# Scripted Components

## Software Architecture, Process, and Management

In the previous lecture I detailed the purpose of architectural patterns. Recall that architectural patterns can be combined. In this lecture I detail an important architectural pattern which can be combined with almost any other architectural pattern.

# Type and Behaviour

- Components rarely have matching interfaces
- Components may have a matching **type** interface but the behaviour or requirements interface is subtly different
- Hence existing components are difficult to re-use
- This leads to something of a vicious cycle, without the expectation of re-use, there is little incentive for making reusable components

# Rules of Three

- There are two "rules of three" in reuse:
  1. It is three times as difficult to build reusable components as single use components
  2. a reusable component should be tried out in three different applications before it will be sufficiently general to accept into a reuse library
- Of course these are "rules" in the sense of a *rule of thumb*

# Building Reusable Components

- Advantage: There can be some tangible benefit to making software components reusable, **even if** they are not ultimately reused
- Disadvantage: There can be a tangible disadvantage to such unnecessary generalisation
  - This is arguably more common
  - This is termed *"YAGNI"* for *"You aren't gonna need it"*

# Replaceable Components

- Many undergraduate software development courses, teach the idea of *encapsulation* or *hiding* usually behind an interface
- The idea is that the user of a component should not rely on the **implementation** details of that component
- Where this is true, then the component can be substituted out for one which is implemented differently, but has the same interface
- This can work well for some very well defined problems, such as sorting, dictionaries, matrices etc.
- Great for the platform dependent parts
- For less well studied problems, it is less clear you would ever want to swap out the implementation

# Background: Types of Typing

- Languages differ about enforcing types at run or compile time:
  - Static typing:
    - Types of objects known at compile time
    - Type of each object not usually stored in executable
    - Examples: C and derivatives, Java, Haskell
  - Dynamic typing:
    - Types of objects known only at run time
    - Each object stores its type
    - Examples: Python, Ruby, LISP

# Background: Types of Typing

- And on how strongly they enforce types:
  - Strong typing:
    - Objects of the wrong type cause failures
    - e.g. `3+"5"= error`
    - Examples: Java, Python, ML, Haskell
  - Weak typing:
    - Objects of the wrong type are converted
    - e.g. `3+"5"="35"` or `3+"5"= 8`
    - Examples: JavaScript, PHP, Perl5, Tcl

# Background: Types of Typing

- And on whether typing is declared explicitly or implicitly:
  - Nominative typing: Name of declared type must match
    - checkable at compile time; C++ (except templates), Java
  - Structural typing: Full set of methods must match (Haskell)
  - Duck typing: *"If it waddles and quacks like a duck, it's a duck"*
    - C++ templates (e.g. iterators and pointers), Python, Ruby
- Different type systems offer vastly different levels of flexibility, optimisation capability, speed, memory usage, etc.
- For more info, see many places but a good start is **the wikipedia page**

# Types vs Tests

- Types can restrict legitimate uses of a particular value/method
- Tests can only detect the presence of bugs not their absence
- Types can additionally impose complexity onto what could be otherwise simple code
- Testing code is yet more code that must be maintained
- My own take is that types are useful, but we should not try to contort the type system to ensure properties for which it is inappropriate
- Tests should be used to ensure/enforce properties which we would like to ensure with some mythical type system/program analysis but are unable to

I have similar thoughts on assertions and exceptions, i.e. they should be used in place of a non-perfect type system

# Requirements for Building Components

- To implement useful, high-performance primitives, you typically need:
  - Speed
  - Memory efficiency
  - Bit-level access to underlying hardware and OS
- Where this is not the case, you can build the whole thing in a high-level language so you do not have such a problem

# Systems Languages

- The requirements for building components are met by systems languages such as C
- Systems languages allow, and typically require, detailed control over program flow and memory allocation
- With such power available, strong, static, nominative typing (declaring object and argument types, checking them at compile time against strict inheritance hierarchies, etc.) is necessary to prevent catastrophic errors
- Of course, these requirements depend entirely on the system that you are creating, for example performance might not be a major concern

# Requirements for Gluing Components

- To glue primitive components written by multiple independent developers into an application, you typically want:
  - Non-nominative typing, to allow different interfaces to connect freely
  - A small number of high-level, widely shared interface datatypes (e.g. strings, lists, dictionaries, basically that provided by JSON)
  - Automatic memory management, etc., to allow one-off data structures to be created easily for gluing
  - Graceful user-relevant error handling, debugging

# Scripting Languages

- Gluing components from independent developers in systems languages requires huge amounts of code and much time debugging, often swamping any benefit of reuse
- Scripting languages excel at gluing, because they insulate the user from the details of program flow, memory allocation, and the operating system
- Scripting languages are good for manipulating (analysing, testing, printing, converting, etc.) pre-defined objects and putting them together in new ways without having to worry much about the underlying implementation

# Scripting Language Features

- Typically:
  - Interpreted: for rapid development and user modification
  - High-level: statements result in many machine instructions
  - Garbage-collected: to eliminate memory allocation code and errors
  - Dynamically typed: to simplify gluing
  - Slow: for native code (but can use fast external components)
  - Examples: Python, Perl, Scheme/LISP, Tcl, Ruby, Visual Basic, sh/bash/csh/tcsh

# Scripted Components

Problem Solution Advantages Liabilities

- Use a scripted language interpreter to glue reusable components together, packaging an application as:
  - An interpreter
  - A component library (preferably mostly pre-existing)
  - Scripts to coordinate the components into a meaningful system
- Applications can be tailored to specific tasks by modifying the script code, potentially by end users
- Configuration options can be saved within the scripting language itself
  - See Eric S. Raymond's **report on configuration for fetchmail in Python: http://www.linuxjournal.com/article/3882**

# Scripted Components

Problem Solution Advantages Liabilities

- Helps make maintaining a large code body practical
- Increases long-term maintainability because the application can be reconfigured as needs change
- Promotes reuse, and thereby development of reusable components
- Provides separation between high-level and low-level issues
- Greatly reduces total size of code, and/or expands functionality

# Scripted Components

Problem Solution Advantages Liabilities

- Can be complicated to bind languages together (SWIG, weave)
- Must learn and maintain source code in multiple languages
- Can be slow if critical components are mistakenly implemented in the script language
- Largest benefit requires existing components
  - but see e.g. the huge Python, Ruby and Perl standard libraries

# Scripted Components Example: Emacs

- Core rarely-changed code written in C (265 KLOC), implementing custom LISP interpreter and performance-critical components
- Rest in LISP (580 KLOC + huge external codebase), most of it user-contributed (i.e., written independently)
- Maintained continuously for more than 30 years, by hundreds (possibly thousands) of people

# Scripted Components Example: Git

- At its heart, Git is really a serialised data structure representing a code repository
- It is supported by command-line tools to manipulate that data structure
- These are often separated into two groups:
  - The plumbing: low-level commands that enable basic content tracking and the manipulation of directed acyclic graphs
  - The porcelain: commands that most Git end users are likely to need to use for maintaining repositories and communicating between repositories for collaboration

# Custom vs Off-the-shelf

- Most existing large, long-lived programs use custom languages for macros or configuration, but those are:
  - hard to maintain
  - hard to learn
  - not shared between programs (limiting reuse), and
  - limited in functionality
- Modern approach: Plug-in scripting languages
  - Many now available freely, with large bodies of reusable component libraries. Just download one and get to work!
  - E.g. Python, Guile (Scheme), Tcl, Perl

# Eager Optimisation vs Lazy Optimisation

- Eager Optimisation: is the act of optimising a piece of code before it is known whether it requires optimisation
- Like eager evaluation, evaluates all expressions whether the value is later used or not
- Lazy Optimisation: only optimises code when it is known to be causing unacceptable performance degradation
- Judging by the student coursework I grade, eager optimisation is a temptation difficult to resist