Announcements

- Group dynamics can be hard.
  - Meet, in person, *regularly*.
  - Your first priority should be resolving problems internally.
  - But we will listen if you have group/pair problems to report.
  - In the end, git tells us a lot. So does talking to you in person.

- Term project demos will be the last two days of Reading Period.
Team Design

John Jannotti

course/cs0320/www/docs/lectures/

Apr 11, 2017
Web Browsing

- Web architecture
  - It all works using sockets and messages
  - Server listens on port 80
  - Browser connects socket to that port
- Sends message(s) requesting a page(s)
- Using HTTP format
The split between the browser and server
  ▶ Well-defined interface (175 pages)
  ▶ Allows separate implementations.
  ▶ Of the top 10 OS browsers and servers, only MS does both.

This is an example of team design.

Why is this needed?
Team Problems

- If one person works on a project —
  - Works alone, keeps design/changes in head.
- If two people work on a project —
  - One channel of communication.
- If four people work on a project —
  - Six channels of communication.
- If ten people work on a project?
- If 100 people work on a project?
Communication Problems

- It takes time
  - Meetings.
  - Waiting for people (finding the right people).
  - Non-productive time.
- It is error-prone
  - Miscommunication is likely
  - Missing assumptions
  - Different vocabularies
- Want to minimize it
  - Maximize “real” design/programming.
Principles of Team Design

- Separation of concerns
- Abstraction
- Simplicity
- Well-defined interfaces
- Minimize connections
- Minimize risk
Separation of Concerns

- Divide the project into *independent* parts
- Implementation independence
  - A part’s implementation choices shouldn’t affect other parts
  - Design implementations independently
- Typically matches separation of people
  - Especially for term project sized projects.
  - One person might have multiple concerns.
Abstraction

- Isolate the implementation from its uses
  - Implementation hidden by an abstraction
  - Callers work with the abstraction
- Isolate code of different programmers
  - Each programmer provides an abstraction
  - Others code to this abstraction
- Work in terms of manageable units
  - Don’t implement something too big at once
  - Able to keep track of what to use, others work, etc.
- Put off certain decisions as long as possible
  - Those that affect implementations
Law of Demeter

- Each unit should have limited knowledge of other units
  - Each unit should only talk to its friends
  - Don’t talk to strangers

- Friends of a method
  - Methods of this
  - Methods of argument classes
  - Methods of fields of this
  - Rule of thumb: Avoid two dots (x.y.z()) in an expression.
  - Call Order.getEmail(), not Order.getUser().getEmail()
  - Easier transition to anonymous orders or group orders.
Simplicity

- Complexity for an “expected” problem is often not worth it
- YAGNI (You Ain’t Gonna Need It)
- Don’t worry about performance
  - At least not at first
  - Hard to tell where performance is important.
  - Hard to tell where fancy code is required.
  - Never optimize increase complexity for performance without profiling.
- Keep things as simple as possible.
- Operate at the highest abstraction-level possible.
Interface Simplicity

- Minimize exposed information.
- Keep interfaces small.
  - Small number of methods.
  - Only what is needed, no more.
  - No fields should be exposed.
- Work in terms of interfaces.
  - Not data structures. (Map, not HashMap)
  - Not algorithms. (A stable sort, not Quicksort)
  - Packages should have few public classes.
- Mutability is a **serious** form of complexity.
  - You’ve been spared the tribulations of tracking “ownership” in C.
  - Don’t recreate them in a GC’d language.
Code simplicity

- Minimize number of arguments.
- Each new type/class should have a single, succinct purpose.
- Use standard, basic types where possible.
  - Don’t make your own Day class that holds a year, month, day.
  - But never “encode” complex types in strings, integers.
  - Don’t use a String where a Date or small class is more accurate.
  - An Actor is not a String (ID or name? How to get films?)
Minimize Communication

- Between system components
  - By using minimal interfaces.
- Between classes and interfaces
  - Minimize number of methods in interfaces
  - Minimize communications paths
- Coupling & cohesion
  - Minimize dependencies between classes
  - Do your classes *really* need to know each other exists?
Minimize Risk

- Start the design with what you don’t know
  - Determine what is hard about your problem.
  - Determine what you don’t understand.
  - Determine what might not work.

- Encapsulate these problems
  - Isolate the implementation of these.
  - Prototype or experiment as needed.
Working in Teams

- Principles are great
  - but they don’t tell you what to do
- We’ll discuss actual high-level system designs
  - What works well for teams
  - What approaches are best
Client-Server Design

- This is one we’ve seen and used
  - Client and server are independent
  - What are their interfaces?
    - Messages that go each way.
    - These need to be simple and well-defined.
    - Usually requests and replies.
    - Messages are naturally “immutable” when serialized.

- Not limited to networks
  - “Server” might be a thread with a work list.
  - “Client” might get response by callback.
  - You lose “natural” immutability.

- Problems
  - Only separates two components
  - Each might still be complex (Web Server/Browser)
Core + Extensions

- Identify a minimal system core
  - Essential elements used by all of system
  - The heart of the application
  - Remember: *minimal*

- Everything else is a separate extension
  - Extensions only talk to the core.
  - Extensions don’t talk to each other.
  - OS device drivers follow this model.

- Core is written first, tested the most.
- Extensions can be added as needed, independently
- Obvious implementation strategy: person per extension
Libraries

- **Simplest form of team-design**
  - Library is an independent module that solves a complete subproblem.
    - Create a PDF. Send a fax. Charge a credit card.
    - Library might provide an abstraction, but should not use abstractions from the “application”.
    - The Factory pattern is minor work-around.
    - Think *Layers*. Dependencies go down.
  - Beware of complexity/coupling creep.

- **Consider libraries that need to interact with caller’s control-flow**
  - Thread pool
  - jQuery (really, the DOM event model)
  - JSON parser for big or incremental documents
  - How do they accomplish this?
Publish-Subscribe

- Each module defines extension points
  - Where it might provide information
  - Mouse/Key events, Timer event, Database save/load

- Other Modules subscribe to these points
  - If they need the information
  - If they want a chance to act
Publish-Subscribe “Internal” Frameworks

- Callbacks (Swing, HTML parser)
  - Publisher defines an interface class
  - Subscriber provides implementation
    - Passed to the publisher
  - Has become nicer with Java 8 lambdas and method references.

- Subclassing
  - Publisher provides inheritable class
    - With abstract/dummy methods for messages
  - Subscriber provides subclass
    - Providing proper method implementations
  - Usual drawbacks of inheritance.
External Frameworks

- Publisher provides a register interface
  - Subscribers tell publisher their interests
  - Publisher then sends out messages to those who have subscribed for particular data
  - Can work across network

- Messages stating interest
- Messages with information
- Central message service
  - Handles subscription requests
  - Handles all outgoing messages
  - Resends messages to proper targets
Advantages of Publish-Subscribe

- Provides very clean separation of code/concerns
  - Each module defines extension points

- Extension points
  - Provide small, well-defined interfaces
  - Can request or provide functionality

- Uses
  - QT, GTK+ GUI toolkits: events, keypresses, signals, slots
  - Eclipse: all about plug-ins with extensions
  - Javascript: Part of DOM model. Libraries like jQuery extend to user-defined events.
Iterative Process
- You wont get it right the first time
- Interfaces will change
  - Changing requirements, specifications
  - To match actual implementations

Continually refactor the design
- Always ask how can I do this better
  - How can it be simpler?
  - What can be eliminated?
- Abstract repeated code patterns.
- Get the best design for the long haul.
Think about creating independent parts
  ▶ Each person takes on one or more parts
  ▶ You should be able to demo them separately.

Each part defines a minimal interface
  ▶ Defined as Java interfaces
  ▶ Defined as messages
  ▶ Defined as callbacks

You must demo each person’s work *independently*!
You should already be using git.
- Members must be able to compile the entire project.
- Commit often, but don’t “Break the build”

Tests need to be as easy to run as project to compile.
- Best: Run them as part of the build process.
- Good enough: When you refactor, fix the tests before committing.
- Use a “build robot” to grab latest, build, and run tests.
- These tests might be even more extensive (slower to run).
- And email everyone about breakage.

Use github’s issue tracker.
Term Projects

- All your projects interact with users in some way.
  - User interface design is a critical part.
- User-friendly
- Easy to use / learn
- User efficient
- Something people want to use.
- Prototype it early, watch users!