Objectives

- Analysis and Design
- Designing APIs

Take-Home: SLogo Design

- SLogo: Final project
  - Sketch of interface
  - Sketch of classes
- Those who already started: <= 20 minutes
- Those who haven't started: <= 30 minutes

➔Due Wednesday

ANALYSIS & DESIGN: FORMALIZED

Design Heuristics

- Model real world whenever possible
- Avoid all-powerful (omnipotent) classes
- Distribute system intelligence among classes evenly
  - Top-level classes should share work uniformly
  - More easily understood system
  - More easily communicated design
- Minimize # of messages between class and helper
  - Reduce coupling btw class and helper

Analysis Phase

- Create an abstract model in client's vocabulary
- Strategy:
  - Identify classes that model (shape) system as set of abstractions
  - Determine each class's purpose, or main responsibility
    - member functions
    - data members
  - Determine helper classes for each
    - Help complete responsibilities

Analysis Phase Discussion

- Expect to iterate
  - Won't find all classes at first
    - Especially helpers
  - Won't know all responsibilities
- Uncertainty in problem statement
  - May be concerns that need to be settled
  - Try to understand requested software system at level of those requesting software
- Rarely one true correct best design

Identification of Classes

- Potentially model the system
- Usually nouns from problem description
  - Or from domain knowledge
- Model real world whenever possible
  - More understandable software
  - Helps during maintenance when someone unfamiliar with system must update/fix code

Identifying Responsibilities

- Responsibilities convey purpose of class, its role in system
- Questions to Ask:
  - What are the other responsibilities needed to model the solution?
    - Which class should take on this particular responsibility?
  - What classes help another class fulfill its responsibility?

Have You Modeled Everything?

- Strategy: Role playing
- Act as different classes: can you do everything you want in various scenarios?
  - Fill in missing classes, responsibilities
  - Methods: parameters, what returned
  - Restructure as necessary
    - No code yet so not actually refactoring
- Example use cases/scenarios:
  - User borrows a video and returns it two days late
  - User tries to borrow book that is already checked out

Discussion

- What else can use cases be used for?

Discussion

- What else can use cases be used for?
  - Test Cases

Josh Bloch's

DESIGNING A GOOD API

How to Design a Good API

- APIs can be among a company's greatest assets
  - Customers invest heavily: buying, writing, learning
  - Cost to stop using an API can be prohibitive
  - Successful public APIs capture customers
- Can also be among company's greatest liabilities
  - Bad APIs result in unending stream of support calls
- Public APIs are forever - one chance to get right

How Does API Design Relate To This Class?

How Does API Design Relate To This Class?

- You are an API designer
  - Good code is modular–each module has an API
- Useful modules tend to get reused
  - Once module has users, can't change API at will
  - Good reusable modules are corporate assets
- Thinking in terms of APIs improves code quality

Characteristics of a Good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

API DESIGN PROCESS

API Design Process

- Gather requirements
  - Eye towards generality
  - Know what is actually required
- Write short specification
- Write to API early
- Maintain realistic expectations

One-Page Specification

- Agility trumps completeness
- Bounce spec off as many people as possible
  - Listen to their input and take it seriously
- If you keep the spec short, it's easy to modify
- Flesh it out as you gain confidence
  - Necessarily involves coding

Write to Your API Early and Often

- Start before you've implemented the API
  - Saves you doing implementation you'll throw away
  - Similar to role playing
- Start before you've even specified it properly
- Saves you from writing specs you'll throw away
- Continue writing to API as you flesh it out
  - Prevents nasty surprises
  - Code lives on as ***examples, unit tests***

Maintain Realistic Expectations

- Most API designs are over-constrained
  - You won't be able to please everyone
  - Aim to displease everyone equally
- Expect to make mistakes
  - A few years of real-world use will flush them out
  - Expect to evolve API

DESIGN PRINCIPLES

API Should Do One Thing and Do it Well

- Functionality should be easy to explain
  - If it's hard to name, generally a bad sign
  - Good names drive development
  - Be amenable to splitting and merging modules

API: As Small As Possible, But No Smaller

- API should satisfy its requirements
- When in doubt leave it out
  - Functionality, classes, methods, parameters, etc.
  - You can always add, but you can never remove
- *Conceptual weight* more important than bulk
- Look for a good *power-to-weight* ratio

Implementation Should Not Impact API

- Why?

Implementation Should Not Impact API

- Implementation details
  - Confuse users
  - Inhibit freedom to change implementation
- Be aware of what is an implementation detail
  - Do not overspecify the behavior of methods
  - For example: do not specify hash functions
- All tuning parameters are suspect
  - Don't let implementation details "leak" into API
  - On-disk and on-the-wire formats, exceptions

Minimize Accessibility of Extensibility

Minimize Accessibility of Extensibility

- Make classes and members as private as possible
- Public classes should have no public fields
  - With the exception of constants
- This maximizes information hiding
- Allows modules to be used, understood, built, tested, and debugged independently

Names Matter – API is a Little Language

Names Matter – API is a Little Language

- Names should be largely self-explanatory
  - Avoid cryptic abbreviations
- Be consistent–same word means same thing
  - Throughout API, (Across APIs on the platform)
- Be regular–strive for symmetry
- Code should read like prose

```
if (car.speed() > 2 * SPEED_LIMIT)
     generateAlert("Watch out for cops!");
```

Documentation Matters

Documentation Matters

Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

-- D. L. Parnas. "Software Aging." In *Proceedings of 16th International Conference Software Engineering*, 1994

Document Religiously

- Document every class, interface, method, constructor, parameter, and exception
  - Class: what an instance represents
  - Method: contract between method and its client
  - Preconditions, postconditions, side-effects
  - Parameter: indicate units, form, ownership
- Document state space very carefully

Consider Performance Consequences of API Design Decisions

Consider Performance Consequences of API Design Decisions

- Bad decisions can limit performance
  - Making type mutable
  - Providing constructor instead of static factory
  - Using implementation type instead of interface
- Do not warp API to gain performance
  - Underlying performance issue will get fixed, but headaches will be with you forever
  - Good design usually coincides with good performance

Effects of API Design Decisions on Performance are Real and Permanent

- `Component.getSize()` returns `Dimension`
  - `Dimension` is mutable
  - Each `getSize` call must allocate `Dimension`
- Causes millions of needless object allocations
- Alternative added in 1.2
  - Old client code still slow