DESIGN FOR REUSE
LOGISTICS

- HW5 due on 11/29
- Project presentation on 12/6
  - Will focus on a single design decision common to your reference system and two additional systems
  - For each of the three systems, describe the alternative design choices each of these systems made. What are the consequences of these design choices?
  - Presentation should be 5 minutes. To ensure we have enough time for all presentations, we will stop you and you will lose points if go over 6 mins. Please practice to ensure your talk is the correct length.
EXAMPLE

- Your goal: build a todo application in React
- How?
  - Need to learn how to reuse React framework
WHAT IS REUSE?

- Making use of previously written code rather than writing new code
- Often, reuse takes form of reusing a library or a framework
- Once made choice to reuse a library or framework, need to understand how to achieve specific behavior with library or framework
  - Often finding code snippets that achieve desired behavior
APPLICATION PROGRAMMING INTERFACE (API)

- Boundary between code to be reused (library or framework) and client which reuses code
- We've looked previously at abstractions
  - Design goal: chose operations which make key reuse scenarios short
  - Choice of what operations to support one of the most important choices in API design
- Today we'll look more broadly at additional considerations in designing code for reuse
API QUALITY ATTRIBUTES

- Largely similar to normal system design, but for client code

- Usability
  - Learnability
  - Error prevention
  - Consistency
  - Matching user mental models

- Power
  - Extensibility: ability for users to create new elements
  - Evolvability: ability for designers to change API
  - Performance: speed, memory consumption
  - Security
SOME EXAMPLES OF API DESIGN DECISIONS

**Structural Design Decisions**
(Separating Functionality into Classes and Interfaces)
- Design Patterns
  - Factory Pattern
  - Asynchronous Pattern
- Package Design
  - Naming
  - Hierarchy
  - Size
  - Composition vs inheritance
  - Maximizing information hiding
  - Matching organization roles
  - Class vs struct
  - Class vs interface
  - Interfaces vs abstract classes

**Class Design Decisions**
(Separating a Class’s Functionality into Methods and Fields)
- Class name
- Returning arrays
- Finalizers
- Collections
- Arrays
- Static
- Nested
- What Methods and Fields to Provide
- Individual Method / Field Design
- Exceptions
  - Standard types
  - Types
  - When to use
  - Message
  - Caught vs uncaught
  - Performance
- Naming
  - Static
  - Synchronized
  - Sealed
  - Protected
  - Virtual
  - Final
  - Abstract
- Parameter Design
  - Constructors
  - Types
  - Naming
  - Ordering
  - Events
  - Parameter checking
  - Float vs double

**Architectural Decisions**

**Language Level Decisions**
MORE API DESIGN DECISIONS

- Documentation
  - What to cover
  - How to communicate: descriptions of methods? examples?
  - Audience: experts? novices? users of competing APIs?
HOW DEVELOPERS REUSE CODE

- In order to understand how to successfully design for developers reusing code, helpful to understand how developers reuse code

- Software engineering researchers run user studies to identify general strategies and challenges developers experience

- Companies with large API ecosystems (e.g., Google, Facebook, Microsoft) run user studies to evaluate and improve specific API designs
CHALLENGES WITH REUSE

- **Design** barriers—inhomogeneous cognitive difficulties of the programming problem, separate from notation used
  - I don’t know what I want the computer to do
- **Selection** barriers—finding programming interfaces available to achieve a particular behavior
  - I don’t know what to use
- **Coordination** barriers—constraints governing how languages & libraries can be combined
  - I don’t know how to make them work together
- **Use** barriers—determining how API how to use API
  - I don’t know how to use it
- **Understanding** barriers—environment properties such as compile & runtime errors that prevent seeing behavior
  - It didn’t do what I expected
- **Information** barriers—environment properties that prevent understanding runtime execution state
  - I think I know why didn’t behave as expected, but don’t know how to check
CHALLENGES WITH REUSE

- Mapping an abstract conceptual solution into the appropriate elements
  - “How do I create a rectangle? Why is there no Rectangle tool?”
- Understanding control & data flow, hidden dependencies due to run-time binding or inheritance, between classes in the API
  - “I’m over-riding SelectionTool, and in particular mouseDown() so that when the figure is clicked the box is drawn. This bit works, however when trying to drag the figure, if I do something similar the rectangle flickers like mad.”
- Understanding how functionality works
  - “How does … work?”, “What does … do?” or, “Where is … defined/created/called?”
- Making changes consistent w/ architectural constrains of API
  - Violating constraints of MVC architecture by passing references in prohibited ways

VOCABULARY PROBLEM

- Developers are familiar with concepts using one set of terminology.
- API, tutorials, or other resources use different terminology.
- How do developers find the right concepts with alternative terms?
CHALLENGES MAY VARY BY CONTEXT

- Size of desired snippet
  - Reusing a line of code? A whole algorithm?

- Alternatives
  - How many alternatives are there? How important is it to find the best alternative?

- Integration
  - What libraries or frameworks does a snippet require? How can they be integrated?
SOME EXAMPLES OF REUSE STRATEGIES

▸ You'd like to reuse method x in framework f. How do you figure out how to do this?

▸ Example reuse strategies
  ▸ Read the documentation
  ▸ Read tutorials
  ▸ Find StackOverflow snippets
  ▸ Find similar code in your own codebase that also reuses x
  ▸ Try out API functions, see what they do
OPPORTUNISTIC VS. SYSTEMATIC DEVELOPERS

- Developers vary in which sorts of strategies they prefer
- Key choice: how completely do you need to understand API before deciding your understanding is "good enough"
  - Systematic: as much as possible
  - Opportunistic: as little as possible
- This leads to different developers preferring different types of strategies
  - Opportunistic developers more likely to start with example code
  - Systematic developers more likely to read the documentation first
- ---> API documentation should support both types of strategies
STRATEGIES VARY WITH DEGREE OF PRIOR KNOWLEDGE OF API

<table>
<thead>
<tr>
<th>WEB SESSION INTENTION:</th>
<th>LEARNING</th>
<th>CLARIFICATION</th>
<th>REMINDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason for using Web</td>
<td>Just-in-time learning of unfamiliar concepts</td>
<td>Connect high-level knowledge to implementation details</td>
<td>Substitute for memorization (e.g., language syntax or function usage lookup)</td>
</tr>
<tr>
<td>Web session length</td>
<td>Tens of minutes</td>
<td>~ 1 minute</td>
<td>&lt; 1 minute</td>
</tr>
<tr>
<td>Starts with web search?</td>
<td>Almost always</td>
<td>Often</td>
<td>Sometimes</td>
</tr>
<tr>
<td>Search terms</td>
<td>Natural language related to high-level task</td>
<td>Mix of natural language and code, cross-language analogies</td>
<td>Mostly code (e.g., function names, language keywords)</td>
</tr>
<tr>
<td>Example search</td>
<td>“ajax tutorial”</td>
<td>“javascript timer”</td>
<td>“mysql_fetch_array”</td>
</tr>
<tr>
<td>Num. result clicks</td>
<td>Usually several</td>
<td>Fewer</td>
<td>Usually zero or one</td>
</tr>
<tr>
<td>Num. query refinements</td>
<td>Usually several</td>
<td>Fewer</td>
<td>Usually zero</td>
</tr>
<tr>
<td>Types of webpages visited</td>
<td>Tutorials, how-to articles</td>
<td>API documentation, blog posts, articles</td>
<td>API documentation, result snippets on search page</td>
</tr>
<tr>
<td>Amount of code copied</td>
<td>Dozens of lines (e.g., from tutorial snippets)</td>
<td>Several lines</td>
<td>Varies</td>
</tr>
<tr>
<td>from Web</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediately test copied code?</td>
<td>Yes</td>
<td>Not usually, often trust snippets</td>
<td>Varies</td>
</tr>
</tbody>
</table>

TYPES OF REUSE

- Learning—relies on selecting highest quality tutorials tutorials
  - e.g., “update web page without reloading php”
- Clarification—learning syntax based on exiting understanding of the domain concepts
  - e.g., reminding use of syntax of HTML forms
  - Often search by analogy to domain concepts in other languages / frameworks
    - e.g., Perl has a function to format dates as strings, what’s the one for PHP?
- Reminder—using web as external memory aid
  - e.g., forgot a word in a long function name
  - e.g., 6 lines of code necessary to connect and disconnect from MySQL database copied hundreds of times by individual

EFFECTS OF API DESIGN CHOICES: METHOD PLACEMENT

- Where to put functions when doing object-oriented design of APIs
  - `mail_Server.send( mail_Message )`
  - vs.
  - `mail_Message.send( mail_Server )`
- When desired method is on the class that they start with, users were between 2.4 and 11.2 times faster

[Stylos FSE, 2008]
EFFECTS OF API DESIGN CHOICES: REQUIRED PARAMETERS IN CONSTRUCTORS

- Compared default constructor (create-set-call)
  - var foo = new FooClass();
  - foo.Bar = barValue;
  - foo.Use();
- Results
  - All developers assumed there would be a default constructor
  - Required constructors imposed premature commitment: had to figure out how to construct object before could decide if it was the right object for task
  - Did not insure valid objects - passed in null

- vs. required constructor
  - var foo = new FooClass(barValue);
  - foo.Use();

[Stylos & Clarke, ICSE’07]
EFFECTS OF API DESIGN CHOICES: FACTORIES

- Compared “normal” creation: Widget w = new Widget();
- vs. creation using factory pattern
  - AbstractFactory f = AbstractFactory.getDefault();
  - Widget w = f.createWidget();
- Factory pattern frequently in Java (>61) and .Net (>13) and SAP

Results

- Time to develop using factories took 2.1 to 5.3 times longer compared to regular constructors (20:05 vs 9:31, 7:10 vs 1:20)
- All developers had difficulties using factories in APIs
  --> Very important if using factory to document how to create objects
- Particularly in class developers might start with

[Ellis 2007]
RECOMMENDATIONS FOR API DESIGN AND DOCUMENTATION

▸ Given these (and other) findings, what should you do to design for reuse?

▸ Some recommendations

▸ Create effective documentation

▸ Make API design choices which optimize for usability and power quality attributes
CREATE EFFECTIVE DOCUMENTATION

- Include **short code snippets** that document API usage patterns of how multiple methods work together and capture **best way to use API**

- Focus on documenting higher level usage, not boilerplate documentation that adds little beyond method signatures

- Match **scenarios** capturing common use cases to how to do that in API

- Include discussion of **performance** consequences of specific API usage
MAKE EFFECTIVE DESIGN CHOICES FOR USABILITY

- Design problem similar to designing for software for users more generally
  
  - Can apply Nielsen's Heuristic evaluation heuristics to API design (see SWE 632 for more!)
VISIBILITY OF SYSTEM STATUS

- Should be easy for API user to check state of framework
  - e.g., whether file is open or closed
- Using wrong operation for the current state should generate appropriate feedback
  - e.g., writing to closed file should generate meaningful error message
MATCH BETWEEN SYSTEM AND REAL WORLD

- Names given to methods and organization of methods into classes should match API users' expectations
  - e.g., user wanting to write to File most likely to look for File class first, not FileOutputStream
- Users often interact with class first by creating an instance
USER CONTROL AND FREEDOM

- API users should be able to abort or reset operations and return the API back to previous state
CONSISTENCY AND STANDARDS

- All design choices should be consistent across API
  - e.g., naming of classes and methods, naming of arguments, order of arguments, placement of methods into classes

- Example violation: order of arguments in opposite order
  - void writeStartElement(String namespaceURI, String localName)
  - void writeStartElement(String prefix, String localName, String namespaceURI)
ERROR PREVENTION

- API should guide user into doing the right thing
- Have defaults that match users' expectations
- Avoid using String parameters, particularly long sequences of String parameters
  - Compiler cannot check if arguments in correct order
  - e.g., void setShippingAddress (String firstName, String lastName, String street, String city, String state, String country, String zipCode, String email, String phone)
RECOGNITION RATHER THAN RECALL

- Developers often try to find the right method through autocomplete
- Make names clear and understandable, so users can recognize what they want
FLEXIBILITY AND EFFICIENCY OF USE

- API users should be able to accomplish their tasks efficiently
HELP USERS RECOGNIZE, DIAGNOSE, RECOVER FROM ERRORS

- When a developer uses API incorrectly, API should offer error messages that explain the problem and offer suggestions on how to resolve issue
SUMMARY

- Developers spend much of their time interacting with libraries and frameworks through APIs.
- Developers differ in use of opportunistic and systematic strategies for reuse, requiring different considerations in API and documentation design.
- Documentation that focuses on scenarios and best practice usages, rather than boilerplate, can make a big impact in usability.
- Many design choices such as naming, organization of functionality into classes, and error messages can have a profound choice on usability.
- Can apply usability heuristics to API design.
IN CLASS ACTIVITY
APPLY API DESIGN HEURISTICS

▸ Work individually, pick a framework (e.g., .NET framework, Java standard library, React, ...)

▸ Critique the framework using API design heuristics (Slides 24-31)

▸ Identify one example for each heuristic (8 total) where the framework either follows or violates the heuristic
  
  ▸ For example of the 8 examples, list the name of the heuristic, give an element within the framework (e.g., method, class), and describe how element either follows or violates the heuristic

▸ Hand in through Blackboard
  
  ▸ Your name, choice of framework, description of 8 examples