


**CSU CS 314**  
**Software Engineering Notes 5:**  
**Requirements Analysis for OO**  
**Software**



James M. Bieman



CSU CS314 Copyright © James M. Bieman 2000-2016 5-1

**What should we build?**


DILBERT BY SCOTT ADAMS

CSU CS314 Copyright © James M. Bieman 2000-2016 5-2

**Focus: Determining and Modeling Requirements**

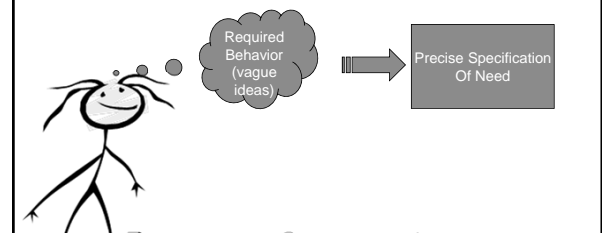

- Overview of requirements.
- Requirements documents.
- Use-case analysis.
- System operations.
- Domain/conceptual modeling.
- Evaluating requirements.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-3

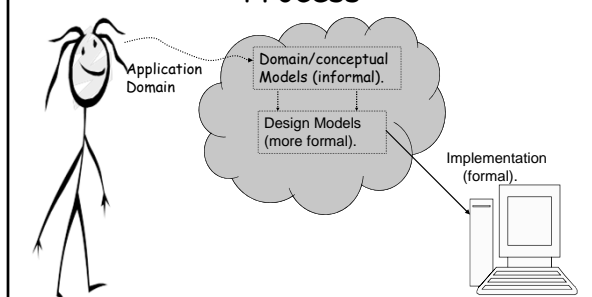

**Requirements**

We must define software needs precisely, yet most human problems are not precisely understood.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-4


**Requirements in the Software Process**

CSU CS314 Copyright © James M. Bieman 2000-2016 5-5

**Consider a Hotel Front Desk**


- Guest check-in, check-out.
- Track restaurant tabs.
- Reservations.
- Keys.
- Maintenance.
- Housekeeping.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-6

## Income Tax Software


- Tax code is not formally defined.  
Meaning is determined case by case via court decisions.
- Yet, tax software precisely implements a view of the tax code.
- Even with systems that cannot be precisely specified, requirements must be spelled out.
  - They must be precise enough to bid on contracts, estimate costs, and to start a design



CSU CS314 Copyright © James M. Bieman 2000-2016 5-7

## Requirements


- Requirement: An abstract statement of the service provided by the system and constraints on the system.
  - Does not imply a predefined solution.
  - May be the basis for a contract bid.
- Requirements specification document: a detailed formal definition of a system problem.
  - Commonly written by a contractor to show a clear understanding of the problem.
  - May serve as the basis for a contract.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-8

## Functional Requirements

- Essential behavior.
  - Services: Users can cut, paste, delete, and save files.
  - Interactions between system and external world:  
Produce paychecks every two weeks.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-9

## Non-functional Requirements

Specify operational properties & constraints.


- The system must operate on Linux.
- The code must be written in Java.
- The system must have a specified performance.
- The system must have a specified reliability.
- The system must have a specified availability.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-10

## Problem Definition

- Identifies the user's problem.
- Often vaguely written.
- States what is to be done not how it is to be solved.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-11

## Example Problem Statement

“The lawn mower evenly cuts the grass. Users can set the height of trimmed grass.”

- What is wrong with this as a specification?
- Could a mechanical engineer design a lawn mower to meet the above requirements?




CSU CS314 Copyright © James M. Bieman 2000-2016 5-12

## Requirements Specification

- A more formal document, which is understood by the developer.

“The lawn mower can be set to cut grass within a range of 1 to 3 inches in height.”

- More precise, but still is focused on the problem not the solution.
- Can serve as a basis for design.



CSU CS314
Copyright © James M. Bieman 2000-2016
5-13

## “Fun” Requirements Problems

- Air traffic control system.
- Denver International Airport baggage system.
- On-line retail store.
- Cell phone support system.
- Class scheduling & classroom reservation system.
- Election software.




CSU CS314
Copyright © James M. Bieman 2000-2016
5-14

## Requirements Issues

- Understanding difficult problems often comes only during development.  
Requirements documents are usually incomplete and inconsistent.
- New SW can change the way an organization operates, changes software needs.
- Different users may have widely varying needs.


Prototyping & Spiral Development can help.  
Agile processes may help even more.



CSU CS314
Copyright © James M. Bieman 2000-2016
5-15

## “Traditional” Requirements Document Structure

1. Introduction.
2. Glossary.
3. System models.
4. Functional requirements specification.
5. Non-functional requirements specification.
6. System evolution.
7. Appendices and Index.



CSU CS314
Copyright © James M. Bieman 2000-2016
5-16


## High Level View of Requirements Development in an Agile Process

- Develop/revise user stories given feedback from users on the running version produced in the last development cycle.
- Convert to use cases, and develop scenarios.
- Select the use cases to implement in the current cycle.
- Develop system tests for the selected use cases.
- Depending on the agile process develop/extend domain model of the system.

Non requirements activity:

- Implement selected use cases for this cycle:  
design, test, code.

Repeat this process.




CSU CS314
Copyright © James M. Bieman 2000-2016
5-17

## System Models

Model requirements, not the design.


- Relationships between components (classes), subsystems, system, & external environment.
- Use case scenarios.
- System Sequence Diagrams.
- Domain/Conceptual models.
- Other models.



CSU CS314
Copyright © James M. Bieman 2000-2016
5-18

### Requirements Document Purposes

- It is a **contract** between a client and developer.
- It is a reference tool for system development and maintenance.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-19

### Developing Requirements

- Use abstraction to separate concerns.
- Express specifications in terms of problem domain concepts.


Specifications should be tolerant of incompleteness; some requirements will be missing.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-20

### Specifications Should Be


- Easy to modify.
- Unambiguous.
- Verifiable - Testable.
- Consistent.
- Traceable.
- Usable during operation and maintenance.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-21

### Importance of Good Requirements

- Errors detected later are more expensive to fix.
- Demarco: 56% of all bugs are due to requirements errors.
- Errors: incorrect facts, omissions, inconsistencies, & ambiguities.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-22

### Object-Oriented Analysis

1. Obtain customer requirements.  
Identify Use Cases and their scenarios.
2. Develop System Sequence Model.  
Identify system operations.
3. Model the application domain concepts.
4. Check Domain Model against Use Cases.
5. Revise Domain Model, System Sequence Diagrams, Use Cases.


Repeat the process.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-23

### Use-Case Analysis


- Generalized scenario that describes system use.
- Ex: Automated retail terminal used as a store's computerized cash register.
- Retail terminal uses include:
  - Customers buy items: BuyItems Use Case.
  - Customers return items: ReturnItems Use Case.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-24

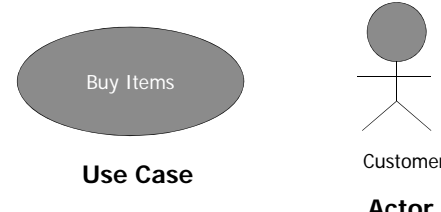
### Use Case Components

- **Actors:** external entities that interact with the system.
- **Scenarios:** describe interactions between actors and the system.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-25

### UML Icons



**Use Case**


**Actor**



CSU CS314 Copyright © James M. Bieman 2000-2016 5-26

### A Use Case Answers Questions

- What are the main tasks performed by the actor?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?




CSU CS314 Copyright © James M. Bieman 2000-2016 5-27

### More Use Case Questions

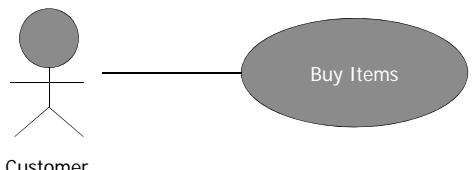
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

A use case is a narrative that describes the role of an actor & its interaction with the system.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-28

### UML Use Case Diagram




Customer



CSU CS314 Copyright © James M. Bieman 2000-2016 5-29

### Use Case Scenario Analysis


1. Identify Actors.
2. Identify and develop Scenarios.
3. Determine Use Case generalizations of scenarios.
4. Revise



CSU CS314 Copyright © James M. Bieman 2000-2016 5-30

## Identify Actors


- Identify system boundaries.
- **Actors:** people or devices that use the system.
- Actor is a **role** that a user may play.  
One user may play many roles.
- Primary actor: identified early.
- Secondary actor: identified later.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-31

## Actors May Include


- Users supported by the system to do their work.  
The store clerks who use the Retail Terminal system.
- Users that execute the main system.  
Customers who initiate the sales and returns that are recorded by the Retail Terminal system.
- Users of secondary functions.  
System maintenance or administration.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-32

## Use Case Overview


- Focus on system activities that actors take part in.
- Narrative that describes the role of an actor and its interaction with the system.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-33

## Retail Terminal Example

Use case name: BuyItems  
 Actors: Customer (initiator), Cashier.  
 Overview: A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects payment. On completion the Customer leaves with the items.  
 Type: Primary and essential.  
 Cross Reference (to other parts of the Requirements Document): Functions: R1.1,R1.2




CSU CS314 Copyright © James M. Bieman 2000-2016 5-34

## BuyItems Use Case (continued)

Typical Flow of Events.


1. Customer arrives at a Retail Terminal checkout.
2. Cashier records identifier from items.
3. System determines item price and adds item information to running sales transaction. System presents description and price of current item.
4. Cashier indicates to the Retail Terminal that item entry is complete.
5. System calculates and presents sale total.
6. Cashier tells Customer the total.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-35

## BuyItems Flow of Events (cont)


7. Customer gives payment, either cash, credit card, or check. If cash, the payment may exceed sales total.
8. Cashier records payment.
9. System shows either the cash balance due to Customer, or indicates approval of credit card or check.
10. Cashier deposits cash, check, or credit slip; gives change and receipt to Customer.
11. Customer leaves with items purchased.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-36

### Alternative Courses


- Action 2: Invalid identifier entered. System indicates error.
- Action 7: Customer did not have enough cash. Cancel sales transaction.
- Action 9: System does not approve check or credit card. System requests alternative payment.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-37

### Key Point


- A Use Case is not an individual step.
- It is an end-to-end process description, usually with many steps.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-38

### Identifying Use Cases


- Actor-based approach:
  1. Identify actors related to a system or organization.
  2. For each actor, identify the process that they initiate or participate in.
- Event-based approach:
  1. Identify external events that a system must respond to.
  2. Relate the event to actors and Use Cases.



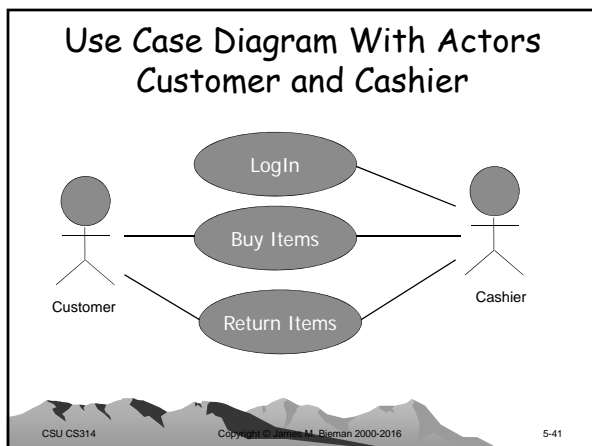
CSU CS314 Copyright © James M. Bieman 2000-2016 5-39

### Retail Terminal System Actors & Processes That They Initiate

- Cashier: LogIn, CashOut.
- Customer: BuyItems, ReturnItems.
- Manager: UpdatePrices, OrderInventory, AddCashier.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-40



### Use Cases Model Processes


- Withdraw cash from an ATM.
- Order a product.
- Register for courses at a school.
- Drop a course at a school.
- Check spelling of a document in a word processor.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-42

### Scenarios With Actor Instances


- Instances of a Use Case Scenario.
- Ex: Police Incidence Response System.
  - Use Case Scenario: ReportIncident
  - Instances:
    - WarehouseOnFire.
    - FenderBender.
    - CatInTree.
    - Earthquake.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-43

### Specify Functional Requirements from Use Cases


- Develop system behavior models.
- Convert use cases into System Sequence Diagrams.
- Serve as high-level system functional specification.
  - Describe system behavior as a black box.
  - Specify *what* a system does, *not* how it does it.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-44

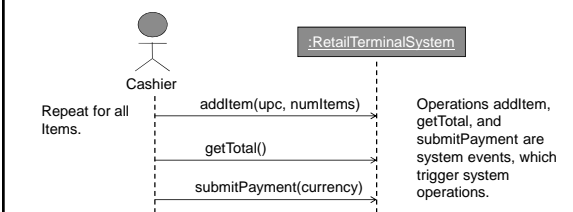
### System Sequence Diagrams

- Includes only the system operations that use case actors initiate.
- Shows for a specific use case scenario:
  - Events that external actors generate.
  - Order of the events.
- Generate system sequence diagrams for use case *typical course of events & key alternative courses.*



CSU CS314 Copyright © James M. Bieman 2000-2016 5-45

### System Sequence Diagram: BuyItems Use Case



Repeat for all Items.

Cashier


:RetailTerminalSystem

addItem(upc, numItems)

getTotal()

submitPayment(currency)

Operations addItem, getTotal, and submitPayment are system events, which trigger system operations.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-46


### Identifying System Operations

- Each system event has a corresponding operation.
- Buy Items operations:
  - addItem(UPC, numItems)
  - getTotal()
  - submitPayment(currency)

RetailTerminalSystem

addItem()  
getTotal()  
submitPayment()


System as a black-box domain concept.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-47

### System Events & Operations

- System event: external input event generated by an actor.
  - Events initiate a system operation, which responds to the event.
- System operation: executes in response to a system event.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-48




### System Sequence Diagram Recipe

Start with a use case scenario (typical course of events):

1. Draw a vertical line representing the system.
2. Draw a vertical line representing each actor that *directly* acts on the system.
3. From the use case scenario, identify each system event that the actors generates. Put them on the diagram.


There is a direct connection between the scenario and the sequence diagram.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-49

### Naming System Events & Operations


- Start event names with a verb.
- Use “level of intent” when naming events.
- Names should be meaningful at an abstract level.
  - getTotal() is better than enterKeyPressed().
  - Poor name: enterAmountTendered(amount).
  - Better name: submitPayment(amount).



CSU CS314 Copyright © James M. Bieman 2000-2016 5-50

### Why Develop System Sequence Diagrams?


- They model the external interface & behavior of the overall system.
- Result: identification of system operations.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-51

### Domain Modeling


- Devise a precise, concise, understandable, & correct model of the application domain.
- Aim: understand the problem.
- Examine & analyze requirements.
- Restate them rigorously.
- Defer dealing with small details.
- Model domain concepts using UML Class Diagrams:
  - UML Class boxes represent application domain concepts.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-52

### Domain Modeling Process

1. Identify concept “classes and objects”.
2. Identify key attributes of these concepts.
3. Identify concept responsibilities and attributes.
4. Determine associations and generalizations between concepts.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-53

### Identifying Domain Concepts

Study problem statement & use cases looking for:

- Tangible objects or devices: valve.
- Roles: Supervisor.
- People: Worker.
- Places, locations: Home, Office.
- Other systems
- Concept instances (objects) tend to be pronouns.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-54

### All Nouns May Not Represent Essential Concepts

Properties of essential concept:


- Retained information: the entity has state.
- Needed services: it will probably have some operations.
- Multiple problem-domain attributes.
- Attributes are common to all instances of a concept.
- All instances of a concept will have common operations.
- The entity is an essential requirement.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-55

### Examine Use Cases Carefully


- If 2 use cases refer to the same concept, the concept in both use cases should be the same.
- If 2 objects share the same name, but refer to different concepts, rename them.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-56

### Identify Key Attributes: Information Held By Objects


NOUN (Concept/Object)	ATTRIBUTE
Customer	name
Checkout	register ID
Cashier	badge number
Identifier	
Items	price, description
Sales Transaction	item, prices, payment, balance
Payment	amount
Balance	amount



CSU CS314 Copyright © James M. Bieman 2000-2016 5-57

### Concept (Class) Responsibilities


- Ultimately represented by methods.
- May defer method definition to design.
- Verbs in the problem statement & Use Cases are potential class responsibilities & methods.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-58

### Concept (Class) Responsibility Hints


- System intelligence should be evenly distributed.
- Responsibilities should be stated as generally as possible.
- Information & related behavior should reside in the same class.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-59

### Concept (Class) Responsibility Hints (2)


- Information about 1 thing should reside in a single concept, not distributed across multiple concepts.
- Share responsibilities among related concepts.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-60

### Deriving Associations

- Look for collaborators.
- **Association:** represents a contract between client & server concepts.
- **Relationship types:**
  - Is-part-of (composition).
  - Has-knowledge-of.
  - Depends upon, or uses.
- Delay defining generalization-specialization relationships.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-61

### Ex: Pizza Ordering System

**System Capabilities:**


- It should be accessed via the www. It should allow a customer to order pizza to be delivered to the customer's location.
- Customers can specify the pizza size, toppings, delivery address, and method of payment.
- Customer orders will be recorded and placed where the restaurant staff can find and process the order.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-62

### Primary Use Case


**Use case name:** OrderPizza  
**Actors:** Customer (initiator), KitchenStaff, Driver.  
**Overview:** A Customer uses a browser to order a pizza for delivery.  
**Type:** Primary and essential.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-63

### Typical Flow of Events


1. Customer finds Pizza Web Page using a browser.
2. Customer selects PizzaOrder.
3. System brings up an order form.
4. Customer selects pizza size and toppings. Gives address, phone, and payment method.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-64

### Typical Flow of Events


5. System records customer's information, calculates the price, and then presents confirmation window.
6. Customer confirms the order.
7. System sends order to the kitchen and delivery staff.
8. Kitchen staff makes pizza.
9. Driver delivers it to the Customer.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-65

### Alternative Courses

- **Action 5:** System finds an error in the order and brings up an error window.
- **Action 6:** Customer does not confirm. The System gives the customer a choice: correct error or quit.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-66

### Examine Nouns In Problem Statement

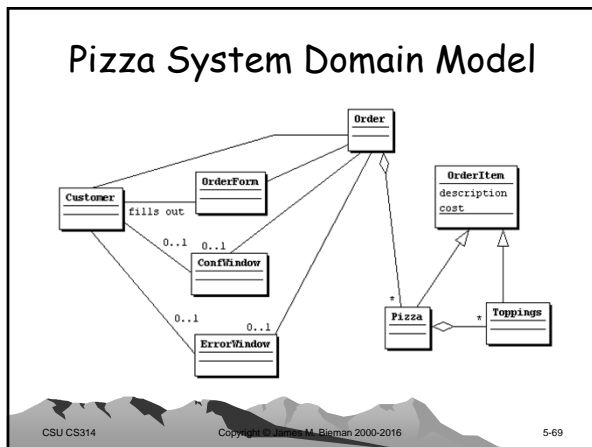
- Customer
- Order
- Kitchen Staff
- Driver
- Pizza Web Page
- Order Form
- Pizza
- Toppings
- Confirmation Window
- Error Window

CSU CS314 Copyright © James M. Bieman 2000-2016 5-67

### Concepts In My Model

- I omit the driver & kitchen staff.
- Use *OrderForm* as the entry into the system (I won't include the main web page).
- Included are *Customer*, *Order*, *Order Form*, *Pizza*, *Toppings*, *ConfirmationWindow*, and *ErrorWindow*.
- Additional classes can be added later.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-68



### Potential Additions

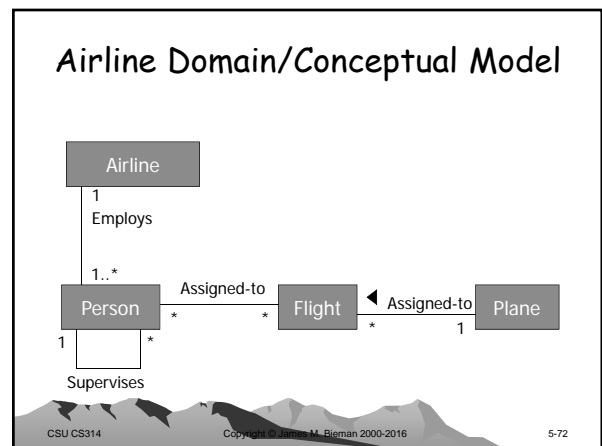
- **Attributes:**
  - Customer: name, address, phone number, lastOrder.
  - Order: timePlaced, cost, paymentType.
- **Methods:**
  - Order concept would have certain methods: calculateCost(), recordOrder(), validateOrder()
  - Some experts recommend against including methods.

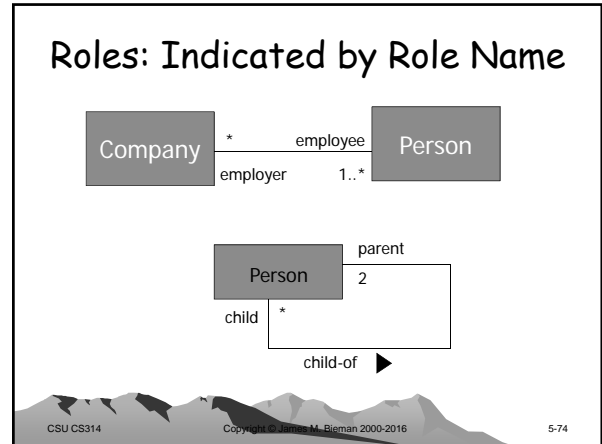
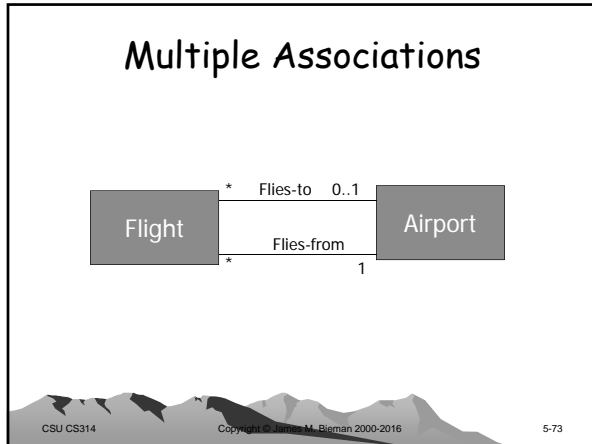
CSU CS314 Copyright © James M. Bieman 2000-2016 5-70

### Determining Relationships Between Concepts

1. Identify concept responsibilities.
2. Identify collaborator concepts that help a concept meet defined responsibilities.
  - This establishes a connection between concepts.
  - Collaborators are always related in some way.
3. Specify nature of the relationship.
  - Direction.
  - Arity: one-to-one, one-to-two, one-to-many.
  - Classify: aggregation, inheritance, association.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-71





### Requirements Evaluation

A good requirements document should be:

- Unambiguous.
- Verifiable.
- Consistent.
- Complete.
- Should not dictate a particular design.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-75

### Unambiguous Requirement Specification Document

- Each statement has one interpretation.
- Use a glossary.
- Ambiguous terms: “user”, “user account”, “transaction”.
- Use terms from application domain.
- Avoid vague terms: “some”, “sometimes”, “often”, “usually”, “ordinarily”.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-76

### Avoid Ambiguous Pronouns

“Module A talks to Module B and its control flag is set.”

Which module does “its” refer to?

CSU CS314 Copyright © James M. Bieman 2000-2016 5-77


### Verifiable Requirement

- Can be tested.  
Must be finite, cost-effective ways to check a program to see if it meets its requirements.
- Non-verifiable statements:
  - “should be reliable”,
  - have a “friendly user-interface”,
  - or “responses are usually within 10 seconds”

CSU CS314 Copyright © James M. Bieman 2000-2016 5-78

### Consistent Requirements

- No two statements contradict.
- Avoid using different terms for the same concept.
- Conflicting real world properties:  
"stop at red only ... stop at green only"
- Logical or temporal statements may conflict:  
"A follows B ... A acts simultaneously with B"




CSU CS314 Copyright © James M. Bieman 2000-2016 5-79

### Complete Requirements

Adequately describes all significant requirements.


- Define all responses for all inputs in all situations.
- Label all requirements, tables, figures, & diagrams.
- Reference these labels.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-80

### Requirements Should Not Imply a Design


- Entities, models, and descriptions should be in the language of the clients problem domain.
- Avoid:
  - Data structures: "arrays, stacks, queues, trees ..."
  - "Procedure, threads, tasks, remote procedure calls, synchronization" are design entities.
  - "pixels" (unless it's part of the problem domain").
  - Algorithm details.
  - Design patterns.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-81

### Evaluating Use Cases


- Should cover all types of **end-to-end** scenarios.
- A use case is not an individual operation.
  - UpdateInventory is probably not a use case in a video store system. It's an operation that is part of use cases such as RentVideo, ReturnVideo, etc.
  - Operations such as update account balance, calculate late fees, give cash for purchases, and use a coupon.  
These are not use cases. Rather they are part of use cases.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-82

### Evaluating Domain/Conceptual Models


- The information needed by a concept should be modeled either as an attribute or an association (possibly composition).
- Make sure that you specify the multiplicity when an association may be to more than one conceptual object.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-83

### Evaluating Domain Models


- A verb should not be the primary word used to describe a concept.
  - ``tracks the instances of items'' does not sound like the description of a concept.
  - The action is likely to be a service provided by a concept.
- Make sure that you have associations to indicate a mechanism for a conceptual object to obtain the necessary information from other concepts in order to fulfill its responsibilities.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-84

### Functional vs. Non-functional Requirements


- *Functional requirements* describe what we want the system to do.  
For the video store, “record video rentals and returns” is a functional requirement.
- *Non-functional requirement* defines required or necessary constraints.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-85

### Non-Functional Requirements


- Response time.
- Implementation language.
- OS platform.
- Portability.
- Ease of use.
- Reliability.
- Maintainability.
- Testability.
- Robustness.
- Availability.
- Safety.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-86

### Non-Functional Requirements

- Must be specified in a testable manner.  
Quantify them:
  - Give numerical values for response times.
  - Be specific about portability requirements.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-87

### Requirements: Functional & Non-functional.


- Functional requirements tell what we want a system to do.
- Non-functional requirements often state what we don't want the system to do.  
Safety requirements state that the system should “do no harm”.  
A good safety requirement must be testable.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-88

### Summary

- Requirements describe what a system is to do, not how to do it.
- Requirements documents must be “rigorous” descriptions of the problem.
- We model the problem space via use-case analysis, system sequence diagrams, and domain/conceptual models.
- Requirements must exhibit a set of desirable properties.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-89

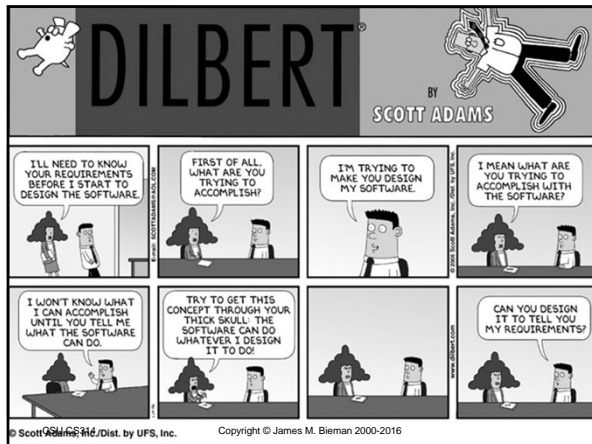
### Requirements Notes Addendum

- More on Domain Modeling.
- Nonfunctional requirements.
- The trouble with natural language specifications.
- How specifications live forever.

First, a word from Dilbert.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-90



### Domain Modeling with UML Class Diagrams CRC cards

**What is a Domain Model?**

- Illustrates meaningful conceptual classes in the problem domain.
- Represents real-world concepts, not software components.
- Software-oriented class diagrams are developed later, during design.

Source: Prof. Glenn Blank, Lehigh Univ.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-92

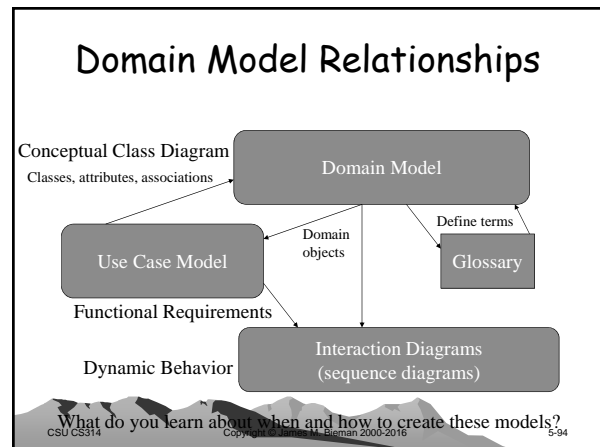
### A Domain Model is Conceptual, not a Software Artifact

**Conceptual Class:**      **Software Artifacts:**

Sale	vs.	SalesDatabase
amt		Sale
item		Double amt;
		Item item;
		void print()

What's the difference?

CSU CS314 Copyright © James M. Bieman 2000-2016 5-93



### Identify conceptual classes from noun phrases

Analyze Problem Statement, Use Cases and Use Case Scenarios, and Glossary.

However:

- Words may be ambiguous or synonymous.
- Noun phrases may also be attributes or parameters rather than classes:
  - If it stores state information or it has multiple behaviors, then it's a class.
  - If it's just a number or a string, then it's probably an attribute.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-95

### From Noun Phrases (NPs) to classes or attributes

Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.  
 If it is, then the customer can check the account balance, deposit cash, and withdraw cash.  
 Checking the balance simply displays the account balance.  
 Depositing asks the customer to enter the amount, then updates the account balance.  
 Withdraw cash asks the customer for an amount to withdraw; if the account has enough cash, the account balance is updated. The ATM prints the customer's account balance on a receipt.

Analyze each **subject** and **object** as follows:

- Does it represent a person performing an action? It's an actor, 'R'.
- Is it also a verb (such as 'deposit')? It may be a method, 'M'.
- Is it a simple value, such as 'color' (string) or 'money' (number)? It is probably an attribute, 'A'.
- Which NPs are unmarked? Make it 'C' for class.

Verbs can also be classes, for example:

- Deposit is a class if it retains state information

CSU CS314 Copyright © James M. Bieman 2000-2016 5-96



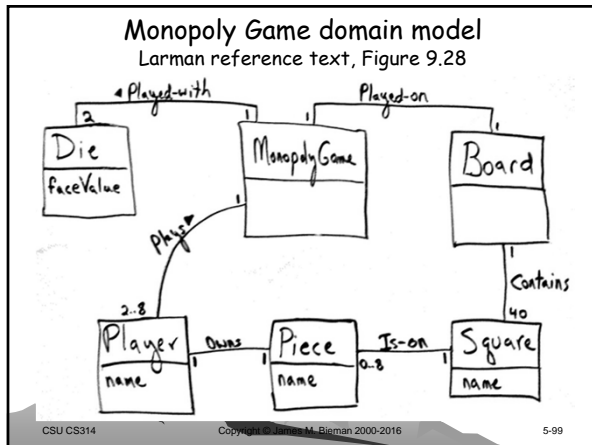
### Steps to create a Domain Model

- Identify candidate conceptual classes
- Draw them in a UML domain model
- Add associations necessary to record the relationships that must be retained
- Add attributes necessary for information to be preserved
- Use existing names for things, the vocabulary of the domain

CSU CS314 Copyright © James M. Bieman 2000-2016 5-97

### Monopoly Game domain model (first identify concepts as classes)

CSU CS314 Copyright © James M. Bieman 2000-2016 5-98



### Discovering the Domain Model with CRC cards

- Developed by Beck and Cunningham at Tektronix
  - See <http://c2.com/doc/oopsla89/paper.html>
  - This is the same Kent Beck that later wrote the book pioneering Extreme Programming (XP)
- CRC cards are now part of XP

Class Name	
Responsibilities	Collaborators
..	..

CSU CS314 Copyright © James M. Bieman 2000-2016 5-100

### Low-tech

- Ordinary index cards
  - Each card represents a **class** of objects.
  - 3x5 is preferable to 4x6 at least early on - Why?
- Each card has three components
  - Name, Responsibilities, Collaborators

Class Name	
Responsibilities	Collaborators
..	..

CSU CS314 Copyright © James M. Bieman 2000-2016 5-101

### Responsibilities

- Key idea: objects have responsibilities. As if they were simple agents (or actors in scenarios).
- Anthropomorphism of class responsibilities avoids thinking of classes as just data holders. "Object think" focuses on their active behaviors.
- Each object is responsible for specific actions. Client can expect predictable behaviors.
- Responsibility also implies independence:
  - Trust an object to behave as expected and rely upon its autonomy and modularity.
  - It's hard to trust objects caught up in dependencies due to global variables and side effects.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-102

### Class Names

- **Class Name** creates the vocabulary of our analysis.
  - Use nouns as class names; think of them as simple agents.
  - Verbs can be made into nouns, if they are maintain state. "reads card" suggests **CardReader**.
- Use pronounceable names:
  - If you cannot read the name out loud, it is not a good name.
- Capitalization the first letter in Class names and use CamelBack for multi-word names:
  - CardReader rather than CARDREADER or card\_reader
- Avoid obscure, ambiguous abbreviations:
  - Is "TermProcess" something that terminates or something that runs on a terminal?
- Avoid using digits within a name (e.g., CardReader2).
  - Its better for instances than classes.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-103

### Responsibilities section

- Describes a class's **behaviors**.
  - Describe *what* is to be done, not *how*.
- Use short verb phrases:
  - "reads card" or "look up words".
- The small size of index cards guides class analysis.
  - Limits the complexity of card entries.
  - If you cannot fit enough tasks on a card, maybe you need to divide tasks between classes, on different cards.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-104

### Collaborators

- Lists important **suppliers** and possibly clients of a class.
- Classes that supply services are important here.
  - Suppliers are needed to describe responsibilities.
- As you write down responsibilities for a class, add any suppliers needed.
  - For example "read dictionary" obviously implies a "dictionary" as a collaborator.
- Developing CRC cards is a process of discovering classes and their responsibilities.
  - People naturally perceive the world as categories of objects.
  - During OO analysis, you discover new categories relevant to a problem domain.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-105

### Non-functional Requirements

Define system properties and constraints:

- Properties: reliability, response time and storage requirements.
- Constraints: I/O device capability, system representations, etc.

- Process requirements may mandate a particular program development environment, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless, or worse.

Source: UCSD.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-106

### Non-functional Classifications

- **Product requirements:** the delivered product must behave in a particular way.
  - Execution speed, reliability, etc.
- **Organisational requirements:** a consequence of organisational policies and procedures.
  - Process standards, implementation requirements, etc.
- **External requirements:** factors that are external to the system and its development process.
  - Interoperability requirements, legislative requirements, etc.

CSU CS314 Copyright © James M. Bieman 2000-2016 5-107

### Non-functional requirement types


```

graph TD
    NFR[Non-functional requirements] --> PR[Product requirements]
    NFR --> OR[Organizational requirements]
    NFR --> ER[External requirements]
    PR --> ER1[Efficiency requirements]
    PR --> RR[Reliability requirements]
    PR --> PR1[Portability requirements]
    OR --> IIR[Interoperability requirements]
    OR --> ER2[Ethical requirements]
    ER --> UR[Usability requirements]
    ER --> DR[Delivery requirements]
    ER --> IR[Implementation requirements]
    ER --> SR[Standards requirements]
    ER --> LR[Legislative requirements]
    UR --> PR2[Performance requirements]
    UR --> SR1[Space requirements]
    IIR --> PR3[Privacy requirements]
    IIR --> SR2[Safety requirements]
    
```

CSU CS314 Copyright © James M. Bieman 2000-2016 5-108

### Non-functional requirements examples


- Product requirement  
4.C.8 It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set.
- Organisational requirement  
9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95
- External requirement  
7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system



CSU CS314 Copyright © James M. Bieman 2000-2016 5-109

### Goals and requirements


- Non-functional requirements can be very difficult to state precisely, but imprecise requirements may be difficult to verify.
- Goal: A general intention of the user.  
Ex: ease of use
- Verifiable non-functional requirement  
A statement using some measure that can be objectively tested
- Goals are helpful to developers as they convey the intentions of the system users



CSU CS314 Copyright © James M. Bieman 2000-2016 5-110

### Examples


- A system goal:  
The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.
- A verifiable non-functional requirement:  
Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-111

### Non-functional Requirements Measures


- Speed:
  - Processed transactions per second.
  - User event response time.
  - Screen refresh time.
- Size: Kbytes, number of RAM chips
- Ease of use: training time, number of help frames.
- Reliability: mean time to failure, rate of failure occurrence.
- Availability: proportion of time that the system is available.
- Robustness:
  - Time to restart after failure.
  - Percentage of events causing failure.
  - Probability of data corruption on failure.
- Portability:
  - Percentage of statements that are platform dependent.
  - Number of target platforms.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-112

### Requirements Interaction

- Conflicts between different non-functional requirements are common in complex systems.
- Spacecraft system:
  - To minimise weight, the number of separate chips in the system should be minimised.
  - To minimise power consumption, lower power chips should be used.
  - However, using low power chips may mean that more chips have to be used.




Which is the most critical requirement?



CSU CS314 Copyright © James M. Bieman 2000-2016 5-113

### Domain requirements


- Based on the application domain: system characteristics and features that reflect the domain.  
May include new functional requirements, constraints on existing requirements or define specific computations
- If domain requirements are not satisfied, the system may be unusable.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-114

### Domain requirements problems


- Understandability:
  - Expressed in the language of the application domain.
  - Often not understood by the software developers.
- Implicitness:
  - Domain specialists understand the area so well that they do not think of making the domain requirements explicit.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-115

### User requirements


- Should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge.
- User requirements are defined using natural language, tables and diagrams.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-116

### Problems with natural language

- Lack of clarity.
  - Precision is difficult without making the document difficult to read.
- Requirements confusion.
  - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
  - Several different requirements may be expressed together.




CSU CS314 Copyright © James M. Bieman 2000-2016 5-117

### Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.


**Avoid the use of computer jargon !!!**



CSU CS314 Copyright © James M. Bieman 2000-2016 5-118

### Requirements and design


- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable.
  - A system architecture may be designed to structure the requirements.
  - The system may inter-operate with other systems that generate design requirements.
  - The use of a specific design may be a domain requirement.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-119

### Problems with Natural Language (NL) specification

- Ambiguity.
  - The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult
- Over-flexibility.
  - The same thing may be said in a number of different ways in the specification
- Lack of modularization.
  - NL structures are inadequate to structure system requirements.



CSU CS314 Copyright © James M. Bieman 2000-2016 5-120

## The Trouble with Natural Language Specifications

A case-study on a text-processing problem.

Naur's Specification: Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to a line-by-line form in accordance with the following rules:

1. line breaks must be made only where the given text has BLANK or NL;
2. each line is filled as far as possible, as long as
3. no line will contain more than MAXPOS characters.

CSU CS314

Copyright © James M. Bieman 2000-2016

5-121

## Specifying a Text-processing Problem.

Goodenough and Gerhart's Specification:

The program's input is a stream of characters whose end is signaled with a special end-of-text character, ET. There is exactly one ET character in each input stream. Characters are classified as:

- break characters - BL (blank) and NL (new line);
- nonbreak characters - all others except ET;
- the end-of-text indicator - ET.

CSU CS314

Copyright © James M. Bieman 2000-2016

5-122

## Goodenough and Gerhart's Specification (cont)

A word is a nonempty sequence of nonbreak characters. A break is a sequence of one or more break characters. Thus, the input can be viewed as a sequence of words separated by breaks, with possibly leading and trailing blanks, and ending with ET.

The program's output should be the same sequence of words as in the input, with the exception that an oversize word (i.e. a word containing more than MAXPOS characters, where MAXPOS is a positive integer) should cause an error exit from the program (i.e. a variable, Alarm, should have the value TRUE). Up to the point of error, the program's output should have the following properties:

1. A new line should start only between words and at the beginning of the output text, if any.
2. A break in the input is reduced to a single break character in the output.
3. As many words as possible should be placed on each line (i.e. between successive NL characters).
4. No line may contain more than MAXPOS characters (words and BLs).

CSU CS314

Copyright © James M. Bieman 2000-2016

5-123

## Mili's Specification

Given a non-negative integer MAXPOS and a character set which includes two break characters, blank and newline. For a sequence of character(s) s, we define a word as a non-empty sequence of consecutive non-break characters embedded between break characters or the endpoints of s (i.e. the beginning and end of sequence s).

The program shall accept as input a finite sequence of characters and produce as output a sequence of characters satisfying the following conditions:

1. If the input sequence contains MAXPOS+1 consecutive non-break characters then the output sequence consists of a blank.
2. If the input includes at least one break for any consecutive MAXPOS+1 characters, then
  1. All the words of the input appear in the output, in the same order; all the words of the output appear in the input.
  2. Furthermore, the output must meet the following conditions:
    1. It contains no leading or trailing breaks, nor does it have two consecutive breaks.
    2. Any sequence of MAXPOS+1 consecutive characters includes a newline.
    3. Any subsequence of the output made up of no more than MAXPOS consecutive characters and embedded between the head of the output or a newline on the left and the tail of the output or a break on the right does not contain a newline character.

CSU CS314

Copyright © James M. Bieman 2000-2016

5-124

## How Specs Live Forever

The Question:

The US Standard railroad gauge (distance between the rails) is 4 feet, 8.5 inches.

That's an exceedingly odd number. Why was that gauge used?

- Because that's the way they built them in England, and the US railroads were built by English expatriates.
  - Why did the English people build them like that?
- Because the first rail lines were built by the same people who built the pre-railroad tramways, and that's the gauge they used.
  - Why did "they" use that gauge then?
- Because the people who built the tramways used the same jigs and tools that they used for building wagons, which used that wheel spacing.

CSU CS314

Copyright © James M. Bieman 2000-2016

5-125

## Why did the wagons use that odd wheel spacing?

- Well, if they tried to use any other spacing the wagons would break on some of the old, long distance roads, because that's the spacing of the old wheel ruts

So who built these old rutted roads?

- The first long distance roads in Europe were built by Imperial Rome for the benefit of their legions. The roads have been used ever since. And the ruts? The initial ruts, which everyone else had to match for fear of destroying their wagons, were first made by Roman war chariots. Since the chariots were made for or by Imperial Rome they were all alike in the matter of wheel spacing.

CSU CS314

Copyright © James M. Bieman 2000-2016

5-126

## How Specs Live Forever

### The Answer:

Thus, we have the answer to the original questions. The United States' standard railroad gauge of 4 feet, 8.5 inches derives from the original specification for an Imperial Roman army war chariot.

Specs and Bureaucracies live forever.

So, the next time you are handed a specification and wonder what horse's ass came up with it, you may be exactly right. Because the Imperial Roman chariots were made to be just wide enough to accommodate the back-ends of two war horses.

Numerous sources: Search on "How specs live forever".

