

CSCI0320

Introduction to Software Engineering

Lecture 3

UML

Mechanics

- **GALAXY handout updated**
- **Hand in first galaxy design**
 - `/course/cs032/bin/cs032_handin_galaxydesign1`
 - In a directory with a README file
- **Labs**
 - Hand in lab1eclipse
 - Hand in lab1junit
 - Hand in lab2swing

Descriptions of Software

- **Required for software engineering**
 - **Requirements:** what is needed
 - **Specifications:** what should be built
 - **Design:** how the system should be built
 - **Coding:** what the code looks like
- **Purpose**
 - Understanding
 - Explaining to others
 - Providing a basis for implementation

Description Formats

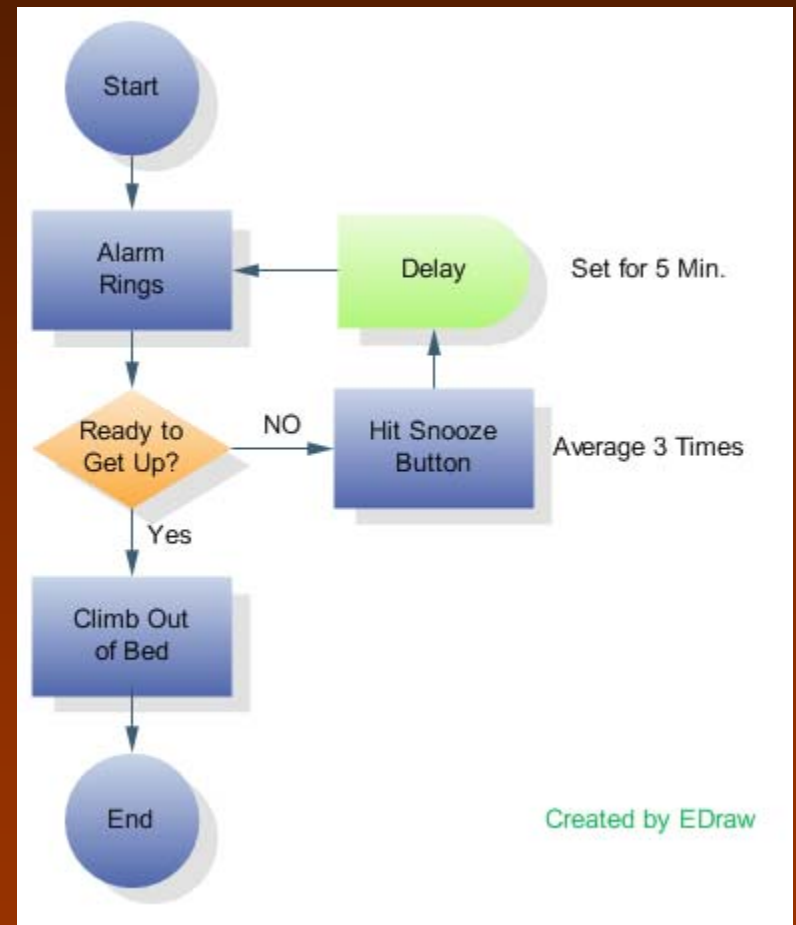
- **Natural Language**
 - **Pros:** easy to write, lots of tools
 - **Cons:**
 - Not always that descriptive
 - Not always easy to understand
 - Not easy to check completeness, consistency
- **Formal Methods (Mathematics)**
 - **Pros:** checkable, precise
 - **Cons:**
 - Only applicable at some levels
 - Hard to understand; harder to write
 - **Partial formal specifications are useful**

Software Diagrams

- A **picture** is worth a thousand words
- **Pros**
 - Easier to understand
 - Can be compact and concise
 - Provides good overview + details
 - Good for expressing relationships
- **Cons**
 - Needs textual annotations
 - Ambiguous without a common meaning

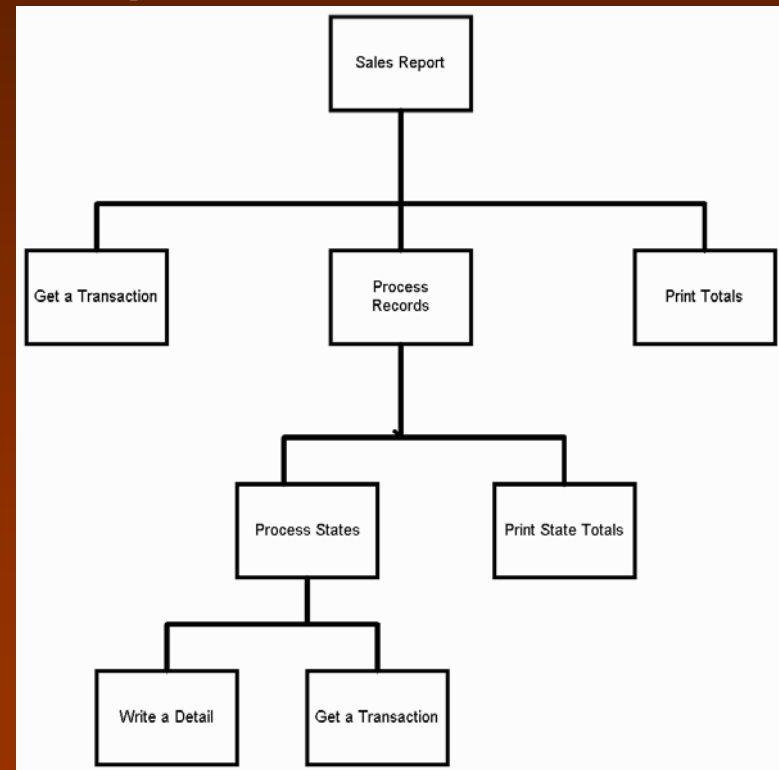
The Meaning of a Picture

- **Flow charts**
 - Different node types
 - Start/End
 - Action
 - Conditional
 - Standardized



The Meaning of a Picture

- **Module diagrams**
 - Structured programming
 - Relatively standard
 - And simple



Object-Oriented Diagrams

- **Lots of early representations**
 - **OMT: Rumbaugh**
 - **Booch diagrams**
 - **OOSE: Jacobson**
 - **Object life cycles**
 - **Odell classification**
 - **Fusion**
 - **Wirfs-Brock**
 - **Statecharts: Harel**
- **BABEL**
 - **Couldn't understand each others pictures**

UML

- Evolved from these early representations
 - Goal is to have a common form
 - With well understood syntax & semantics
 - Anyone can understand the diagrams
 - **Union** (oops)
- Concept of **modeling**
 - Build a model of the system
 - Requirements, specifications, design, code
 - Diagrams show that model
- UML
 - Implies a consistent underlying **model**
 - Diagrams are **views** of that model

UML Advantages

- **Common, well-understood** representation
- **Multiple consistent** views of one model
- **Comprehensive**
 - Covers requirements, specifications & design
 - Covers high-level and detailed design
- **Easily maintainable documentation**
 - Can be kept up to date
- **Can be used to generate code**
 - Model driver development
- **Can be generated from code**
 - Round-trip engineering

UML Disadvantages

- **The semantics are sometimes ambiguous**
- **Data flow diagrams are missing**
- **Evolves separately from code**
- **While top-level pictures are easy to edit,**
 - **Details are hard to enter or display**
- **Diagrams can be complex**
 - **Nuances can be meaningful**
 - **Nuances not always picked up by reader**
 - **Not always created correctly**
- **Tools are buggy**

Practical UML

- Use for **Understanding** a software system
 - Diagrams for requirements
 - Top-level design diagrams
 - Diagrams showing possible interactions
- Good for **documenting** a design
 - Should be kept up-to-date
 - Round trip engineering can do this
 - But loses all the formatting and layout
 - Less useful in the long-run
 - Good for the initial design

Requirements

- **Purpose**
 - Provide an understanding of users' needs
 - How the system will be used
 - What the system has to do for users
- **Techniques**
 - **Scenarios** (Agile programming)
 - Single use of the system
 - Single feature or complete run
 - **Story boards**
 - Sequence of sketches of the system in action
 - **UML**: use case diagrams

Problem to Consider

- **Consider a bibliography program. The program should maintain a bibliography of papers. It should let the user add citations to papers in the bibliography from the word processor. It should manage in-line references and the bibliography section of the paper. Bibliographical references should be easy to define.**

ARGOUML: Use Cases

- **Run ARGO**
- **Components**
 - **Actors**
 - **Human actors: users of the system**
 - **Nonhuman actors: other users**
 - External systems, external devices
 - **Use Cases (actions)**
 - **What the actor does**
 - **Actions the system needs to perform**
 - **Relationships**
 - **Between actors and use cases**
 - **Between use cases**
- **Get description of use of problem and attempt to cast as use case**
 - **Ensure you add text**

UML-based Design

- **Best use of UML**
- **Initial design**
 - **Finding** candidate classes
 - **Organizing** those classes
- **Class design**
 - **Denoting** class methods and fields
 - **Defining** interfaces
- **UML class diagrams are good for these**

ARGOUML: Class Diagrams

- **Run ARGO**
- **List potential classes for problem**
 - Just drop them on the display
- **Organizing the classes**
 - Do groupings
 - Show hierarchies
- **Describing the classes**
 - Show relationships (containment)
 - Show dependencies (calls or uses)
 - Define operations (methods)
 - Define attributes (fields)

UML-Based Detailed Design

- Can define all the fields and methods
 - Define enough to understand the design
 - Show the classes and how they interact
 - Provide an overview of the system
 - Probably not worth doing detail
 - **Interfaces** may be better at some point
 - But they don't show the class structure
 - And they don't give the overview
- Can define **how** things work
 - Good when behavior is complex
 - Documents the design
 - Before attempting the implementation

State Diagrams

- **Finite state automata**
 - Editor provide to make drawing easier
 - Good for showing overall system behavior
 - Good for showing object behavior
 - Good for showing method actions
- **Actually these are state charts**
 - Multiple parallel automata
 - Why might you want this?
 - Synchronization and events
- **Provided by ARGUML**

Sequence Diagrams

- Show a **sequence** of method calls
 - Among a set of objects
 - Illustrate how a complex action works
- Can be generalized for other examples
 - Where time and interaction is important
 - **Message-based interfaces** (games)
- Provided for by **ARGOUML**

Contracts

- Detailed design: **interfaces** and **classes**
 - Providing methods
 - Name, parameters
 - **Also need to describe what it does**
 - How to describe what a method does
 - **English** (not precise, very general)
 - **Contracts** (precise, not that general)
- What is a **contract**
 - What the caller guarantees on input
 - What the callee guarantees on output

Contract Components

- **Preconditions**
 - Statements about what the caller guarantees
 - **Argument must not be null; List must be sorted**
 - **Integer must be 0..4**
 - Defensive programming
- **Postconditions**
 - Statements about what is true on return
 - **The result is the distance between input objects**
 - **A new list is returned; Output not null**
- **Class conditions**
 - Invariants about a class
 - **X contains the set of all active objects**
 - **True on the return of any public method**
 - Properties of class usage
 - **Order of method calls**

Implementing Contracts

- **Contracts document methods**
 - Especially good for interfaces
 - Want to include them in the interface
 - Want to include in code as needed
- **Javadoc**
 - Provides English description that includes the contract conditions
 - This is the **minimum** that should be done
 - But it does no checking

Assertions

- This is a means of adding the checking
 - `assert <condition> [: <message>];`
 - Condition should not have side effects
 - Message describes the condition
- Compile with `-ea` (enable assertions)
 - This causes code to be checked
 - `AssertionError` thrown if violated
 - Without `-ea`, this is a comment
- Often easier/clearer than defensive code
 - But harder to recover from

Java Modeling Language

- **JML is a notation for defining contracts**
 - Preconditions, postconditions, invariants
 - Support in Eclipse if plugin installed
 - Some support for checking
 - Static checking
 - Dynamic checking
- **Uses stylized comments**
 - `/*@ <JML statement> ... @*/`
 - `//@ <JML statement`

JML Statements

- **requires** <expression>;
 - Specifies precondition
 - Expression can access parameters and fields
- **ensures** <expression>;
 - Specifies postcondition
 - Parameters, fields
 - **\old(expression)** : computed w/ calling values
 - **\result** : the return value
 - **\forall**, **\exists**
- **signals** (Exception e) <expression>;
 - Postcondition when an exception returned

JML Class Statements

- `invariant <expression>;`
 - Expression can contain fields
 - Applies after public methods
 - Not checked after internal/private methods
- `non_null`
 - Inside a field definition
 - Short for `invariant <field> != null;`
- Other constructs
 - Assert statements
 - Can define JML fields and methods

Review

- **Requirements, Specifications & Design**
 - Can be documented **on-line**
 - Provide the information for implementation
 - Understanding what you need to do
 - Explaining the system to others
- **UML**
 - Formalizes scenarios
 - Class diagrams show system structure
 - Good for initial design
 - Good for presentation
- **Contracts**
 - Document methods in interfaces & classes

After the Break

- **Requirements**
 - Preparation for final project requirements
- **XML**
- **Design patterns**