Reuse

So far, we've assumed that the developer is working in a vacuum.
In a realistic current project, a company will have a **base of existing software** to leverage.
Also, one commonly faces **make/buy tradeoffs** for software components.
- Do we write the software ourselves,
- Or do we buy it from others?
- Or do we get it for free from open source repositories?

Goals of this lecture

Understand the make/reuse decision in detail.
Learn how to incorporate software reuse into design.
Why lecturing on software engineering is difficult: Lots of the field is about "stating the obvious". In this lecture, I am going to "state the obvious" many times.

But I am saved from complete oblivion and boredom by one crucial fact:
Many things that are obvious once stated are not obvious until stated!

This is "Couch's first rule of visualization"...!
A few obvious motivations...

- **Changing code is more difficult than writing it**
  - especially if you are *not* the original author, or
  - even if you *are* the original author and *others* have modified it!

  So we can forget about rewriting existing code as a reuse strategy. It's almost never practical.

- **Writing reusable code costs nontrivial money**
  (compared to writing code that is not constructed to be reused).
  - Must think beyond the current projects toward other future projects.
  - Must document not just capabilities, but also limits and shortcomings.

- **But there is high motivation for using reusable code wherever possible.**
  - It saves effort and reduces time to market.
  - Existing reusable code has a known cost (compared to the cost of writing it).
  - So, your cost of using reusable code is its cost plus your cost to adapt your code to use it.

- Any reuse that is going to occur should be **part of the design**, and not left to the implementation.
**Reusable:** can be employed to accomplish previously unforeseen tasks.

**Portable:** can run properly in various architectures and/or environments.

Portable does not imply reusable, nor does reusable imply portable.
Kinds of reusable software:

**Libraries:** collections of subroutines that accomplish specific, cohesive tasks

**Programs:** standalone software components that can be called like subroutines to accomplish specific tasks.

**Services:** running programs on remotely located hosts that answer requests and provide responses.

**Patterns:** documentation of the shape of code required to solve a specific problem, with or without language support.
Attributes of good reusable code

Monday, September 24, 2012
2:48 PM

Documented
Scope: what it does.
Limits: what it doesn't do.

Maintained/managed
Tested: that it does what is claimed.
 Managed:
 Releases occur on predictable schedules.
 Deprecations are announced in advance.
A practical story: my android phone

- Every once in a while, a **new release of android** comes out.
- Installing it breaks about 1/2 of my applications.
- Every Sat morning, Google releases updates of, e.g., 4 or so applications and the phone asks me whether to load them. **I never say no.**
- Some of them work; some don't.
- Thus on average, about 1/4 of the applications loaded on my phone don't work properly!
- Including ones that **I paid for!**
Another typical reuse horror story

I used Perl's library HTML::Parser library to construct a grading page.

The developers changed the semantics of parsing.

I am a very busy person!

It took me a couple of years to adapt to the change, because I had not planned for something that ridiculous.

In the meanwhile, my 40,000-line grading package was useless.

"For want of a nail, a horseshoe was lost."
Some basic principles of reuse

Must **couple your code maintenance** to that of whatever reused code you employ. For commercial code, bug fixes are a **contractual obligation** of the vendor. Reuse of open-source software requires understanding **social dynamics**.
Coupling maintenance with reusable code
  o the user of reusable code is at the mercy of the implementor of the reusable code.
    ▪ the user must adapt uses to changes in the reused code.
    ▪ this leads to a lag between updates to the reusable code and the code that uses it.
    ▪ Often, social inertia is all that protects the user from ad-hoc changes.
Committing to reuse also commits one to lag behind developers of the reused code. maintain your code according to their development cycle.

An arms race:
MS doesn't have a strong "desire" for average developers to be able to manipulate, e.g., Word documents. So, every release, there is a "lag" between the MS release and the 3rd-party products that try to read MS-format files. Committing to interact with Word files is a commitment to slave your own development cycle to the software that reads them, which is slaved to the MS development cycle.
Coping with open-source reusable code

Not necessarily tested.
Features that everyone uses will work.
Features that no one uses are almost guaranteed not to work.
Choice of vanilla "configuration" is very important.
Design-centered reuse
design must adapt to pre-existing conditions.
much of the action is in choosing an architecture
in which reuse can occur.

**object-oriented**: objects and inheritance
input: existing base classes.
reuse model: inheritance/override
characteristics: works when desirable
behavior is "a kind of" existing behavior.

**component-based**: components and interfaces
input: existing components.
reuse model: precondition/postcondition
analysis.
characteristics: works well when parts of
desirable behavior are implemented in
existing components.

**service-oriented architecture**: services and clients
input: existing services (independent
programs running on distributed computers)
reuse model: schema-based verification of
input and output.
characteristics: freeform input and output.

**design patterns**: patterns and instantiations
input: "patterns" are templates for
components
reuse model: fill-in-the-blanks.
characteristics: avoid re-inventing the wheel for common algorithms and borrow correctness from the templates!
Some relatively obvious comments

It is much easier to **build a subclass** than to **change a class**.

The former involves only **verifying new behaviors** that will be used by **new clients**. The latter involves **reverifying the base class** and all of its uses. **Every user** of the class is at risk!

This is one form of **local-global argument** that arises repeatedly in forms of reuse:

It is easier to create a **new local behavior**, than to influence an **existing and potentially large base of uses**!
Two modern reuse enablers
Service-oriented architectures (today)
Design patterns (tomorrow)
Service-oriented architectures

Idea: programs are composed from pieces distributed on different servers.
Ideal application: maintenance of large corpora (databases).
Example: IP to GPS coordinates service:
   Caller provides an IP address
   Service returns Latitude/Longitude
More than software: google commits to update and curate the coporum.
SOA properties

loosely coupled with application
dependent upon internet
use **XSchemas** (or **REST**) to verify input and output
XSchema

Part of the eXtensible Markup Language (XML) standard.
Extension of XML document-type-definitions
Define acceptable content for a service request or response in XML.
Equivalent to a regular graph grammar with some extensions
What is an XSchema?

Basically, a very powerful way of expressing both preconditions and postconditions for a service. Limited to context-free assertions about structure of requests and responses, verifiable by pushdown automata.

Any data must match a regular expression pattern. Not turing-equivalent!

In other words, there are lots of things you might want to do in your input, e.g.

Foo:
  Item 1
  Item 2

Bar:
  Item 3
  Item 4

that are contextual (upon the label Foo and Bar) and thus cannot be represented. You must represent these as something like:

```xml
<Document>
  <Foo>
    <item number='1'/>
    <item number='2'/>
  </Foo>
  <Bar>
    <item number='3'/>
  </Bar>
</Document>
```
for an XSchema to apply to it and verify it.

For more details, see Comp120: Web Engineering.
Special properties of SOA

you do not have control of processing time
scalability is the responsibility of the service
main difficulty is performance prediction
Main challenge of SOA: **performance engineering**

**Service-Level Objective** (SLO): how fast you would like your application to perform, e.g., "response to customer in under 5 seconds"

This is about **what you want**.

**Service-Level Agreement** (SLA): how fast you agree to make your application for a **client**, e.g., "Response time to customer will be under 5 seconds."

This is about **what your customer requires**.

There is **no penalty to violating an SLO**, but if you **violate an SLA**, you might have to **pay the customer**.

An **SLA** is a **legal agreement**.
Example of simple SLA analysis

You are writing a service with an SLA of two seconds. You call another service twice, which has an average response time of 0.5 seconds. Thus

your response time
= your processing time
+ time spent waiting for services

So your average response time is whatever time you spend, plus 1 second.
Key tool in SLA/SLO analysis: expected value

Recall that the \textit{expected value} of a stochastic process is the sum of probabilities of outcomes times the values of the outcomes.

\[ E = \sum p_i v_i \]

\[ p_i = \text{probability of value } v_i \]
Suppose that you know that your service runs in constant time, and that it depends upon two calls to another service, where about 1/4 of the time, the call takes over one second and the rest of the time, it takes under one second.

Suppose that calls to the service are independent of one another (and depend, instead, on other factors, e.g., network load)

Then you can conclude that
about $1/16=1/4*1/4$ of the time, the calls will definitely take over 2 seconds to complete (two independent events).
about $3/4*3/4 = 9/16$ of the time, the calls will definitely take under 2 seconds.
and the rest is indeterminate.

Note that the problem of determining runtime and response time is typically under-determined.
The key factor in deciding whether a service architecture is valid: **cost/value tradeoffs**.

    An SLA you hold costs something to you.
    An SLA you provide gives you value, and has a penalty if you do not provide it.

(Monthly charges, or charges by request)
A key tool in understanding cost and value: decision trees branches represent options as to what happens. Each branch is labeled with a probability. Branches are **mutually exclusive**. Levels are **independent**.
Decision tree example:

You are required to return an answer to a user query within 1 second. You are paid $.10 for each query within that time, and $.00 for each query over that time. You calculate that you do this 90% of the time. What are you paid on average per query?

\[
\text{Expected value} = 0.9 \times 0.10 + 0.1 \times 0 = 0.09
\]
Multi-level decision trees: enumerate options for multiple entities.

Example: your behavior is as before. But there is another implementor who is paid $.15 for each query within 2 seconds, and $.25 for each query within 1 second, and $.00 otherwise. The probability of being within 2 seconds is .8, while the probability of being within 1 second is .6.

What is the average payoff for a query to both of you?
Cover weakest precondition principle and analysis, as well as antibugging strategies based upon this. Link to Xschemas for service development.