CSE584: Software Engineering

Lecture 5: Design (B)

David Notkin
Computer Science & Engineering
University of Washington
http://www.cs.washington.edu/education/courses/584/

Design lectures

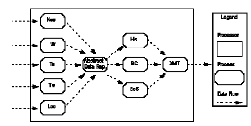
- **Last week**
  - **Basic issues in design, including some historical background**
  - **Well-understood techniques**
    - **Information hiding, layering, event-based techniques**
- **This week: neo-modern design**
  - **Problems with information hiding (and ways to overcome them)**
  - **Architecture, patterns, frameworks**

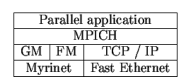
Software architecture

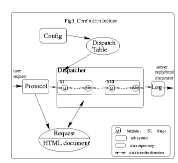
- **An area of significant attention in the last decade**
  - **Garlan and Shaw**
  - **Perry and Wolf**
- **There are two basic goals (in my opinion)**
  - **Capturing, cataloguing and exploiting experience in software designs**
  - **Allowing reasoning about classes of designs**

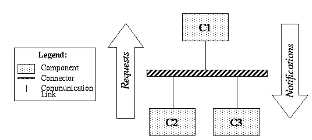
Box-and-arrow diagrams:

taken from the web without attribution









These diagrams

- **Clearly, these diagrams give value**
  - **You can find them all over the web, in textbooks, in technical documents, in research papers, over whiteboards in your office, on napkins in the cafeteria, etc.**
- **At the same time, they are generally ill-defined: what does a box represent? an arrow? a layer? adjacent boxes? etc.**
- **One view of software architecture research is to determine ways to give these diagrams clearer semantics and thus additional value**

An aside: compilers I

- **The first compilers had *ad hoc* designs**
- **Over time, as a number of compilers were built, the designs became more structured**
  - **Experience yielded benefits**
    - **Compiler phases, symbol table, etc.**
  - **Plenty of theoretical advances**
    - **Finite state machines, parsing, ...**

An aside: compilers II

- Compilers are perhaps the best example of shared experience in design
  - Lots of tools that capture common aspects
  - Undergraduate courses build compilers
  - Most compilers look pretty similar in structure
- But we still don't fully generate compilers
  - Despite lots of effort and lots of money
  - In any case, the code in compilers is often less clean than the designs
- Despite this, the perception of a shared design gives leverage
  - Communication among programmers
  - Selected deviations can be explained more concisely and with clearer reasoning

Other domains?

- Which other domains are as successful in this regard as compilers?
- Quite a few, but generally much more narrow
  - DARPA ran a large project, Domain-Specific Software Architectures (DSSA) a few years ago
    - ISI: Command and control message processing
    - Honeywell: Guidance, navigation and control
    - …
  - Some 4GL approaches are basically domain-specific systems
- Essentially: (Parnas) program families in which systems have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them"
  - His OS example is tempting but has not really come to fruition

Back to software architecture

- One hope is that by studying our experiences with a variety of systems, we can gain leverage as we did with compilers
- Capture the strengths and weaknesses of various software structures
  - Perhaps enabling designers to select appropriate architectures more effectively
- Benefit from high-level study of software structure

- D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (Oct 1992)
- D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (ed.), *Advances in Software Engineering and Knowledge Engineering* (1993).

Another motivation: architectural mismatch

- Garlan, Allen, Ockerbloom tried to build a toolset to support software architecture definition from existing components
  - OODB (OBST)
  - graphical user interface toolkit (Interviews)
  - RPC mechanism (MIG/Mach RPC)
  - Event-based tool integration mechanism (Softbench)
- It went to hell in a handbasket, not because the pieces didn't work, but because they didn't fit together
- Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software 12*, 6 (Nov. 1995). Best paper of the year in *IEEE Software*.

Mismatches included

- Excessive code size
- Poor performance
- Needed to modify out-of-the-box components (e.g., memory allocation)
- Error-prone construction process
- …

So what?

- The claim is that many of the problems were of an architectural nature
  - What assumptions are made, need they be made, etc.?
- With some forethought, many of these mismatches could, in principle, be avoided

Some classic definitions:
http://www.sei.cmu.edu/architecture/definitions.html

- …architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and the interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design [Perry and Wolf].
- An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition [Booch, Rumbaugh, and Jacobson, 1999]

More definitions

- ...beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives [Garlan and Shaw].
- The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time [Garlan and Perry].
- ...an abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections [Hayes-Roth].
- …

Components and connectors

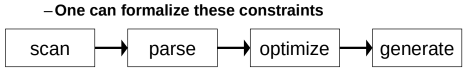
- (Most people now agree that) software architectures includes *components* and *connectors*
- *Components* define the basic computations comprising the system
  - Abstract data types, filters, etc.
- *Connectors* define the interconnections between components
  - Procedure call, event announcement, asynchronous message sends, etc.
- The line between them may be fuzzy at times
  - Ex: A connector might (de)serialize data, but can it perform other, richer computations?

Architectural style

- Defines the vocabulary of components and connectors for a family (style)
- Constraints on the elements and their combination
  - Topological constraints (no cycles, register/announce relationships, etc.)
  - Execution constraints (timing, etc.)
- By choosing a style, one gets all the known properties of that style
  - For any given architecture in that style
- These properties can be quite broad
  - Ex: performance, lack of deadlock, ease of making particular classes of changes, etc.

Not just boxes and arrows

- Consider pipes & filters, for example (Garlan and Shaw)
  - Pipes must compute local transformations
  - Filters must not share state with other filters
  - There must be no cycles
- If these constraints are not satisfied, it's not a pipe & filter system
  - One can't tell this from a picture
  - One can formalize these constraints



WRIGHT

- WRIGHT provides a formal basis for architectural description (ADL = architectural description language)
- Language for precisely defining an architectural specification, as a basis for analyzing the architecture of individual software systems and families of systems
- Underlying model in CSP (communicating sequential process, Hoare), checkable using standard model checking technology
  - Defines a set of standard consistency and completeness checks
- More on WRIGHT
  - Allen and Garlan. A formal basis for architectural connection. *ACM TOSEM 6*, 3 (1997)

Defining a connector in WRIGHT: client-server

```
connector C-S-connector =

   role Client = (request!x → result?y → Client) ∏ §

   role Server = (invoke?x → return!y → Server)   §

   glue =
         (Client.request?x → Service.invoke!x →
          Service.return?y → Client.result!y → glue)
  §
```

Pipe connector in WRIGHT

```
Connector Pipe =
   role Write = write → Writer ∏ close → √
   role Reader =
      let ExitOnly = close → √
      in let DoRead =
           (read → Reader   read-eof → ExitOnly)
      in DoRead   ExitOnly
   glue = let ReadOnly =
           Reader.Read → ExitOnly
           Reader.read-eof → Reader.close → √
           Reader.close → √
```

- **Ensures (among other things) that there is a way to notify reader than pipe is empty when writer closes the pipe**

Decoding a little bit

- **Connectors represent links to components on the roles, which are ports of the connectors**
  - **The WRIGHT process descriptions describe the obligations of each connector**
- **The glue process coordinates the behavior of the roles**
  - **Essentially, it defines a high-level protocol**
- **One can then prove properties about the stated protocols**

Benefits

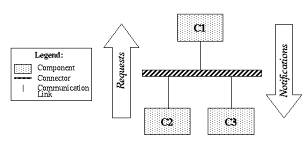
- **In the pipes & filters example, the constraints ensure a lack of deadlock**
  - **In any instantiation of the style that satisfies the constraints**
- **One can think of the constraints as obligations on the designer and on the implementor**
  - **Some properties can be automatically checked**

Specializations

- **Architectural styles can have specializations**
  - **A pipeline might further constrain an architecture to a linear sequence of filters connected by pipes**
  - **A pipeline would have all properties that the pipe & filter style has, plus more**

C2 Architecture:
UC Irvine (Taylor et al.)

- **Based on generalization of a collection of designs of user interface systems**
- **Informally, a C2 architecture is a network of concurrent components linked together by connectors**
- **http://www.ics.uci.edu/pub/c2/c2.html**



C2 Composition

- The top of a component may be connected to the bottom of a single connector
- The bottom of a component may be connected to the top of a single connector
- There is no bound on the number of components or connectors that may be attached to a single connector
- When two connectors are attached to each other, it must be from the bottom of one to the top of the other

C2 Communication

- Solely by exchanging messages
- Each component has a top and bottom domain
  - The top specifies the set of notifications to which a component responds, and the set of requests it emits upwards
  - The bottom specifies the set of notifications that a component emits downwards and the set of requests to which it responds
- Central principle: limited visibility (substrate independence)
  - A component within the hierarchy can only be aware of components "above" it and is completely unaware of the components "beneath" it

Well, do they help?

- I like the basic software architecture research as an intellectual tool
  - The work is helping us better understand classes of software structures that have shown themselves as useful
  - Simply improving our shared terminology is a benefit
- It may not be fully distinct from Parnas' families of systems, but enough to benefit

Open question I

- What properties can be analyzed?
  - WRIGHT [Allen & Garlan]
    - Reason about architectures in terms of protocols, using a CSP-like language
    - Roughly, type-checking of architectural styles
  - Of these, which are sufficiently important to justify the investment
    - The investment is high, but in theory amortized
  - What about across heterogeneous architectures?

Open question II

- How does one produce new architectural styles?
- When?

Open question III

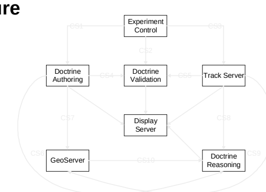
- What is the relationship between architectural and implementation?
  - Does architectural information aid in going from design to implementation?
  - What happens as the implementation evolves in ways inconsistent with the architecture?
    - Which properties still hold, and how do we know this?

Experience

- It’s a hot area, with lots of companies paying attention
- Allen & Garlan reported on a case study in applying architectural modeling to the AEGIS Weapons System
  - Used formalism to help “expose and resolve some of the architectural problems that arose in implementing the system”
- Similar advantages for the HLA project
  - Distributed simulation for the DoD

AEGIS

- AEGIS Weapons System, control of US Navy ships
  - Model problem for work in software architecture



Example benefits in AEGIS

- Clarifying client-server misconceptions
  - Which party initiated interactions?
  - Re-established after every request?
  - Synchronous or asynchronous?
- WRIGHT used to clarify
  - Avoiding deadlocks
  - Reducing unnecessary synchronization
  - And to simplify instrumentation of the architecture

Forcing discussions

- In some ways, the primary benefit of architecture Garlan is that it forces discussions of some critical issues
  - The Xerox PARC Mesa/Cedar group did roughly the equivalent by spending enormous amounts of times in defining and clarifying interfaces, before coding
- Finding errors earlier is generally considered to be better, of course
- I’m unsure the degree to which the formalism per se helps, although there are surely some supporting examples

On-going research

- Environments to support the design of architectural styles and architectures
- Architectural design languages (ADLs)
- Formal models of architectures
- Architectural case studies
- Use of informal architectures
- ...

Design patterns

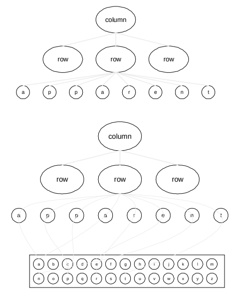
- Design patterns are idioms that are intended to be *“simple and elegant solutions to specific problems in object-oriented software design.”*
- They are drawn from actual software systems
- They are intended to be language-independent

A weak analogy

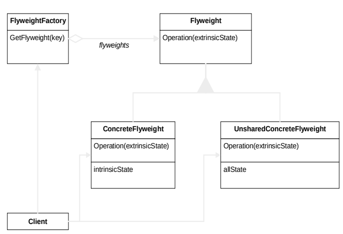
- I view high-level control structures in programming languages as quite the same
  - For example, a while loop is an idiomatic collection of machine instructions
- Knuth's 1974 article ("Structured Programming with go to Statements") shows that this is not a language issue alone
- Patterns are a collection of "mini-architectures" that combine structure and behavior

Example: flyweight **[Gamma et al.]**

- **Intent**
  - Use sharing to support many fine-grained objects efficiently
  - Can't usually afford to have small elements (like characters) be full-fledged objects
- **Separate logical model from physical model**



Flyweight structure



An enlightening experience

- At a workshop a several years ago, I had an experience with two of the Gang of Four
- They sat down with Griswold and me to show how to use design patterns to (re)design a software design we had published
- The rate of communication between these two was unbelievable
  - And much of it was understandable to us without training (good sign for a learning curve)

This is the real thing

- Design patterns are not a silver bullet
- But they are impressive, important and worthy of attention
- I think that (slowly?) some of the patterns will become part and parcel of designers' vocabularies
  - This will improve communication and over time improve the designs we produce
- The relatively disciplined structure of the pattern descriptions may be a plus

The future

- I'm somewhat worried that "second wave" R&D will hurt more than help
  - They may be considered a panacea
  - They are surely going to be misunderstood
    - Everything now is a "pattern", even if it doesn't have the key characteristics
    - There are even antipatterns
  - Tools and languages for patterns may help, but may also hinder
- How do patterns interact?

Patterns resources

- **Patterns Home Page**
  - http://st-www.cs.uiuc.edu/users/patterns/patterns.html
- **Portland Pattern Repository**
  - http://c2.com/ppr/index.html
- **FAQ**
  - http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html
- **Gang of Four book**
  - Design Patterns: Elements of Reusable Object-Oriented Software. Gamma et. al. (as of today @ 11:05AM PT, Amazon sales rank of 111, 88 reviews)
- **OO journals, OOPSLA, etc.**

Do any of you use patterns?

Frameworks

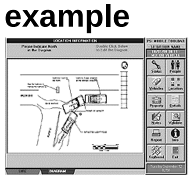
- Frameworks are another design buzzword
- One way to think about them is as upside-down layers
  - That is, layered systems allow us to construct families of systems by sharing lower layers
  - Frameworks allow us to construct families of systems by sharing upper "layers"
- Instantiate and specialize provided classes
  - "More" than patterns

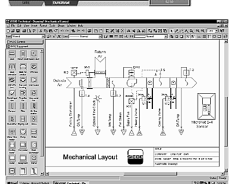
Examples

- DuPont's business model
  - `http://www-cat.ncsa.uiuc.edu/~yoder/Research/catdesc.html`
  - Visual table-based framework for improving financial decisions, etc.
- CHOICES: customizing operating systems
  - `http://choices.cs.uiuc.edu/choices/choices.html`
  - Frameworks for VM, memory management, process management, file storage, exceptions and hardware device drivers, distributed processing and communication

A commercial example

- Visio is in many ways a framework
- It is also a complete application on its own, but it can be specialized (in a number of ways) that is consistent with being a framework





Open implementation

- Last week in discussing information hiding I listed some central premises
- Two important ones are especially questionable
- Kiczales et al. have studied this question carefully, leading to some work generally called Open Implementation
  - `http://www.parc.xerox.com/spl/projects/oi/`

Central premises III and IV:
from last week

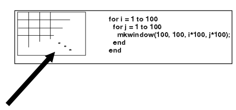
- The semantics of the module must remain unchanged when implementations are replaced
  - Specifically, the client should not care how the interface is implemented by the module
- One implementation can satisfy multiple clients
  - Different clients of the same interface that need different implementations would be counter to the principle of information hiding
    - Clients should not care about implementations, as long as they satisfy the interface

These are often false

- What defines the semantics of the interface?
  - Much is not (cannot?) be defined, but is inferred by the client
- Once properties are inferred, clients start to assume that they are true
- Multiple clients may infer different properties
  - So changing those properties consistently may be impossible
- Client do, in practice, care about (aspects of) the implementation

Examples

- The flyweight pattern example points out a few of these issues
- Logically, any implementation of the interface is OK
  - But not all implementations are equally adequate for all clients
- The Kiczales spreadsheet example



Two approaches often taken

- Programmers often respond to these problems in one of two ways
  - Write own windowing system
  - Clever coding tricks
    - Paging example

The experts say

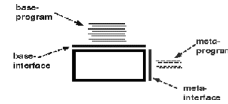
"I found a large number of programs perform poorly because of the language's tendency to hide `what is going on' with the misguided intention of `not bothering the programmer with details'" [Wirth, 1974]

"An interface should capture the *minimum* essentials of an abstraction.

"When an interface undertakes to do too much, the result is a large, slow complicated implementation." [Lampson, 1984]

The OI solution

- Define two interfaces
  - The *base interface*, which provides the essential semantics
  - The *meta-interface*, which is used to customize aspects of the implementation of the base
- Based on experience
  - Common Lisp Meta-Object Protocol (CLOS MOP)
  - Reflective computing



Allows the client to

- Use the module's primary functionality alone when the default implementation is adequate
- Control the module's implementation-strategy decisions when necessary
- Deal with functionality and implementation strategy decisions in largely separate ways

Design issues: OI claims

- The base interface design requires similar techniques to current interface design
- The design of the meta-interface and of the coupling of the meta- and base interface is more complicated
  - Requires expertise in the definition and uses of the components

Design issues: meta-interface

- Scope control
  - Are controls over the implementation for instances, classes, other?
- Conceptual separation & incrementality
  - Can the client of the meta-interface understand and use just parts of it?
- Robustness
  - Are bugs in a client's meta-program limited in effect?

It's not an entirely new idea

- Compiler pragmas
- Multiple implementations of an interface
  - With client choice [Hermes]
- User-directed parallelization
- Unix `madvise`
  - Influence page replacement
- Many more

More recently

- Examples
- Design guidelines
- Analysis techniques
- Aspect-oriented programming, an outgrowth of the work in OI (and some other stuff)
  - We'll breeze through some slides on AOP from Xerox PARC
  - There's a lot more work since this overview (1997-98)
    - Kiczales @ UBC now
    - aspectj.org
    - IBM Research: Multi-dimensional separation of concerns

Recap

- Software architecture
  - Heavy-weight design, with an eye towards ensuring specific properties over families of systems
- Patterns
  - Mini-architectures, allows effective chunking of small combinations of classes/objects
- Frameworks
  - Sharing the "top" of a family of applications (as opposed to the bottom, like in layering)
- Open implementation/AOP
  - Overcoming problems in separation of concerns

Next two weeks

- Software evolution, reverse engineering, etc.