I am teaching to a Rose audience, I discuss the differences between the course materials/RUP and Rose in the following way:
  Rational is committed to providing a complete software development lifecycle solution to its customers, consisting of process, tools, and training. Obtaining and maintaining consistency among these solution components is a balance. There are bound to be some gaps. We are aware of those gaps and document explicit workarounds in the tool mentors. Not only do the tool mentors described how to apply the Rational process using the Rational tool set, providing workarounds as necessary, but they serve as explicit quality gates for ensuring that the two are consistent. You have to understand where the gaps are in order to know how to fix them.
- When I am teaching to a non-Rose audience (it does happen), I don't mention the Rose deficiencies at all

Additional Resources

The OOAD Instructor Certification home page on
http://midnight.rational.com/bu/rationalu/ooad/OOADInstructor/InstrCertification/instcert.html

The certified instructor's e-mail alias: cert_ooad_instr@rational.com.

OOAD Instructor Post Course Activities

Once the OOAD course has been delivered, the instructor must send all evaluation forms to the Rational University Certification Coordinator (see **Rational University Certification Coordinator Contact Information**). These evaluations are used to:
- Continually improve the quality of the course
- Verify that the instructor is delivering the course (a minimum number of courses must be taught in a year for an instructor to maintain his/her certification).
- Verify that the instructor is delivering a value-added course.

OOAD Product Manager Contact Information

The OOAD Product Manager is also interested in any additional examples, exercises, or any other material that you use or develop to support the existing course material. Over time, such information will be gathered and made available to other instructors.

Name	Lee Ackerman
E-Mail	lackerman@rational.com
Business Phone	(425) 497-6186
Business Address	Rational Software Corporation 8383 158th Avenue NE, Suite 100 Redmond, WA 98052

Certification Coordinator Contact Information

Name	Christine Sikorski
E-Mail	csikorsk@rational.com