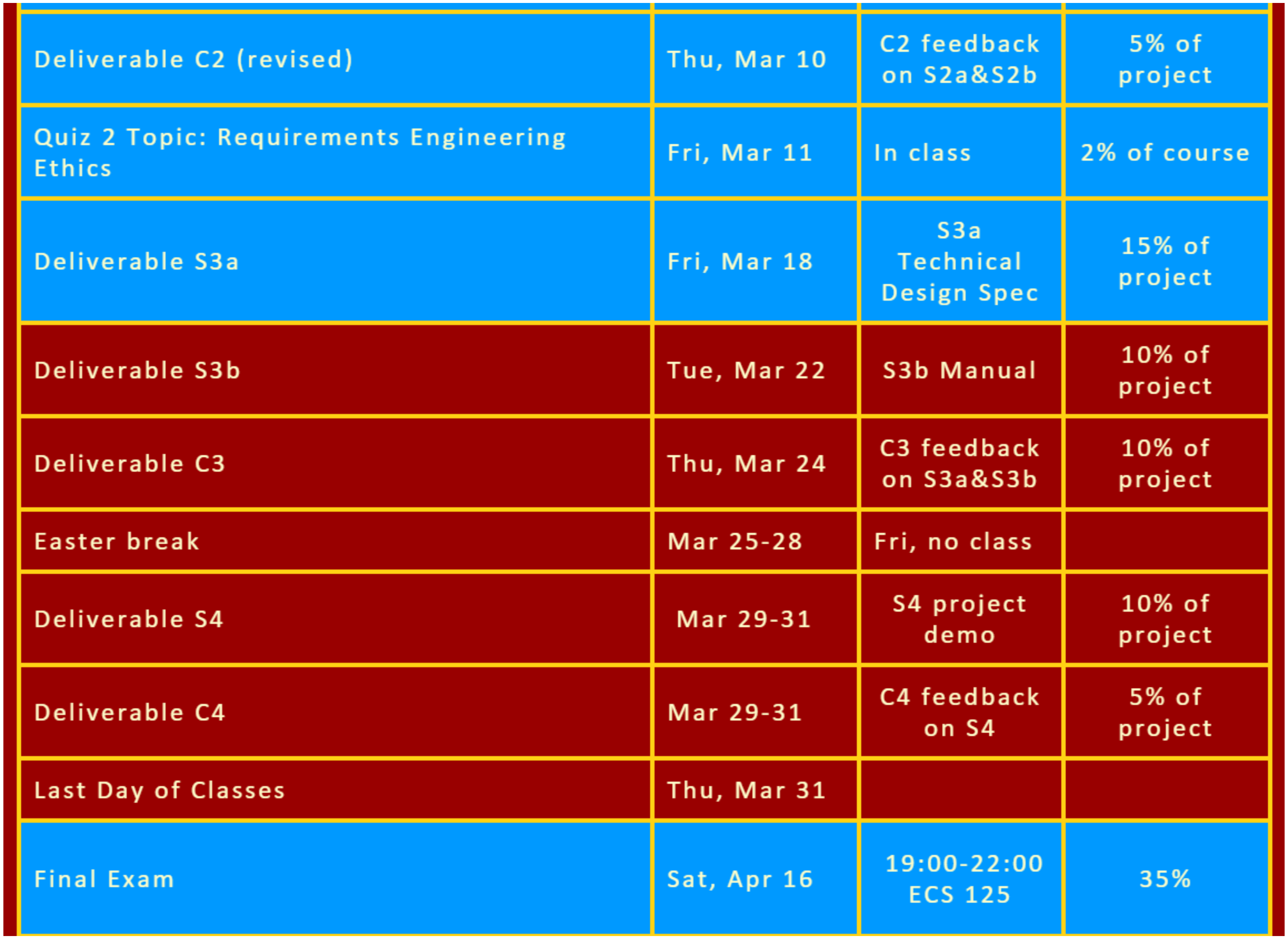
Professor Hausi A. Müller PhD PEng FCAE
Department of Computer Science
Faculty of Engineering
University of Victoria

www.engr.uvic.ca/~seng321/
courses1.csc.uvic.ca/courses/201/spring/seng/321

SENG 321 Calendar



Deliverable C2 (revised)	Thu, Mar 10	C2 feedback on S2a&S2b	5% of project
Quiz 2 Topic: Requirements Engineering Ethics	Fri, Mar 11	In class	2% of course
Deliverable S3a	Fri, Mar 18	S3a Technical Design Spec	15% of project
Deliverable S3b	Tue, Mar 22	S3b Manual	10% of project
Deliverable C3	Thu, Mar 24	C3 feedback on S3a&S3b	10% of project
Easter break	Mar 25-28	Fri, no class	
Deliverable S4	Mar 29-31	S4 project demo	10% of project
Deliverable C4	Mar 29-31	C4 feedback on S4	5% of project
Last Day of Classes	Thu, Mar 31		
Final Exam	Sat, Apr 16	19:00-22:00 ECS 125	35%

Announcements

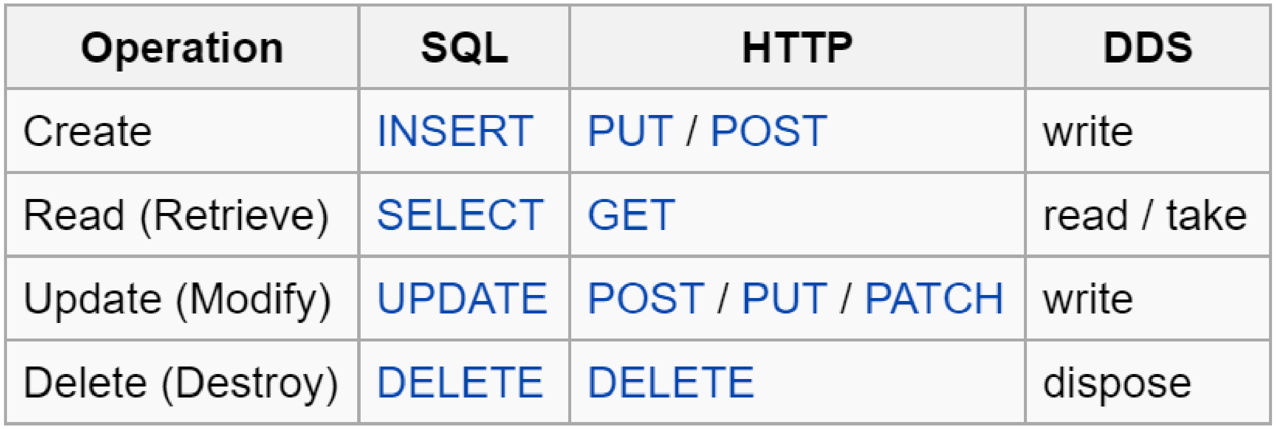
- Fri, March 18
  - S3a due
  - Detailed technical design
  - Include ethics requirements
  - NOT A MANUAL (!)

Final Exam

- Sat, April 16
- 19:00-22:00
- ECS 125

CRUD Technique

- Key data operations
  - Create
  - Read
  - Update
  - Delete



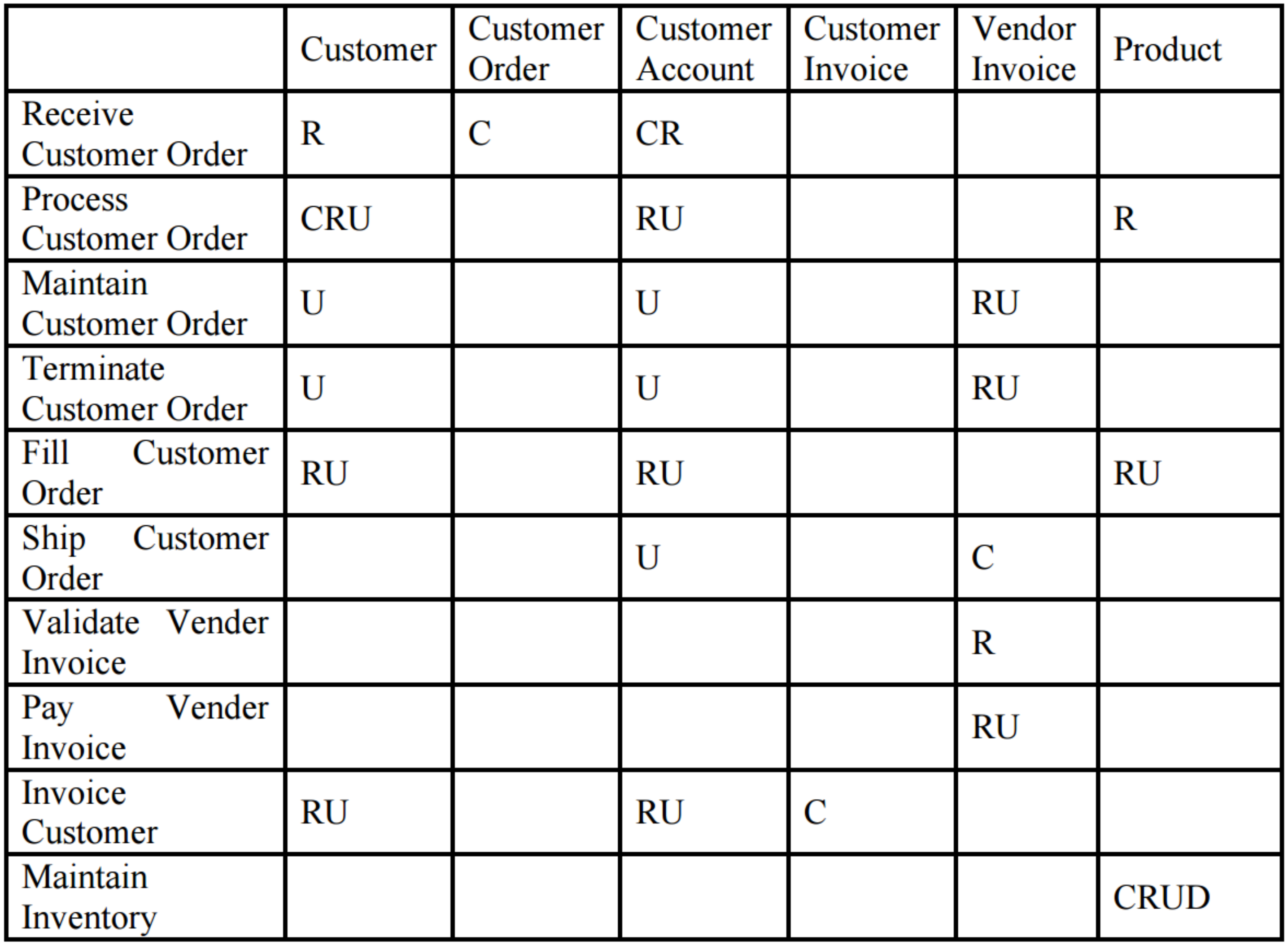
Operation	SQL	HTTP	DDS
Create	INSERT	PUT / POST	write
Read (Retrieve)	SELECT	GET	read / take
Update (Modify)	UPDATE	POST / PUT / PATCH	write
Delete (Destroy)	DELETE	DELETE	dispose

- Identify use cases that support CRUD operations
- Check data entities or domain classes
  - *Customer*, *OrderItem*

CRUD: Modeling Data and Process Interactions—E-Commerce

Entities

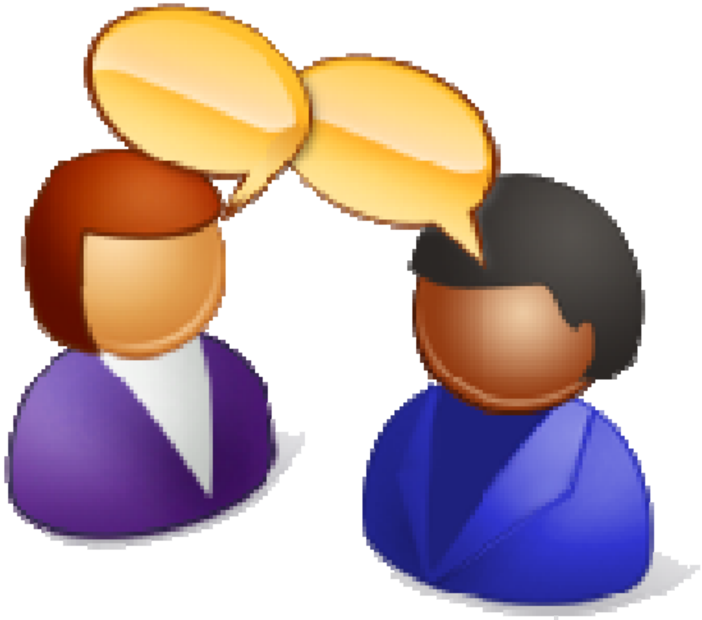
Tasks, processes



	Customer	Customer Order	Customer Account	Customer Invoice	Vendor Invoice	Product
Receive Customer Order	R	C	CR			
Process Customer Order	CRU		RU			R
Maintain Customer Order	U		U		RU	
Terminate Customer Order	U		U		RU	
Fill Customer Order	RU		RU			RU
Ship Customer Order			U		C	
Validate Vender Invoice					R	
Pay Vender Invoice					RU	
Invoice Customer	RU		RU	C		
Maintain Inventory						CRUD

CRUD Matrix—Hotel Room Booking

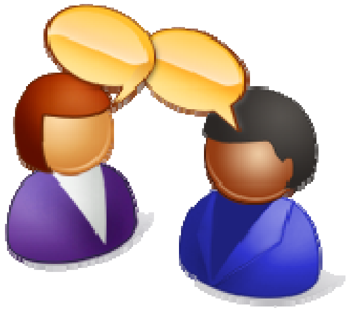
Create, Read, Update, Delete, Overview



CRUD Matrix—Hotel Room Booking

Create, Read, Update, Delete, Overview

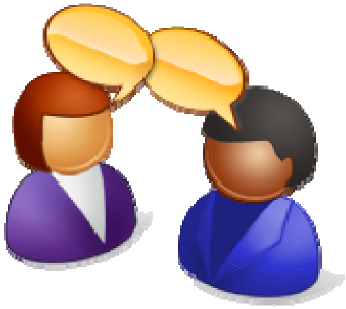
Entity Task	Guest	Stay	Room	RoomState	Service	ServiceType
Book						
CheckinBooked						
CheckinNonbkd						
Checkout						
ChangeRoom						
RecordService						
PriceChange						



CRUD Matrix—Hotel Room Booking

Create, Read, Update, Delete, Overview

Entity / Task	Guest	Stay	Room	RoomState	Service	ServiceType
Book	C U O	C	O	U O		
CheckinBooked	RU	U O	O	U O		
CheckinNonbkd	C U O	C	O	U O		
Checkout	U	U O	R	U		
ChangeRoom	R	R	O	U O		
RecordService			O		C	R
PriceChange			C UDO			C UDO

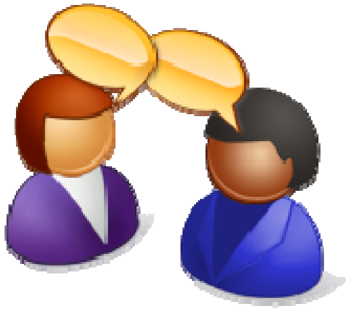


From: Soren Lauesen: Software Requirements
© Pearson / Addison-Wesley 2002

CRUD Matrix

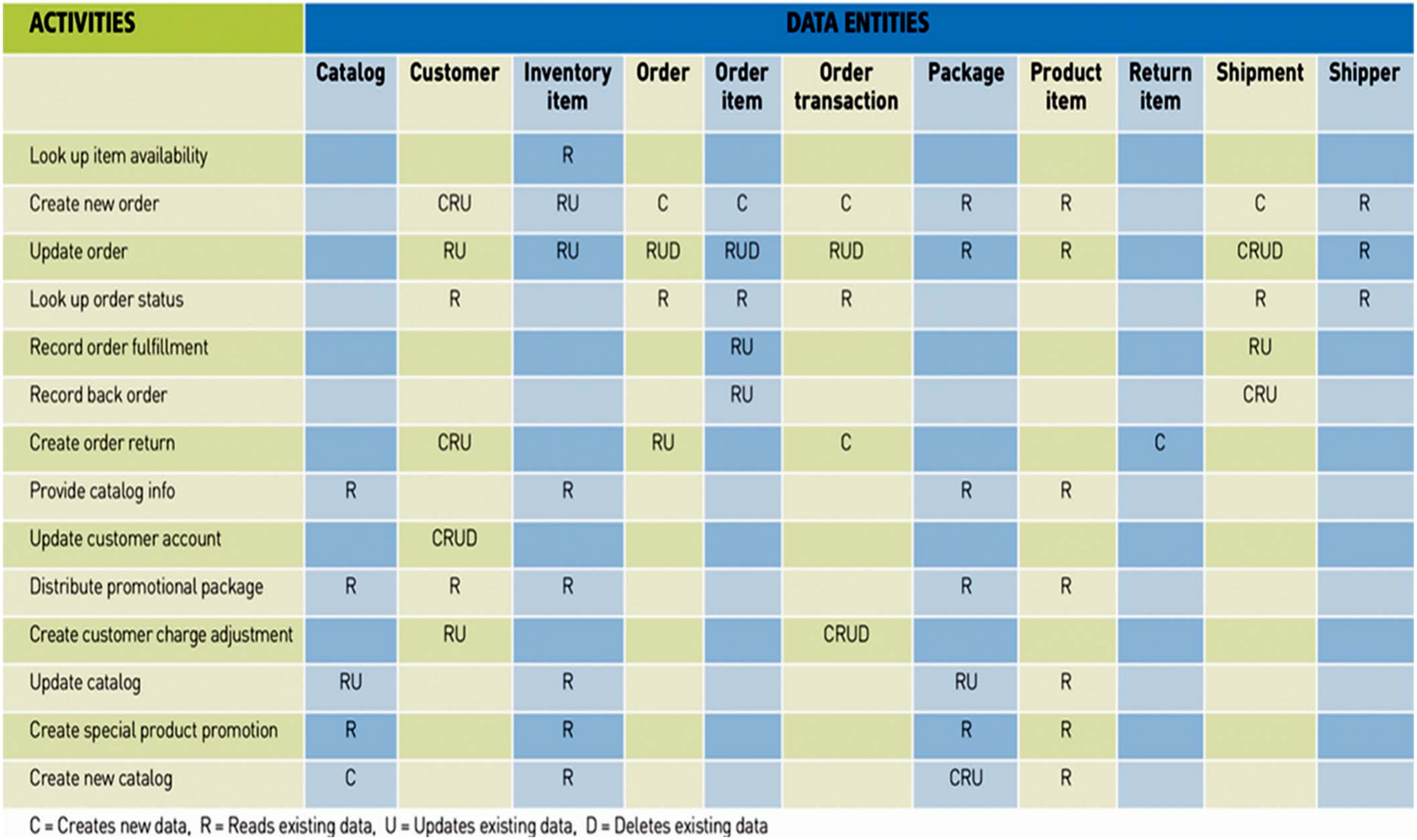
Create, Read, Update, Delete, Overview

- Develop a CRUD matrix for your project
  - Five entities
  - Five tasks
- Fill in matrix with letters
  - Create: C
  - Read: R
  - Update: U
  - Delete: D
  - Overview: O



Excellent Final Question

Another CRUD Matrix Example



ACTIVITIES	DATA ENTITIES										
	Catalog	Customer	Inventory item	Order	Order item	Order transaction	Package	Product item	Return item	Shipment	Shipper
Look up item availability			R								
Create new order		CRU	RU	C	C	C	R	R		C	R
Update order		RU	RU	RUD	RUD	RUD	R	R		CRUD	R
Look up order status		R		R	R	R				R	R
Record order fulfillment					RU					RU	
Record back order					RU					CRU	
Create order return		CRU		RU		C			C		
Provide catalog info	R		R				R	R			
Update customer account		CRUD									
Distribute promotional package	R	R	R				R	R			
Create customer charge adjustment		RU				CRUD					
Update catalog	RU		R				RU	R			
Create special product promotion	R		R				R	R			
Create new catalog	C		R				CRU	R			

C = Creates new data, R = Reads existing data, U = Updates existing data, D = Deletes existing data

Validation Techniques

- Reviews
  - Walkthroughs
  - Formal inspections
  - Focused inspections
  - Active inspections
  - Checklists
- **Testing**
- Prototyping
- Formal validation

Testing

- There are two kinds of testing that affect requirements engineering:
  1. Testing the requirements themselves, aka validation
  2. Planning for the testing of the implementation against the requirements

- Validation — Evaluate SRS wrt. customer requirements:
  - ❖ Are we building the right system?
  - ❖ Is the specification what the customer wants?
- Verification — Evaluate software artifact wrt. existing artifacts:
  - ❖ Are we building the system right?
  - ❖ For example, does the design implement the spec?

Quantifiable & Non-Quantifiable Requirements

- Quantifiable Requirements
  - R: The system must respond quickly to customer enquiries
  - Find a property that provides a scale for measurement within the context (e.g., mins)
  - Under what circumstances would the system fail to meet this requirement?
  - The stakeholders review the context: failure if a customer has to wait longer than 3 minutes for a response
  - “3 minutes" becomes the quality measure for this requirement
- Non-Quantifiable Requirements
  - R: The automated interfaces of the system must be easy to learn
  - There is no obvious measurement scale for "easy to learn“
  - Investigate the meaning of the requirement within the particular context, identify limits for measuring the requirement.
  - What is considered a failure to meet this requirement?
  - Novice users: stakeholders want novices to be productive within half an hour
  - Quality measure: a novice must be able to complete a customer order transaction within 30 mins of first using the system

S. Robertson. An Early Start to Testing:
How to Test Requirements, EuroSTAR '96

Testing Requirements

- Running an executable specification and checking certain scenarios
  - Simulating the product—good for getting customer approval
  - Type checking
  - Completeness and consistency checks
  - Best if not performed by author of specification
- The boundary between *testing a specification* and *demonstrating a prototype for customer feedback* is difficult to define

Advantages of Testing

- Low-level details checking is usually more reliable when done by tools
- Requirements testing can be done earlier in the development lifecycle than most other testing

Disadvantages of Testing

- Many notations for requirements specification are not executable or even (usefully) checkable
- It is labour intensive and costly
  - Hand holding of tools, designing of test cases
  - Automated testing can help
- No clear stopping rule
  - Law of diminishing returns definitely a factor in testing
  - 80/20 rule applies

> Testing can only be used to show the presence of errors, but not their absence.
> —Edsger Dijkstra

Test Case Planning

- The requirements specification should describe how to ascertain if the final product satisfies the requirements
  - Often called ***acceptance testing***
- It should include a complete test plan
  - An extensive collection of test cases
  - For each test case, specify the expected response of the system
- For large systems this is a separate document called a *Test Plan*

Test Case Planning

- There are two basic kinds of test cases
  1. Those generated from the specification (black box)
     - Test what the system is supposed to do according to the specification or interface, treating the implementation (at the system level) as a black box.
  2. Those generated from code / implementation (white box)
     - Design == structure, so this is testing of the representation of the system, rather than the idea of the system (the specification)
     - Metaphorically, structural testing is about looking for likely weak spots in the structure of the system, ignoring the black box semantics.
       - Do all loops terminate? Are all if conditions tested? Is there any dead code (unreachable by any execution)? …
- Obviously, at the requirements stage, the only kind that can be considered is black-box test cases, generated from the requirements.

Granularity of Tests

- When we are testing code, we start with **unit tests**, which are at the level of a class / module / file (depending on the language)
  - We try to rigorously test each method / procedure of each unit.
- You should have both black box and white box tests for each unit.
  - The black box tests are designed against the externally visible interfaces of the unit
    - For each method, think of ways of testing it using only your knowledge of what it is supposed to do, not how it is implemented.
  - The white box tests are designed against the way in which the code is written
    - For example, try to test all paths through a method, try to exercise all test conditions in **`ifs`** and **`loops`**, boundary values, *etc.*

Integration Testing

- Gradually combine the units into logical subsystems—**integration testing**
  - Do more black box testing against the interface of the whole subsystem
  - More white box testing against our understanding of how the subparts depend on and interact with each other
  - For a big system, there may be several phases of integration testing as the subparts are merged to form larger and larger subsystems

System Testing

- Black box test cases based around what the system as a whole is designed to do
  - Use the top level interface
- White box test cases designed around our understanding of the structure of the design
- It is integration testing at the top level

Testing Granularity

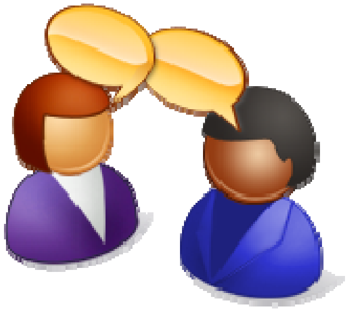
- System-level test cases are based on what the system can do, not what the customer expects.
- We design test cases around the requirements with customer input— **acceptance tests**
  - Any system that can pass the acceptance tests is capable of satisfying the customer (and the requirements model).
  - Obviously, we can use the SRS and the customer to design the acceptance tests; the other tests require design information.

Scenarios as Test Cases

- Scenarios developed for the purpose of identifying requirements are basically test cases.
- For example, a scenario gives for each user input the system's response, and lays them out in the order in which they should occur in one computation in the system.

Requirements Testing Example

- Pick a requirement (e.g., functional requirement)
- For this requirement, think of ways of testing it
  - using only your knowledge of what it is supposed to do
  - not how it is implemented



Excellent Final Question

Validation Techniques

- Reviews
  - Walkthroughs
  - Formal inspections
  - Focused inspections
  - Active inspections
  - Checklists
  - Testing
- **Prototyping**
- Formal validation

Prototyping

- The purpose of a prototype is to obtain a credible validation response.
- Prototype—a quick and dirty implementation of the most uncertain parts of the system, to demonstrate to the users how the requirements analysts understand requirements
  - User interface prototyping is very useful and effective for buy-in, fostering common understanding
  - If the specification is executable, it is a prototype
  - If not, then it is useful even to put together an application that simulates the execution of documented scenarios

Mock-up User Interfaces, Screens, and Prototypes

- Very common and useful
    - A picture is worth a thousand words
- Mock-up UIs, screens, and prototypes should not be used before a good understanding of the requirements is reached
    - Customers and users can react quite negatively to a mock-up UI
        - Convey the wrong message
        - Not esthetically pleasing
- Use task descriptions instead
    - Much more difficult to disagree with a task than with a UI mock-up
- Customer that these are just suggested screens
- Establish links between customers and prototype developers and user interface designers

Validation Techniques

- Reviews
  - Walkthroughs
  - Formal inspections
  - Focused inspections
  - Active inspections
  - Checklists
  - Testing
- Prototyping
- **Formal validation**

Formal Validation

- Ways to check if a formal specification has certain desirable properties
  - Completeness
  - Consistency
  - Mutual exclusion
  - Particular temporal properties
- Techniques
  - Model checking (for formal specification methods)
  - Theorem proving (more general for any formal spec)
  - Formal verification involves checking *all possible execution paths* of the specification