Motivation

- Over the lifetime of a project, it is easy for documentation and implementation to diverge
  - Usually, documentation and code are not both simultaneously living entities
- Goal: Single point of control over change
  - Basic principle of software design (modularity)
  - If decision X changes, 1 modification needed
  - Alternative: changes needed in A, B, C, etc
- When that is not possible:
  - Make (logical) coupling between A, B, C obvious
  - When they get out of whack, code starts to smell
  - Items need to be co-located and visually linked

Bad Practice: Bad Hungarian

- Adding programming language type as prefix to variable name
  - eg Done (f for boolean flag)
  - Obfuscation, inconsistencies, redundancy, concrete coupling
- Adding semantic information to variable name, however, can be useful
  - eg radSunAzimuth
  - Can help to expose unit errors
    - if (radSunAzimuth == degMoonAzimuth) ...
    - inTableCircumference = 2*PI*cmTableRadius

Javadoc Comments

- Comment = main description + block tags
  - First sentence of main description is “summary”
  - Terminated by “.” followed by white space/new line
  - Appears at the top of page
  - Write comments in html (<p>, <pre>...)
  - Use html character entities (&lt; &gt; &amp;)
  - Avoid <h1> <h2>
- Block tags
  - @author, @param, @return, @see, @throws, @deprecated, ...
- Inline tags
  - Used within text of a documentation comment
    - {@link}, {@value}, {@code}, {@literal}, {@inheritDoc}, ...
Example

```java
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute URL. The name argument is a specifier that is relative to the url argument.
 * This method always returns immediately, whether or not the image exists.
 * @author Sun
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 */
public Image getImage(URL url, String name) {
    . . .
}
```

Standard Javadoc Tags

- `@param`: documents a single parameter of a method
  - Use one for each parameter of the method
  - Syntax: `@param parameter-name description`
  - Example:
    ```
    @param max the maximum number of words to be read
    ```
- `@return`: documents the return value of a method
  - Use one for each type of exception the method throws
  - Example:
    ```
    @return the number of words actually read
    ```
- `@throws`: documents an exception thrown by the method
  - Specify classes/interfaces with their simple names
  - Give full name if class/interface is from another package
  - Examples:
    ```
    @throws NullPointerException The name is `{code null}`
    ```
- `@see`: creates a cross-reference link to other javadoc documentation
  - Forms a "See also" section at the end of the documentation
  - Qualify the identifier sufficiently
  - Examples:
    ```
    @see #getName
    @see Attr
    @see com.hostname.attr.Attr
    @see com.hostname.attr.Attr#getName
    @see com.hostname.attr.Attr#getName(String, Object)
    @see com.hostname.attr.Attr#getName(String)
    @see <a href="spec.html#attr">Attribute Specification</a>
    ```
- `@version`
- `@since`: denote when the tagged entity was added to your system
  - Example:  Graphics.java    Output Documentation
    ```
    $ javadoc Graphics.java
    ```

Miscellaneous Features

- `@link`: similar to `@see`, but it embeds a cross reference in the text of your comments
  - Syntax: `@link package.class#member [label]`
  - Identifier specification follows the same requirement for `@see`
  - Example:
    ```
    Changes the value returned by calls to `{link #getValue}`
    ```
- `@deprecated`: marks that an identifier should no longer be used. It should suggest a replacement.
  - Example:
    ```
    @deprecated Use `{code setVisible(true)}` instead
    ```
- `@author`
- `@version`
- `@since`: denote when the tagged entity was added to your system
  - Example:  Graphics.java    Output Documentation
    ```
    $ javadoc Graphics.java
    ```
**Demo with Eclipse**

- Viewing Javadoc for JDK or current project
  - Mouse hover, or F2 for javadoc of method in editor window
  - Shift+F2 opens browser (prettier HTML display)
  - (Aside: F3 opens source!)
- Javadoc view
- Generating Javadoc
  - Add boiler-plate comments to a method/class/interface
    - Include @param, return, throws, author, see, deprecated
  - Customize these templates
  - Window > Preferences > Java > Code Style > Code Templates > Comments
  - Project > Generate Javadocs...
  - For details, see a later slide
- Formatting and validating Javadoc
  - Source > Format (Ctrl+Shift+F)
  - Window > Preferences > Java > Compiler > Javadoc

**Package Documentation**

- A package is not defined in one source file
- To generate package comments, add a package.html file in the package directory
  - The contents of the package.html between <body> and </body> will be read as if it were a doc comment.
  - @deprecated, @author, and @version are not used in a package comment
  - The first sentence of the body is the summary of the package
  - Any @see and (@link) tag must use the fully qualified form of the entity's name, even for classes and interfaces within the package itself
- You can also provide an overview comment for all source files by placing an overview.html file in the parent directory
  - The contents between <body> and </body> is extracted
  - The comment is displayed when the user selects "Overview"

**Good Practice: A Uniform Style**

- Consistency among team members
  - Omit { }'s from method names
  - Except: for overloaded methods, list parameter types in ()
  - Phrase for param's beginning with article + type: the image observer to be inserted in the selected buffer
  - For person descriptive tags: the image observer to the queue of active observers
  - Required optional tags
  - Ordering of block tags
    - param, return, throws, author, see, deprecated
- Sun's style guide
  - "How to Write Doc Comments for Javadoc"
  - http://java.sun.com/j2se/javadoc/writingdoccomments/
  - Virtually an industry-wide standard

**Good Practice: Doc the Contract**

- Javadoc comments describe a component's contract not its implementation
  - Describe what a method does, not how it does it
  - What a client component needs to know
  - Contract is usually more stable than implementation
- Describe method assumptions
  - Preconditions on arguments
    - Ceg, observer must be non-Null, list must contain target
  - Preconditions on object state
    - In terms of "public" (ie externally checkable, abstract) state
- Describe method guarantees
  - Postconditions on return value
    - @return true if and only if target is within image boundary
  - Postconditions on object state
  - Describe class invariants

**Tension? API vs Code**

- Documenting the contract is good
  - What clients need
  - What future code maintainers need
  - "Programmer's Guide"
- Documenting the implementation is good
  - Parts of abstract state the method is allowed to modify
  - Precondition (on abstract state and arguments) expected by method
  - @initially
  - Requirements on initialization (ie constructors)
- For which purpose should you use Javadoc?
  - Answer: both!
  - No contradiction if each component consists of both an interface and a class
    - Interface is the abstract component
    - Its Javadoc is for clients
    - Class is the concrete component
    - Its Javadoc is for implementers

**Custom Tags: Client's View**

- Interface-level tags
  - Abstract fields that define client-side view of state space
  - @mathdef
  - Definitions derivable from abstract state
  - @constraint
  - Invariant holding on abstract state
  - @initially
  - Requirements on initialization (ie constructors)
- Method-level tags within interfaces
  - @requires
  - Precondition (on abstract state and arguments) expected by method
  - @alters
  - Parts of abstract state the method is allowed to modify
  - @ensures
  - Postcondition (on abstract state) guarantee by method
Custom Tags: Implementer’s View

- Class-level tags
  - @convention
  - Invariant holding on concrete representation
  - @correspondence
  - Mapping from concrete representation to abstract state

- Constructor and method-level tags
  - None (the specification is in the interface)
  - Exception: helper (ie private) methods
    - Use @requires, @alters, @ensures for these methods
    - Predicates are on concrete representation (ie fields) and arguments

Using Custom Tags with Eclipse

- See:
  - Interface RandomWithParity
  - Classes AlternatingCoin and UnfilteredRandom
- Project > Generate Javadoc...
- Javadoc command: Browse to installed JDK directory, then bin/javadoc
  - eg /class/cse421/local/jdk1.6.0_26/bin/javadoc
- Next, then Next again
- Inside "Extra Javadoc options" box copy the text from cse421JavadocTags.txt (available from class web site)
- Finish
- After doing this once, these Javadoc options become defaults so you don’t have to re-enter them every time

Bad Practices: Miscellaneous

- End-of-function comments
  - public void setRate (int frequency) {
  - } //setRate
  - Obviated by modern editors with code folding

- Commenting bug fixes
  - Version control is a better place for this than Javadoc (more on version control later)

- Comments with no additional value
  - Repeating the parameter name as the description

- Leaving boiler-plate comments in code
  - Automatically generated Javadoc with obvious boiler plate code should never appear in repository
  - Don’t leave it hanging around your own code

Shortcomings

- Java-specific
- HTML output is the only first-class citizen
  - Sun provides only one doclet (produces HTML)
  - Others have been written by 3rd parties
- Geared towards API documentation
  - Contract specification (sort of, see below) only
  - Leaves out documentation for architecture, algorithms, defect tracking, etc
- No tags for pre/post conditions or invariants
  - These conditions should be checked by assertions (not exceptions) so @throws is not helpful
  - Several extensions exist (eg JML, our set of custom tags patterned after RESOLVE)

Alternative: Doxygen

- Javadoc-like comment tags and formatting
  - comment block with description and tags
  - author, param, return etc
- Supports multiple programming languages
  - C/C++, Java, C#, PHP, Python,
  - Comment syntax language dependent
- Supports multiple output formats
  - html, rtf, pdf, latex, man, xml, ...
  - Documentation text less html-ized
- Better support for design-by-contract
  - Has built-in tags for @pre, @post, @invar

Summary

- Structure of Javadoc comments
  - Free-form initial prose
  - Block tags (and in-line tags)
- Standard tags
  - @param, @return, @deprecated, @author, ...
- Custom tags for interfaces
  - @mathmodel, @mathdef, @constraint, @initially
  - @requires, @alters, @ensures
- Custom tags for classes
  - @constructor, @correspondence
- Eclipse support