Software Design Alternatives and Examples

Douglas C. Schmidt
Professor
Department of EECS
Vanderbilt University
d.schmidt@vanderbilt.edu
www.cs.wustl.edu/~schmidt/

Deep Thought

According to C.A.R. Hoare, there are two methods of constructing a software system:
1. One way is to make it so simple that there are obviously no deficiencies
2. The other way is to make it so complicated that there are no obvious deficiencies

Introduction

• A key question facing software architects and designers is:
  – Should software systems be structured by actions or by data?
  – This decision cannot be postponed indefinitely
    * Eventually, a designer must settle on one or the other
    * Note, the source code reveals the final decision...

• Observation:
  – The tasks and functions performed by a software system are often highly volatile and subject to change

• Conclusion:
  – Structuring systems around classes and objects increases continuity and improves maintainability over time for large-scale systems
  – Therefore, “ask not what the system does: ask what it does it to!”

Outline

• This set of slides examines several alternative design methodologies
  – Primarily algorithmic/functional design vs. object/component-oriented design

• These alternatives differ in terms of aspects such as
  1. Decomposition and composition criteria
     – e.g., algorithms/functions vs. objects/components
  2. Support for reuse and extensibility, e.g.,
     – Special-purpose vs. general-purpose solutions
     – Tightly-coupled vs. loosely-coupled architectures
  3. Scalability
     – e.g., programming-in-the-small vs. programming-in-the-large
### Overview of Algorithmic Design

- Top-down design based on the *functions* performed by the system
- Generally follows a “divide and conquer” strategy based on functions
  - *i.e.*, more general functions are iteratively/recursively decomposed into more specific ones
- The primary design components correspond to processing steps in the execution sequence
  - Similar to a recipe for cooking a meal
    * Consider the objects and recipes used in cooking...

### Overview of Object-oriented Design

- Design based on modeling classes and objects in the application domain
  - Which may or may not reflect the “real world”
- Generally follows a “hierarchical data abstraction” strategy where the design components are based on classes, objects, modules, and processes
- Operations are related to specific objects and/or classes of objects
- Groups of classes and objects are often combined into frameworks

### Structured Design

- Design is based on data structures input and output during system operation
- Generally follows a decomposition strategy based on data flow between processing components
- Primary design components correspond to flow of data
  - Program structure is derived from data structure
  - Data structure charts show decomposition of input/output streams
- Often used as the basis for designing data processing systems
- Design tends to be overly dependent upon temporal ordering of processing phases, *e.g.*, initialize, process, cleanup
- Changes in data representations ripple through entire structure due to lack of information hiding

### Transformational Systems

- Design is based on specifying the problem, rather than specifying the solution
  - The solution is automatically derived from the high-level specification
  - Note, each transformation component may be implemented via other design alternatives
- Limited today to well-understood domains
  - *e.g.*, parser-generators, GUI builders, signal processing
Criteria for Evaluating Design Methods

• **Component Decomposability**
  – Does the method aid decomposing a new problem into several separate subproblems?
    * e.g., top-down algorithmic design

• **Component Composability**
  – Does the method aid constructing new systems from existing software components?
    * e.g., bottom-up design

• **Component Understandability**
  – Are components separately understandable by a human reader
    * e.g., how tightly coupled are they?

---

Case Study: Spell Checker Example

• **System Description**
  – ‘Collect words from the named document, and look them up in a main dictionary or a private, user-defined dictionary composed of words. Display words on the standard output if they do not appear in either dictionary, or cannot be derived from those that do appear by applying certain inflections, prefixes, or suffixes’

• **High-level Application Description**
  1. Get document file name
  2. Splits document into words
  3. Look up each word in the main dictionary and a private dictionary
     (a) If the word appears in either dictionary, or is derivable via various rules, it is deemed to be spelled correctly and ignored
     (b) Otherwise, the “misspelled” word is output

• **High-level Application Description**
  1. Get document file name
  2. Splits document into words
  3. Look up each word in the main dictionary and a private dictionary
     (a) If the word appears in either dictionary, or is derivable via various rules, it is deemed to be spelled correctly and ignored
     (b) Otherwise, the “misspelled” word is output

• **Pseudo-code algorithmic description**
  1. Get document file name
  2. Splits document into words
  3. Look up each word in the main dictionary and a private dictionary
     (a) If the word appears in either dictionary, or is derivable via various rules, it is deemed to be spelled correctly and ignored
     (b) Otherwise, the “misspelled” word is output

• **Note, avoid the temptation to directly refine the algorithmic description into the software architecture...**
Program Requirements

- Initial program requirements and goals:
  1. Must handle ASCII text files
  2. Document must fit completely into main memory
  3. Must run “quickly”
     - Note, document is processed in batch” mode
  4. Must be smart about what constitutes misspelled words (that’s why we need prefix/suffix rules and a private dictionary)

- Two common mistakes:
  1. Failure to flag misspelled words
  2. Incorrectly flag correctly spelled words

Data Flow Diagram

- While this diagram is useful for “describing” high-level flow of data and control, avoid the temptation to refine it into system design and implementation...

Algorithmic Design (1/2)

- Spell checker program is organized according to activities carried out during program execution
  - i.e., system is completely specified by the functions that it performs

- Function refinement precedes and guides data refinement

- Important questions:
  1. How is design affected by subsequent changes to the specification and/or implementation?
  2. How reusable are the algorithmic components developed via the approach?

Algorithmic Design (2/2)

- Top-down, iterative “step-wise” refinement of functionality:
  1. Break the overall “top” system function into subfunctions
  2. Determine data flow between these functions, then determine data structures
  3. Iterate recursively over subfunctions until implementation is immediate and “obvious”

- Structure chart shows function hierarchy and data flow
  - Hierarchical organization is a tree with one functional activity per node
Algorithm Design Structure

Design Alternatives

Advantages of Algorithmic Design

- Reasonably well-suited for small-scale, algorithmic-intensive programs
  - *e.g.*, Eight-Queens problem, Towers of Hanoi, 8-tiles problem, sort, and searching, *etc.*
- Easy to understand for small problems
  - Since system structure matches verbal, algorithmic description
- “Intuitive” to many designers and programmers
  - Due to emphasis in early training...

Disadvantages of Algorithmic Design

- Fails to account for long-term system evolution
  - *i.e.*, changes in algorithms and data structures ripple through entire program structure (and the documentation...)
  - Implementation often typified by lack of information hiding, combined with an over-abundance of global variables
    - These characteristics are not inherent, but are often related...
- Does not promote reusability
  - The design is specifically tailored for the requirements and specifications of a particular application
- Data structure aspects are often underemphasized
  - They are postponed until activities have been defined and ordered
Object-Oriented Design (1/2)

- Development begins with extensive domain analysis on the problem space
  - *i.e.*, OOD is not a “cookbook” solution
- Decompose the spell checker by *classes* and *objects*, not by overall processing *activities*
- Organize the program to hide implementation details of information that is likely to change
  - *i.e.*, use abstract data types and information hiding
- The order of overall system *activities* are not considered until later in the design phase
  - However, activities are *not* ignored!

Object-Oriented Design (2/2)

- At first glance, our object-oriented design appears to be incomplete since it does not seem to address the overall system *actions*...
- This is intentional, however, and supports the software design principle of “underspecification”
  - The goal is to develop reusable components that support a “program family” of potential solutions to this and other related problems
- In fact, the main processing algorithm may be quite similar in both algorithmic and object-oriented solutions...

Key Challenges of Object-Oriented Design

- A common challenge facing developers is finding the objects and classes
  - One approach: ‘Use parts of speech in requirements specification statements to’:
    1. Identify the objects
    2. Identify the operations and attributes
    3. Establish the interactions and visibility
  - This methodology is not perfect, but it is a good place to start...
    * *i.e.*, apply it at various levels of abstraction during development
- Another challenge is to ensure that the design can be mapped to an implementation that meets end-to-end QoS requirements

Classifying Parts of Speech

- Example: Spell Checker
  - *Collect words* from the named *document*, and *look them up* in a main dictionary or a private user-defined dictionary composed of *words*. *Display words* on the standard output if they do not appear in either dictionary, or cannot be *derived* from those that do appear by applying certain *inflections*, *prefixes*, or *suffixes*
- Relevant parts of speech:
  - *Common nouns* → classes
  - *Proper nouns* → objects
  - *Verbs* → actions on objects
Design Alternatives

Identifying Classes and Objects for the Spell Checker

- Common noun → class
  - *e.g.*, spell checker, dictionary, document, words, output
- Proper noun or direct reference → object
  - named document, main dictionary, private dictionary, standard output
- Describe using UML notation, CRC cards (“class, responsibility, collaborators”), C++ classes, etc.

Design Alternatives

Identifying Operations and Attributes for the Spell Checker

- Verb → operations performed on a class or by an object of a class
  - *e.g.*, collect (document), look up (dictionary), display (word)
- Adverb → constraint on an operation
  - *e.g.*, insert_quickly (*i.e.*, no range checking)
- Adjective → attribute of an object
  - *e.g.*, “large” dictionary → size field
- Object of verb → object dependencies
  - *e.g.*, “A dictionary composed of words”

Design Alternatives

Applying the Object-Oriented Method

- Visibility should satisfy dependencies and no more
  - In general, reduce global visibility, de-emphasize coupling, emphasize cohesion
  - In particular, Document and Dictionary shouldn’t be visible outside context of Spell_Checker...
- Develop a set of diagrams that graphically illustrate class, object, module, and process relationships from various perspectives

Design Alternatives

High-level Class Diagram

![Diagram of class relationships](image-url)
Design Alternatives

General Class Descriptions (cont'd)

- Building block classes (cont'd)

  NAME | Dictionary
  QUALIFICATIONS | Abstract class
  ACCESS | Exported
  CARDINALITY | Unlimited
  MEMBERS | construct/destruct
    open/close
    insert word
    find word
    remove word
    next word iterator
    ...

  NAME | Dynamic Dictionary
  ACCESS | Exported
  CARDINALITY | Unlimited
  SUPERCLASS | Dictionary
  MEMBERS | construct/destruct
    ...

  NAME | Static Dictionary
  ACCESS | Exported
  CARDINALITY | Unlimited
  SUPERCLASS | Dictionary
  MEMBERS | construct/destruct
    ...

Concrete Class Descriptions

- Building block classes (C++ notation for class interface description)

```cpp
class Word {
public:
    Word (void);
    Word (const string &);
    int insert (int index, char c);
    int clone (Word &);
    int concat (const Word &);
    int compare (const Word &);
    // ...
};

class Document {
public:
    Document (void);
    Document (const Document &);
    virtual int open (const string &filename);
    int sort (int options);
    // ...
};

class Document_Iterator {
public:
    Document_Iterator (const Document &);
    int next_item (Word &);
    // ...
};
```

Design Alternatives

General Class Descriptions

- Building block classes (abstract notation for class interface description)

  NAME | Word
  ACCESS | Exported
  CARDINALITY | Unlimited
  MEMBERS | construct/destruct
    insert/remove characters
    clone
    concatenate
    compare
    ...

  NAME | Document
  ACCESS | Exported
  CARDINALITY | Unlimited
  MEMBERS | construct/destruct
    next word iterator
    sort
    ...

  NAME | Spell_Checker
  ACCESS | Exported
  CARDINALITY | Unlimited
  MEMBERS | construct/destruct
    spell_check
    ...

Design Alternatives
Concrete Class Descriptions (cont'd)

- Building block classes (C++ notation for class interface description)

```cpp
#include "Document.h"
#include "Static_Dictionary.h"
#include "Dynamic_Dictionary.h"

using namespace Dictionary;

typedef Static_Dictionary<Word, Word> Main_Dictionary;

class Spell_Checker {
   public:
      Spell_Checker (void);
      int open (const string &doc_name,
                const string &main_dict_name,
                const string &private_dict_name);
   int spell_check (ostream &standard_output);
   private:
      Document named_document;
      Main_Dictionary main_dictionary;
      Private_Dictionary private_dictionary;
};
```

Spell checker Implementation

- Main class

```cpp
Spell_Checker::Spell_Checker (const string &doc_name,
   const string &main_dict_name,
   const string &private_dict_name) {
   if (named_document.open (doc_name) == -1
       || main_dictionary.open (main_dict_name) == -1
       || private_dictionary.open (private_dict_name) == -1) {
      cerr << "initialization problem";
      throw Invalid_Name ();
   }
}
```

- Main class (cont'd)

```cpp
int Spell_Checker::spell_check (ostream &standard_output) {
   int result = 0;
   Word word;
   named_document.sort (REMOVE_DUPS);
   for (Document_Iterator doc_iter (named_document);
        doc_iter.next_item (word) != -1; )
   { if (main_dictionary.find (word) != -1
            || private_dictionary.find (word) != -1)
      continue; // found word
   else {
      if (main_dictionary.find (word) != -1)
         // erroneous word
         result = -1;
   }
   // ...
```
Design Alternatives

Advantages of Object-Oriented Design

- Increased modularity:
  1. Easier to understand the components in isolation, since data coupling and visibility have been reduced
     - *e.g.,* modules and classes are composed of related activities
  2. More adaptive to specification and implementation changes, since changes are localized
     - *e.g.,* most changes occur in representations, rather than interfaces
     - By hiding objects’ representational details, changes will not ripple through design (unless class specification changes)

Disadvantages of Object-Oriented Design

- Certain problem domains do not necessarily benefit from an object-oriented approach
  - *e.g.,* mathematical routines for numerical analysis, where there is no need for shared state...
- Requires more work in the upstream activities
  1. *e.g.,* analysis, modeling, and architectural design to determine architectural components, relations, and interfaces
  2. Often not as intuitive to determine the objects (without training and practice)
- Requires an object-oriented language for best results

---

Spell Checker Driver

The main program is:

```c
int main (int argc, char *argv[]) {
    if (argc != 4) {
        cerr << "usage: " << argv[0] << " doc-name, main-dict, private-dict"
        << endl;
        return 1;
    }
    Spell_Checker batch_checker (argv[1], argv[2], argv[3]);
    if (batch_checker.spell_check (cout) == -1)
        return -1;
    else
        return 0;
}
```

Note how the object-oriented decomposition uses essentially the same algorithm as the original spell-checker...
- However, the architecture is totally different

Advantages of Object-Oriented Design

- Increased modularity:
  1. Easier to understand the components in isolation, since data coupling and visibility have been reduced
     - *e.g.,* modules and classes are composed of related activities
  2. More adaptive to specification and implementation changes, since changes are localized
     - *e.g.,* most changes occur in representations, rather than interfaces
     - By hiding objects’ representational details, changes will not ripple through design (unless class specification changes)

Disadvantages of Object-Oriented Design

- Certain problem domains do not necessarily benefit from an object-oriented approach
  - *e.g.,* mathematical routines for numerical analysis, where there is no need for shared state...
- Requires more work in the upstream activities
  1. *e.g.,* analysis, modeling, and architectural design to determine architectural components, relations, and interfaces
  2. Often not as intuitive to determine the objects (without training and practice)
- Requires an object-oriented language for best results

---

Class data and member functions are equally emphasized
- However, higher-level structuring of activities is postponed
- Object behavior is independent of temporal ordering
  - *i.e.,* the *shopping list* approach
  - Easier to reuse and extend classes in other systems, since emphasis is on stable interfaces
    - *e.g.,* reuse sort from system sort application
Potential Modifications

- Make the program run interactively, rather than in “batch” mode
  - e.g., integrate with a text editor and make the program work on user-selected regions of the document (e.g., GNU emacs):
    1. Query the user to check if an unrecognized word is misspelled
    2. If it is misspelled then
    3. Replace the word in the document
    4. Potentially add the word to the private dictionary, if user specifies this action
    5. Produce an updated private dictionary
- Remove arbitrary limits on input document size
  - i.e., does not need to fit into memory

Make the program handle multiple input files
- Make the program handle multiple dictionaries
- Modify the program to perform other text oriented tasks, e.g.,
  - Build a document index or cross-referencer
  - Build an interactive thesaurus
- Make the program work on other types of files, e.g.,
  - LaTeX or TeX files
  - nroff files
  - MS Word files
  - postscript or dvi files

Parting Thought

- Sometimes the “best” design is the least elaborate one:

  ```
  % cat tex-spell-check
  dextex $1 | \ # strip tex formatting commands
  tr A-Z a-z | \ # map upper case to lower case
  tr -cs a-z \ '012' | \ # remove all non-words
  sort -u >! /tmp/words # remove duplicates
  comm -2 /tmp/words /usr/dict/words
  ```
- Advantage:
  - Easy to get right (once you understand UNIX tools ;-)), since it is very decoupled...
- Disadvantages
  - Doesn’t work very well for prefixes/suffixes
  - Slow... (many processes, many stages)

Concluding Remarks

- Object-oriented design differs from algorithmic design in several respects:
  - Structure of the system is organized around classes/objects rather than functions
  - Objects are typically more “complete” abstractions than are functions (e.g., they include data emphasis as well as control flow emphasis)
  - Algorithmically decomposed components have verb names, while object-oriented components have noun names
- Advantages of object-oriented design are most evident in
  1. Large-scale systems
  2. Evolving systems