CSE P503: Principles of Software Engineering

David Notkin
Spring 2009

Tonight's agenda

- Software design: information hiding and layering
- Discussion: software disasters – technical, managerial, or otherwise … and what can we and should we do about it?
- Software design: a simple example, patterns, architecture
- Optional one-minute paper

Functional decomposition

- Divide-and-conquer based on functions
  - **`input; compute; output`**
- Then proceed to decompose compute
- This is stepwise refinement (Wirth, 1971)
  - In essence, refining until implementable directly in a programming language (or on an architecture)
- There is an enormous body of work in this area, including many formal calculi to support the approach
  - Closely related to proving programs correct
- More effective in the face of stable requirements

Information hiding

- A very common term in software design
- What do you think it is?

Groups of 3-4

Information hiding

- Information hiding is perhaps the most important intellectual tool developed to support software design [Parnas 1972]
  - *Makes the anticipation of change a centerpiece in decomposition into modules*
- Provides the fundamental motivation for abstract data type (ADT) languages
  - And thus a key idea in the OO world, too
- The conceptual basis is key

Basics of information hiding

- Modularize based on anticipated change
  - Fundamentally different from Brooks' approach in OS/360 (see old and new MMM)
- Separate interfaces from implementations
  - Implementations capture decisions likely to change
  - Interfaces capture decisions unlikely to change
  - Clients know only interface, not implementation
  - Implementations know only interface, not clients
- Modules are also work assignments

Anticipated changes

- The most common anticipated change is "change of representation"
  - Anticipating changing the representation of data and associated functions (or just functions)
  - Again, a key notion behind abstract data types
- Ex:
  - Cartesian vs. polar coordinates; stacks as linked lists vs. arrays; packed vs. unpacked strings

Information hiding: issues

- Can we effectively anticipate changes?
- What is the underlying cost model and is it reasonable?
- The semantics of the module remain unchanged when implementations are changed: the client should only care if the interface is satisfied
  - But what captures the semantics of the module? The signature of the interface? Performance? What else?
- One implementation should satisfy multiple clients, which should only care if the interface is satisfied

Representation change less common

- We have significantly more knowledge about data structure design than we did 25 years ago
- Memory is less often a problem than it was previously, since it's much less expensive
- Therefore, we should think twice about anticipating that representations will change
  - This is important, since we can't simultaneously anticipate all changes

Other anticipated changes?

- Information hiding isn't only ADTs
- Algorithmic changes
  - (These are almost always part and parcel of ADT-based decompositions)
  - Monolithic to incremental algorithms
  - Improvements in algorithms
- Replacement of hardware sensors
  - Ex: better altitude sensors
- …

Best to change implementation?

- Usually, perhaps, but not always the lowest cost
- Changing a local implementation may not be easy
- Some global changes are straightforward: mechanically or systematically
- Rob Miller's simultaneous text editing
- Bill Griswold's work on information transparency

Information hiding reprise

- It's probably the most important design technique we know
- And it's broadly useful
- It raised consciousness about change
- But one needs to evaluate the premises in specific situations to determine the actual benefits (well, the actual potential benefits)

Dependence on implementation

- Gregor Kiczales et al.: clients indeed depend on some aspects of the underlying implementations in a broad variety of domains and situations
- What happens when the implementation strategy for a module depends on how it will be used?
- Aren't we supposed to separate policy from mechanism?
- Example: spreadsheet via many small windows?

```
for i = 1 to 100
  for j = 1 to 100
    mkwindow(100, 100, i*100, j*100);
  end
end
```

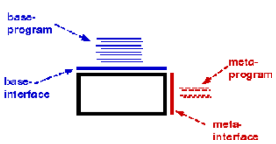

Poor performance often leads to…

"hematomas of duplication"

"coding between the lines"

Open implementation

- Decompose into base interface (the "real" operations) and the meta interface (the operations that let the client control aspects of the implementation)
- Arose from work in (roughly) reflection in the Meta-Object protocol (MOP) and led to the development of aspect-oriented programming



Meta interface examples

- C's **`register`** storage class
  - "A declaration of an identifier for an object with storage-class specifier **`register`** suggests that access to the object be as fast as possible."
- Unix **`nice`**
- High-Performance Fortran
  -
```
      REAL A(1000,1000),B(998,998)
!HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
!HPF$ DISTRIBUTE A(*,BLOCK)
```
- …many many more! Quick examples from you?

Information Hiding and OO

- Are these the same? Not really
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches)
  - Not necessarily based on change
- But they are obviously related (separating interface from implementation, e.g.)
- What is the relationship between sub- and super-classes?

Layering [Parnas 79]

- A focus on information hiding modules isn't enough
- One may also consider abstract machines
  - In support of program families, which are systems that have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them"
- Still a focus on anticipated change

The **`uses`** relation

- A program **`A`** uses a program **`B`** if the correctness of **`A`** depends on the presence of a correct version of **`B`**
- Requires specification and implementation of **`A`** and the specification of **`B`**
- Again, what is the "specification"? The interface? Implied or informal semantics?

`uses` vs. **`invokes`**

- These relations often but do not always coincide
- Invocation without use: name service with cached hints
- Use without invocation: examples?

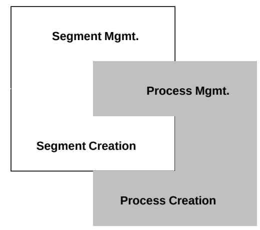
```
ipAddr := cache(hostName);
if wrong(ipAddr,hostName) then
   ipAddr := lookup(hostName)
endif
```

Parnas' observation

- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system
- So, it is important to design the **`uses`** relation using these criteria
  - **`A`** is essentially simpler because it uses **`B`**
  - **`B`** is not substantially more complex because it does not use **`A`**
  - There is a useful subset containing **`B`** but not **`A`**
  - There is no useful subset containing **`A`** but not **`B`**

Modules and layers interact?

- Information hiding modules and layers are distinct concepts
- How and where do they overlap in a system?



Imprecision in design discussions

- Not all boxes in a design are the same thing
- Not all arrows in a design are the same thing
- Imprecision in communication about these boxes and arrows can add significant confusion to a software design process and the resulting design
- Oh, that's the issue of clarity again
  - We'll return to this

Language support?

- We have lots of language support for information hiding modules
  - C++ classes, Ada packages, etc.
- We have essentially no language support for layering
  - Operating systems provide support, primarily for reasons of protection, not abstraction
  - Big performance cost to pay for "just" abstraction
- General observation: design ideas not encoded in a language are less likely to be used

Software disasters

- Technical, managerial, or otherwise?
- And what can we and should we do about it?
- What is *our* responsibility and how can *we* reduce the frequency and consequences of such problems?

Class discussion

A simple system: how to design?

- Consider two sets of integers, A and B
- How would we design a system to ensure that A and B always had the same elements?

`new()`	`{1, 17, 41, -5, 33333333, 0, 5247695,1666}`
`insert(int)`	
`delete(int)`	
`isMember()`	

Class discussion

Key points include…

- Can separate relationship from the base entities – very much like entity-relationship design in databases with the addition of behavior
- Need event-based mechanism
- Separate **`name`** and **`invoke`** relationships
  - Registration of interest is still an issue
- Aside: not all event mechanisms are created equally or used equally – ordering, circularities, etc. tend to rear their ugly head in many situations

Design patterns

- What are they?
- Do you use them?
- Do you like them?

Class discussion

Design patterns

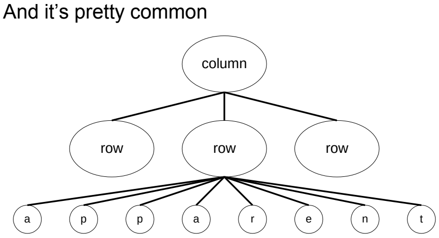
- "a 'well-proven generic scheme' for solving a recurring design problem"
  - Often overcoming limitations of OO hierarchies
- Idioms intended to be "simple and elegant solutions to specific problems in object-oriented software design"
  - Patterns are a collection of "mini-architectures" that combine structure and behavior
- They are drawn from examples in existing systems
  - Not proposed solutions to possible problems, but real solutions to real problems

They are an example of chunking

- Advanced chess players are in part superior because they don't see each piece individually
  - Instead, they chunk groups of them together
  - This reduces the search space they need to assess in deciding on a move
- This notion of chunking happens in almost every human endeavor
- Such chunking can lead to the use of idioms
  - As it has in programming languages
- The following slides show some parts of a particular pattern: flyweight
  - I won't go through the slides, but they give a feel for people who haven't seen more concrete information on patterns

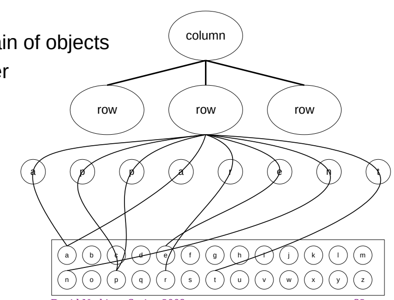
Example: flyweight pattern

- What happens when you try to represent lots of small elements as full-fledged objects?
- It's often too expensive
- And it's pretty common

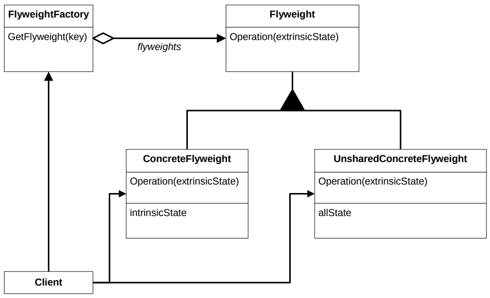


An alternative approach

- Use sharing to support many fine-grained objects efficiently
  - Fixed domain of objects
  - Maybe other constraints



Flyweight structure



Participants

- Flyweight (`glyph` in text example)
  - Interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight (`character`)
  - Implements flyweight interface, shareable, only intrinsic state (independent of context)
- UnsharedConcreteFlyweight (`row`, `column`)
- FlyweightFactory
  - Creates and manages flyweight objects

Sample code

```
class Glyph {
public:
  virtual ~Glyph();virtual
  void Draw(…);
  virtual void SetFont(…);
  …
}
class Character : public Glyph {
  Character(char);
  virtual void Draw(…);
private:
  char _charcode;
};
```

- The code itself is in the domain (glyphs, rows, etc.)
- But it's structured based on the pattern
- The client interacts with `Glyph`, `Character`

A little more code

```
Character* GlyphFactory::CreateCharacter(char c)
  {
  if (!_character[c]) {
     _character[c] = new Character();
  }
  return _character[c];
}
```

- Explicit code for each of the elements in the flyweight structure

An historical aside

- The Gang of Four loosely based their initial work on that of architect Christopher Alexander
  - Not a systems or software architect, but an architecture architect (with planning, too)
  - *The Timeless Way* trilogy
    - *The Timeless Way of Building* (1979), *A Pattern Language: Towns, Buildings, Construction* (1977), *The Oregon Experiment* (1975)
- Not surprisingly, a focus on idiomatic solutions to common design problems

A little more

- Alexander and his influence on CS
  - `www.math.utsa.edu/sphere/salingar/Chris.text.html`
- Too much can be (and is) made of the connection to Alexander
  - In particular, Alexander takes the “big” view of architecture and patterns
  - In software, it is important but still the “little” view

An enlightening experience

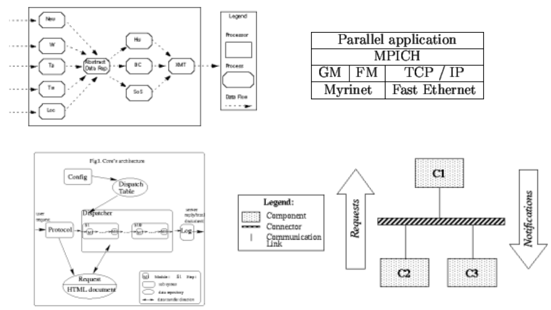
Design patterns: not a silver bullet…

- ..but they are impressive, important and worthy of attention and study
- I think that some of the patterns have and more 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

Software architecture

- An area of significant attention in the last decade or so
  - 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).
  - D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes 17*, 4 (Oct 1992).
- 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

Compilers

- 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, ...

Compilers

- 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

So…

- 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

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

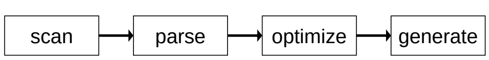
- (Most people now agree that) software architectures includes *components* and *connectors*
- *Components* define the basic computations comprising the system: abstract data types, filters, etc.
- *Connectors* define the interconnections between components: procedure call, event announcement, asynchronous 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 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

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

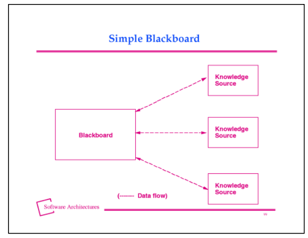
- In the pipes & filters example, the constraints ensure a lack of deadlock
  - In any instantiation of the style that satisfies the constraints
- One can think of the constraints as obligations on the designer and on the implementor
  - Some properties can be automatically checked

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 and filter style has, plus more

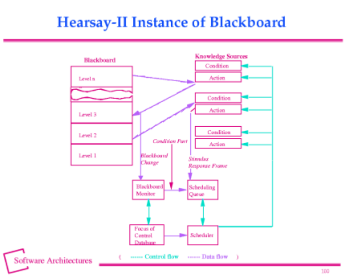
Blackboard architectures

- *The knowledge sources*: separate, independent units of application dependent knowledge. No direct interaction among knowledge sources
- *The blackboard data structure*: problem-solving state data. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
- *Control*: driven entirely by state of blackboard. Knowledge sources respond opportunistically to changes in the blackboard.



Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition.

Hearsay-II: blackboard



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

Open questions

- What properties can be analyzed?
  - Of these, which are sufficiently important to justify the investment: the investment is high, but in theory amortized
- How and when does one produce new architectural styles?
- What is the relationship between architectural and implementation?
  - Does architectural information aid in going from design to implementation?
  - What happens if and when the implementation evolves in ways inconsistent with the architecture?

Forcing discussions

- In some ways, the primary benefit of architecture 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 some supporting examples

Design questions/topics/insights?

Class discussion

Next week: aspect-oriented design

Optional…

- One-minute paper: Key point? Open question? Mid-course correction?