

# CSCI0320

## Introduction to Software Engineering

### Lecture 4

#### Requirements, XML, Patterns

# Requirements

- You now have a project **idea**
  - What is the next step?
    - Go build it?
    - Design it?
  - First you need to **know what to build**
    - But before that
    - You need to know **what the users want**
- Determining what users need is called
  - **REQUIREMENTS**

# Importance of Requirements

- **Poor requirements cause project failures**
  - 13% fail due to incomplete requirements
  - 12% fail due to lack of user involvement
  - 10% fail due to unrealistic expectations
  - 9% fail due to changing requirements
  - 7% fail because the system is no longer needed
- **Requirements is an on-going process**
  - Good first approximation
  - Can't anticipate all the questions
  - Building the system creates opportunities
  - Users are fickle

# Requirements Goals

- Define the problem from **users viewpoint**
- Determine outlines of the “best” solution
- Determine what is required & optional
  - **Set priorities**
- Determine **limitations** on resources
  - Environment, system, target architecture, interacting systems, ...
- Determine acceptance criteria
  - Requirements should be testable
  - Requirements should be precise
- **Do not worry about implementation**

# Obtaining Requirements

- How might you go about doing this?
- Where do you start?
- What do you want to get out of it?

# Obtaining Requirements

- **Interviewing** potential users
  - Structured or unstructured
  - Requires considerable **preparation**
    - Determine what information is needed
    - Find out about interviewees
    - Decide on questions and organization
  - **Process**
    - Move from general to specific questions
    - End with general questions
    - Record the interview
      - Stories, use-cases, scenarios
    - Follow up

# Obtaining Requirements

- **Questionnaires**
  - Where there is a large set of users
  - There are hard to develop as well
  - Leave some open-ended questions
  - Try to avoid bias
- **Observation**
  - Where you are automating an existing setup
- **Prototyping**
  - Develop a prototype and let users play
    - Paper prototypes are okay
  - Prototypes are meant to be thrown away

# Continuous Requirements

- **Requirements is an ongoing process**
  - Initially you get a set of use cases
    - These will not fully cover requirements
    - These are not even internally complete
    - They will probably be inconsistent
  - View these as a starting point
    - Determining what the user really wants
- **Requirements will change**
  - New use cases/scenarios will be developed
  - As users get a better idea of what the system can/will do, their ideas will change
- **Keep the user involved**
  - Agile requirements

# Final Project Requirements

- **This Week: Project Idea**
  - First step toward requirements
  - A few sentences describing the project
- **Early Next Week: Initial Requirements**
  - Describe the system as the user sees it
    - Scenarios describing functionality
    - One to four pages
  - **Survey/questionnaire** to get user input
    - Depends on your target audience
- **Late Next Week: Final Requirements**
  - Incorporate user feedback

# Galaxy Part II

- **How exciting is the example in Solar**
  - Want to explore other models
- **Things to think about**
  - **Handling large numbers of objects**
    - Barnes-Hutt approach
    - Different kinds of objects
  - **Handling lots of different scales**
    - Dynamically computing time increment
  - **Reading and writing models**
    - Reading so we can share models
    - Writing because they need a long time to run

# Creating Component

- **Create components from a file**
  - **How should we define this file**
    - **Language for components**
    - **Application-specific language**
      - **SUN <mass> <x pos> <y pos> <z pos> ...**
    - **Scheme**
      - **(Sun (mass x) (position x y z) (velocity a b c))**
    - **Create a parser & interpret the result**
  - **Today we avoid this**
    - **Parsers are a pain to write**
    - **Languages are a pain to learn and remember**
    - **Typically we use XML**

# XML History

- **SGML**: text markup for publication
  - Flexible, human created and edited
  - A way of getting print quality from multiple devices (printers, displays, typesetters)
  - Used for manuals, technical documentation
- **HTML**
  - Original web needed a simple standard
  - HTML was simplified SGML
  - Easy interpret, easy to write manually

# XML

- **SGML/HTML are difficult to parse**
  - Why?
- **Want something more standard**
- **Want something more general**
  - Why only describe documents for viewing
  - Why not describe arbitrary data
- **Want something that is self-descriptive**

# What is XML

- **Semi-structured** data representation
  - Uses trees to define data structures
  - Nodes containing subnodes
    - With attributes
  - Arbitrary text do define contents
- **Standard format**
  - Easier for a computer to parse
  - No exceptions to open-close
  - No special cases (e.g. attributes)
  - Some simplifications

# Example

```
<SOLAR>
  <OBJECT NAME='Sun' MASS="1.98e30" RADIUS='1000000'>
    <POSITION X='0' Y='0' Z='0' />
    <VELOCITY X='0' Y='0' Z='0' />
  </OBJECT>
  <OBJECT NAME='Object_1' MASS='6e24' RADIUS='12756'>
    <VELOCITY X='0' Y='30000' Z='0' />
    <POSITION X='1.5e11' Y='0' Z='0' />
  </OBJECT>
  <OBJECT NAME='Object_2' MASS='6e24' RADIUS='12756'>
    <POSITION X='-1.5e11' Y='0' Z='0' />
    <VELOCITY X='0' Y='-30000' Z='0' />
  </OBJECT>
</SOLAR>
```

# Other XML Elements

- **Headers**
  - `<?xml version="1.0" ?>`
- **Comments**
  - `<!-- comment --!>`
- **Arbitrary text**
  - `<![CDATA[ data goes here ]]>`
- **Namespaces**
  - `ns:name`

# XML Semantics

- XML provides **syntax only**
  - No semantics
- User can interpret however they want
- To be useful, must have a meaning
  - Reader and writer need to agree
  - Units in Galaxy input
- Examples
  - SOAP, XHTML

# XML is Self-Describing

- **Syntax can be specified**
  - Either as **DTDs** (original XML model)
  - Or as **XML Schema** (more recent)
- **DTDs**
  - Provide a CFG for trees
  - And define properties for attributes
- **XML Schema**
  - Does the same using XML directly

# DTD Example

- `<!ELEMENT OBJECT ( POSITION | VELOCITY )* >`
- `<!ATTLIST OBJECT MASS NMTOKEN #REQUIRED >`
- `<!ATTLIST OBJECT NAME ID #REQUIRED >`
- `<!ATTLIST OBJECT RADIUS NMTOKEN #REQUIRED >`
  
- `<!ELEMENT POSITION EMPTY >`
- `<!ATTLIST POSITION X NMTOKEN #REQUIRED >`
- `<!ATTLIST POSITION Y NMTOKEN #REQUIRED >`
- `<!ATTLIST POSITION Z NMTOKEN #REQUIRED >`
  
- `<!ELEMENT SOLAR ( OBJECT+ ) >`
- `<!ATTLIST SOLAR TIME NMTOKEN #REQUIRED >`
- `<!ATTLIST SOLAR YEARS NMTOKEN #REQUIRED >`
  
- `<!ELEMENT VELOCITY EMPTY >`
- `<!ATTLIST VELOCITY X NMTOKEN #REQUIRED >`
- `<!ATTLIST VELOCITY Y NMTOKEN #REQUIRED >`
- `<!ATTLIST VELOCITY Z NMTOKEN #FIXED "0.0" >`

# XML Schema Example

- `<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">`
- `<xs:element name="OBJECT">`
- `<xs:complexType>`
- `<xs:sequence                  <!-- xs:choice --!>`
- `<xs:element ref="POSITION" />`
- `<xs:element ref="VELOCITY" />`
- `</xs:sequence>`
- `<xs:attribute name="NAME" type="xs:ID" use="required" />`
- `<xs:attribute name="MASS" type="xs:NMTOKEN" use="required" />`
- `<xs:attribute name="RADIUS" type="xs:NMTOKEN" use="required" />`
- `</xs:complexType>`
- `</xs:element>`
  
- `<xs:element name="POSITION">`
- `<xs:complexType>`
- `<xs:attribute name="Z" type="xs:NMTOKEN" use="required" />`
- `<xs:attribute name="Y" type="xs:NMTOKEN" use="required" />`
- `<xs:attribute name="X" type="xs:NMTOKEN" use="required" />`
- `</xs:complexType>`
- `</xs:element>`

# XML Schema Example

- `<xs:element name="SOLAR">`
- `<xs:complexType>`
- `<xs:sequence>`
- `<xs:element ref="OBJECT" maxOccurs="unbounded" />`
- `</xs:sequence>`
- `<xs:attribute name="TIME" type="xs:NMTOKEN" use="required" />`
- `<xs:attribute name="YEARS" type="xs:NMTOKEN" use="required" />`
- `</xs:complexType>`
- `</xs:element>`
  
- `<xs:element name="VELOCITY">`
- `<xs:complexType>`
- `<xs:attribute name="Z" type="xs:NMTOKEN" use="required" fixed="0.0" />`
- `<xs:attribute name="Y" type="xs:NMTOKEN" use="required" />`
- `<xs:attribute name="X" type="xs:NMTOKEN" use="required" />`
- `</xs:complexType>`
- `</xs:element>`
  
- `</xs:schema>`

# XML Usage

- **Creating XML**
  - Lots of editors available
  - Can use plain text
  - View in browser or other tools
- **When** should XML be used
  - For inter-program input/output
  - For data files that need to be read
  - For output designed for people to read
  - For lots of standard program input
- **Uses**
  - Office 2007, Open office (zipped), resource files, web applications (**AJAX**)
- **Alternatives: json, ...**

# XML Pros and Cons

- **Pros**
  - Parsing is free (Java, XercesC, JavaScript)
  - Readable for debugging purposes
  - Human editable
  - Language and byte-order independent
  - Time independent (flexible)
- **Cons**
  - Uses lots of space
  - Does not include semantics

# XML in Java

- **Parsing**
  - **SAX**: callback-based parsing
    - You provide a callback class (DefaultHandler)
    - Parser does callbacks
      - Start/end of document
      - Start/end of each element
      - For each piece of text data
    - Useful for large documents, simple scanning
    - XMLEventReader, XMLStreamReader
  - **DOM**: parser builds XML structure
    - org.w3c.dom.\*
      - Node, Document, **Element**, TextElement, Attribute
      - NodeList
    - Easier to use than SAX

# DOM Parsing

- **Getting an XML document/Element**

```
DocumentBuilderFactory dbf =  
    DocumentBuilderFactory.newInstance();  
dbf.setValidating(false);  
DocumentBuilder db = dbf.newDocumentBuilder();  
FileReader fr = new FileReader(file);  
InputStreamSource ins = new InputStreamSource(fr);  
Document doc = db.parse(ins);  
Element root = doc.getDocumentElement();  
fr.close();
```

# DOM Access

- **Navigating** the document tree
  - `NodeList nl = elt.getElementsByTagName("tag")`
  - `getFirstChild(); getNextSibling();`
  - `getAttribute("Name");`
- **Properties** of a node
  - `getNodeValue()` – contents of a string
  - `getNodeName()` – name of an element
  - `getNodeType()`
    - `Node.ATTRIBUTE_NODE,`  
`Node.ELEMENT_NODE,`  
`Node.TEXT_NODE,`  
`Node.CDATA_SECTION_NODE`

# Helper Routines

- Useful routines to write
  - `getAttr<type>`
    - Convert attribute from string to proper type
    - Handle default values
  - `getText`
    - Get the text associated with an element
    - Concatenate multiple text elements
  - Iterator over children
- `/pro/ivy/xml/src` has examples

# Outputting XML

- Simple text output works fine
- Specialized classes
  - XMLOutputFactory
  - XMLStreamWriter
  - XMLEventWriter
- Write specialized class

# Design Solutions

- Designers work by **recycling** ideas
  - Experience provides more & better ideas
- Key to design
  - What ideas to use when
  - **What ideas will work for what problems**
- **Patterns** provide a way of encoding
  - The basic ideas
  - The information on how & when to use them

# Patterns in Design

- **Code** patterns
  - How to write code fragments
  - Loop over a set, binary search, ...
- **Data structure** patterns
  - How to build data structures
- **Design** patterns
  - How to organize classes and methods
- **Architectural** patterns
  - How to organize systems

# What is a Design Pattern

- What do you think it is?
- Can you give examples

# What is a Design Pattern

- **Problem to be addressed**
  - What the pattern is trying to do
  - Motivation for using the pattern
  - Exactly what the pattern does for you
- **Conditions**
  - When the pattern can be applied
  - Strengths and weaknesses of the approach
- **Implementation**
  - What are the classes and methods
  - How are the classes related
  - How are the methods implemented

# Caveats

- Solutions looking for problems
- **Understand & analyze the problem first**
  - Then think about what patterns might work
  - The problem description of the pattern is essential
- **Experience will tell what works**
  - And what doesn't
  - What is needed; what is not needed
- **Beware of added/excess complexity**

# Standard Patterns

- *Design Patterns*, Gamma et al
  - Also in the **course text** Software Design
  - Lots of other books
    - Specialized and general
  - Antipatterns
- Patterns can be classified by use
  - Factory patterns
  - Delegation patterns
  - Structural patterns
  - Control patterns
- We've seen patterns already
  - And will continue to

# Factory Patterns

- **How to create objects**
  - **Without resorting to new**
  - **Why might you want to do this**
- **Different ones for different circumstances**
- **Usual candidates**
  - **Factory class**
  - **Factory interface**
  - **Factory method**
- **Interesting alternatives**
  - **Prototype**: sample object that is cloned
  - **Singleton**
  - **Flyweight** objects: shared representations

# Delegation Patterns

- **Separate implementation and interface**
  - **Why**
    - Multiple possible implementations
    - Might want to reuse existing implementation
    - Implementation might need to change
    - Implementation might be remote
    - Implementation might be complex
- **Solutions**
  - Use another object: **Adapter, Wrapper**
  - Separate functionality: **Bridge, State**
  - Adding dynamic functionality: **Decorator**
  - Objects might not exist: **Proxy**
  - Objects might be complex: **Facade**

# Other Patterns

- **Structural patterns**
  - **Composite** (groups in Galaxy)
  - **Command**
- **Control patterns**
  - **Iterator**
  - **Strategy**: embedding an algorithm in a class
  - **Template**: algorithm with virtual hooks
  - **Visitor**: apply operations to a hierarchy
- **Algorithmic patterns**
  - **Mediator**: coordinate actions
  - **Memento**: save/restore state
  - **Observer**: publish-subscribe

# Using Patterns

- **When you run into a problem**
  - The solution isn't obvious
  - **Someone has probably seen it before**
    - Might be encoded as a design pattern
  - Have an awareness of the standard patterns
    - Will make finding alternative solutions easier
- **For describing a design**
  - Experienced programmers know patterns
  - Stating a pattern simplifies the description

# Next Time

- We'll start working with threads