Building Object-Oriented Software for Experimental Work

台北科技大學
資訊工程系
鄭有進 副教授
http://www.ntut.edu.tw/~yccheng
Outline

- Experimental work and problem solving
- Characterizing software development for experimental work
  - requirements, expectations, and realities
- A software-centric view of experimental work
  - The best practices
- A case study
  - RANSAC Curve detection
- Conclusion
1. Experimental work and problem solving
Experimental work, problem solving, and software development

- Experimental work involves advanced problem solving
- In many research areas, problem solving is carried out through computer programs,
  - E.g., signal processing, communication, computer vision, multimedia and etc.,
  - A comprehensive system can be complex
Phases for problem solving
[Skowronski 2004]

- **Preparation**
  - Gather information about problem
  - Carry out experiments and test possible solutions

- **Incubation**
  - Let the problem sit and work subconsciously

- **Illumination**
  - The moment when the solution’s central idea appears in a flash of light

- **Verification**
  - Flash of insight expands into a solution, which is then tested against reality
Mapping problem solving phases to software development (I)

- Preparation ⇔ research; building software
  - A software/framework built based on known literature.
  - The software/framework must be robust/efficient enough for implementing well-defined tests of ideas.
  - Should allow you to ask and answer “what if”
  - The purpose is to generate observations about the known methods (esp. those with rooms for improvement)
Mapping problem solving phases to software development (II)

- Incubation ⇄ pre-production use of software
  - Not much parallel here, but the software should readily help exploration
Mapping problem solving phases to software development (III)

- Illumination $\leftrightarrow$ implementation and test of the central idea
  - The "eureka!" idea should be implemented and tested for their correctness.
Mapping problem solving phases to software development (IV)

- Verification ↔ software for the proposed; Comparison with existing methods
  - Well-defined targets of comparison to claim improvement
  - Fairness in input and comparison category
## Software work in research phases

<table>
<thead>
<tr>
<th>phase of Research</th>
<th>Amount of software work</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>preparation</em></td>
<td>★★★</td>
</tr>
<tr>
<td><em>incubation</em></td>
<td>★★</td>
</tr>
<tr>
<td><em>illumination</em></td>
<td>★★</td>
</tr>
<tr>
<td><em>verification</em></td>
<td>★★★</td>
</tr>
</tbody>
</table>
2. Characterizing software development for experimental work
Expectations/Requirements

In addition to being functionally correct, software for experimental work should:

- **Productivity:** Deliver results good enough for publishing, usually before its completion.
- **Extensibility:** Support exploration of new ideas, it must be easy to add (and remove) functionalities.
- **Be explorative and observable:** Support online analysis and drill-downs.
- **Maintainability and reusability:** Accumulate well without constantly rebuilding.
Realities (I)

- Research maybe well-planned, but software development for research work is usually under-planned.
- Software work is usually delivered in a water-fall fashion.
  - Problems are discovered too late and remedies can be costly.
- Frequent re-implementation.
Realities (II)

- Development period is extended
- Development team is unstable
  - People moving in and out (esp. during summer).
  - New staff with unpredictable skill level
- Programs are brittle
- Programs are insufficiently (*barely*) tested
3. A software-centric view of experimental work
A software-centric view of experimental work

- Process
- Domain
- Skills
- People
- Tools

Software

Diagram showing the interconnections between process, domain, skills, people, and tools, centered around software.
A number of best practices

1. Sizing the gap
2. Object-orientation
3. Adopt a process
4. Design patterns
1. Sizing the gap: bridging features to resources [OOMD 2005]

- What are the desired features?
- What are the available resources?
  - Libraries, framework, environments
- Can the desired features be constructed directly with the available resources?
  - If so, just do it
  - If not, add a layer of abstraction
  - recur
How large is the gap?

- Narrowly viewed, the gap is small
  - Recall that Fortran and C used to be (and still is) good enough for building programs for experimental work

- But the gap will be larger if you have more expectations:
  - The “ilities”: Observability, controllability, extensibility, maintainability, reusability and productivity
  - This is especially so if we treat experimental work as a comprehensive software rather than as a program
Gap bridging by building around an existing tool (I)

- Center your system around a software tool (e.g., MatLab, Khoros, etc.)
  - Pros:
    - Good support at API, class library, framework and component levels
    - Powerful UI metaphors for modeling your experiments
    - Productivity increases once mastered
Gap bridging by building around an existing tool (II)

- **Cons:**
  - Architectural impact: tool put constraints on how a function is implemented
  - Customization can be difficult
  - Tool use (UI and API) involves a learning curve and can be non-trivial
  - Performance problems.
2. Object-orientation

Procedure-orientation:

- We view work to be done mainly as a procedure
- Implicit in the process are entities upon which the procedure is carries out.
  - Entities captured in data structures
  - Procedures specified by algorithms
Adage:

Algorithms + Data Structures = Programs

Niklaus Wirth
Object-orientation

- We still view the work to be done as a procedure.
- But participants (entities) in the procedure are made explicit.
- The participants interact to get the procedure done.
- Object-orientation captures the static structure of the application domain
  - More resilient to change
  - Suitable for evolutionary work
Object-orientation

- Encapsulation, messages and methods, inheritance, polymorphism.
- Reification: promote to object where you expect variation
  - Example: design patterns strategy and state
Form and Function (1)

- Software has two aspects: form and function
In mainstream OO processes (e.g., RUP and XP), function and form grow together.
- Architecture baseline takes shape early as primary use cases are built.

In the typical experimental work software development, function gets most of the attention.
- Since the perceived gap is usually small
- Form is non-existing, or is just whatever the outcome of the aggregation of functionalities.
The adage revised for object-orientation:

**Objects + Responsibilities + Collaborations = programs**

[Algorithms + Data Structures = responsibilities through Operations]

We still do it, but at a lower-level and delayed in the process]
3. Adopt a software development process

- Regardless of the available resources, if the gap is large, your task will be non-trivial.
  - you need a *software development process* to guide your work and to address requirements, expectations and realities simultaneously
  - To help new staff in their transition phase in joining the development team
  - To systematically accumulate the artifacts
    - But be light on the artifacts
Processes by weights

Agile methods:
- XP
- Crystal
- JAD
- ...

Heavy weight
- RUP
- OMT
- Fusion
- ...

Light weight
- Few artifacts
- Hot communication
- Small team

Heavy weight
- Many artifacts
- Cold communication
- Large team
Proven process elements (best practices in processes)

- Use cases (requirement capture)
- Increments (staging strategy)
- Iterations (rework scheduling strategy)
- Patterns (design and implementation)
- Unit tests (implementation)
- Continual integration
- Review (all stages)
A simple process and its workflow

1. Write use cases
2. For each use case
   3. Extract domain model
   4. Evolve into design model
   5. Implement and do unit tests
   6. Integrate
   7. Test use case
   8. Review
## Workflow elements, roles and responsibilities

<table>
<thead>
<tr>
<th>Task</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing use cases</td>
<td>Researcher</td>
</tr>
<tr>
<td>Domain modeling</td>
<td>Researcher/staff</td>
</tr>
<tr>
<td>Design modeling</td>
<td>Staff</td>
</tr>
<tr>
<td>Implementation and unit tests</td>
<td>Staff</td>
</tr>
<tr>
<td>Integration and tests</td>
<td>Staff</td>
</tr>
<tr>
<td>Use case tests</td>
<td>Researcher/staff</td>
</tr>
<tr>
<td>Review</td>
<td>Researcher/staff</td>
</tr>
</tbody>
</table>
3.1 Use case modeling

- Use case: a sequence of steps describing an user-system interaction in which a well-defined service is tendered.
- Use case captures functional requirement for a user
- Use cases are high-level, domain-oriented descriptions written in a way understood by project stakeholders
Use case modeling (I)

- As a starting point in system modeling.
  - All the other artifacts are derived from the use cases.

- As a guide to testing the system built.
  - The system built must reflect the process presented in a use case.
  - Useful in function/acceptance tests
Use case modeling (1)

- Write just enough detail to keep the use case interesting
  - Too much detail reverts the development to a structured process
- You don’t need to have all use cases in place to start building system.
  - However, the first use case should not be a trivial use case
3.2 Increments and iterations (1)

- A use case is roughly an increment: it is not the whole thing, but it delivers functionality by coherent chunks.

- Several iterations may be executed on an increment
  - To remove bugs, add functionality, and improve quality
Increments and iterations (II)

- Aim for a working system (at least covering the present use case) at the end of an increment.
  - Early feedback to users can guide late-stage development
- Iterations should be time-boxed: e.g., 2-4 weeks on small projects.
Increments and iterations (III)

- Incremental strategy ensures that transition from domain model to design model is feasible
  - Increment delivers functionality in coherent chunks
  - First increment should shape the architecture baseline
  - Later increments built on top on early increments
Iteration vs. waterfall (I)
Iteration vs. waterfall (II)
3.3 Domain Modeling (1)

- Make domain model explicit
- Extract the static conceptual domain structure.
  - concentrate on the concepts
  - find out how concepts are related
- Capturing the static structure is one of the reason why OO models are more reusable
  - Static structure is less likely to change radically, as compared to functionalities.
Domain Modeling (II)

- Use case-based method for domain modeling:
  1. Scan for nouns in the use cases for concepts
  2. Scan for stative verbs in the use case for associations
- On the other hand, use whatever you feel most comfortable (CRC, semantic nets, conceptual diagram, header files, etc.) to get domain modeling going
Domain Modeling (III)

- The concepts should have a consistent medium-to-coarse granularity level
  - Fine grained concepts bug you down (too much detail)
- Avoid application classes, or classes due to realization platform
  - E.g., GUI classes and database
3.4 Design modeling (I)

- Models the domain as a set of software classes.
  - A design model is also a valid domain model, but it is both domain- and application-dependent, which may be not easy to obtain off-hand.

- Coarse-grained concepts are usually realized be a set of cooperating classes
  - Each coarse-grained concept becomes a micro-controller delegating work to its internal classes
  - Knowledge of design patterns and architecture styles are extremely helpful
Design modeling (II)

- Created by finding what a class must do to realize the operations being requested on the conceptual entities.
  - Responsibilities that are implied in the conceptual model must be made explicit in the design model.
  - Discover the responsibilities in the order of their occurrence.
- Responsibility assignment is the most important activity in OO design.
3.5 Implementation, unit tests and Continuous integration

- Write tests for every operation of an object (more detail on unit tests later)
  - Logical unit tests (LUT)
  - Integration unit tests (IUT)
  - Functional unit tests (FUT)
- Tests should be accumulative and automatic
  - Use a unit test framework such as CppUnit or JUnit
Testing can be an embarrassment-saver

“Requirements for quality [of research and experimental software] may be quite low, and the process may be a little better than debugging and a few regression tests. Nonetheless, the risks of embarrassments from failed public demos and withdrawn papers suggest a greater investment.”

Stuart Feldman
Queue, Feb 2005
3.6 Review

- Peer reviews of requirements, analysis, design, and code
- Process observation and improvement
  - Development speed
- Learning “what you don’t know you don’t know”
  - Holes in your layer to bridge the gap
- Artifacts accumulation and synchronization
  - Use cases, domain model, design model, key interaction, tests
Why design at all

- Design aims to provide a solution for the functional aspect
- Equally important, if not more, it provides a solution for non-functional aspects.
  - The “-ilities”
4. Design Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander

*The Timeless Way*
Four most essential elements of a pattern [GoF]

- **Pattern name**
  - a handle used to describe a design problem, its solutions, and consequences in a word or two.

- **Problem** describes situations to apply the pattern

- **Solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.

- **Consequences** are the results and trade-offs of applying the pattern
  - critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern
Example use of patterns

- Reification of behaviors for extensibility and variation

- examples:
  - Strategy [algorithm as object]
  - State [state as object]
  - Iterator [pointer as object]
Strategy pattern

- before

M x N
Strategy pattern

- After

![Diagram of strategy pattern with classes App, Finder, Sorter, ABetterFinder, ABetterSorter, sorter->sort(), _finder->find() connected]
## Pattern Space [GoF]

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Factory Method (107)</td>
<td>Creational</td>
</tr>
<tr>
<td></td>
<td>Adapter (139)</td>
<td>Structural</td>
</tr>
<tr>
<td></td>
<td>Interpreter (243)</td>
<td>Behavioral</td>
</tr>
<tr>
<td></td>
<td>Template Method (325)</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Abstract Factory (87)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Builder (97)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prototype (117)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Singleton (127)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adapter (139)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bridge (151)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Composite (163)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decorator (175)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Facade (185)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proxy (207)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chain of Responsibility (223)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Command (233)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iterator (257)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mediator (273)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memento (283)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flyweight (195)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observer (293)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State (305)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strategy (315)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Visitor (331)</td>
<td></td>
</tr>
</tbody>
</table>
Pattern purposes

- Creational patterns
  - the process of object creation

- Structural patterns
  - the composition of classes or objects

- Behavioral patterns
  - the ways in which classes or objects interact and distribute responsibility
Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time.

Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.
A Learning Strategy: Contrast

- Pick an example from Motivation or samples code
- Design it from the first principles, or simply just you-know-how
- Read the pattern
- Redo the example according to the pattern
- Compare the two implementations
### Process Context (1)

<table>
<thead>
<tr>
<th>Expectation/ requirement /reality</th>
<th>strategies</th>
</tr>
</thead>
</table>
| Produce result before completing the entire program | 1. Choose the use case covering core functionality as the first increment.  
2. Do at least two iterations on the first increment. First iteration build the core functions; second iteration and on improves architecture and usability. |
### Process Context (II)

<table>
<thead>
<tr>
<th>Expectation/requirement /reality</th>
<th>strategies</th>
</tr>
</thead>
</table>
| “Correctness”                    | 1. Practice “tester’s mentality”: play adversary of the classes/methods by actively thinking how to break them. Do the same to the use case.  
2. Write tests that run automatically using a testing framework such as CppUnit. |
### Process Context (III)

<table>
<thead>
<tr>
<th>Expectation/ requirement /reality</th>
<th>strategies</th>
</tr>
</thead>
</table>
| To support exploration of new ideas; easy to add (and remove) functionalities. | 1. Choose the use case covering core functionality as the first increment.  
2. Architecture styles and design patterns  
3. Add iterations (within this increment or later) to expose the architecture.  
4. To not to deviate from the architecture in later increments |
<table>
<thead>
<tr>
<th>Expectation/ requirement/reality</th>
<th>strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication/staff stability</td>
<td>1. Create self-documenting code with tests</td>
</tr>
<tr>
<td></td>
<td>2. Choose a minimal set of artifacts</td>
</tr>
<tr>
<td></td>
<td>3. Mandate that artifacts are in place at the end of each increment</td>
</tr>
<tr>
<td></td>
<td>4. Sync artifacts</td>
</tr>
</tbody>
</table>
4. Case studies
Case study: curve detection

- RANSAC curve detection: the input is a binary edge map from which curves (circles and ellipses) are extracted.
- Detection performance (missing rate/false positive rate/time/space) must be compared with existing RANSAC and randomized Hough methods.
- Improvement to sampling/transform might be found from analyzing false positives.
- Pre-run a long experiment.
Use cases

- Run circle detection on a binary edge map
  - Core steps and variants
- Compare two circle detection methods
  - Identical resource constraint
  - Best effort comparison
- Play back the detection process
- Analyze false positives
Use case for first increment

*Use case: Run circle detection on a binary edge map*

1. This use case begins when user request circle detection done on a binary edge map.
2. User prepares a binary edge map.
3. User prepares detection profile.
4. Circle detector reads the profile to set up the detection, including sampling strategy, transform, and
5. Detector carries out the detection. Grader is notified when detection is done..
6. Grader reports detection results base on reference keys in detection profile.
7. Grader reports performance characteristics, including missing rate, false positive rate, and computing time.
Review use case and domain model

- Once use case and domain model are ready (first draft), call for a review
- Purpose of the review: validation and learning
- In the review, staff learn how conceptual model might interact to satisfy the use case
- Another method for discovering concept CRC (Class/Responsibility/Collaboration)
Design model: sampler

- Responsibility: take random samples from the edge map
- Variation: pre-segmented edge map
Sampler as a filter controller
Increment 1 design model

Figure 1
Iterations

- Two additional iterations are added
  - To expose the filter architecture
  - To expose the MVC architecture for Sampler-Transform-Testing Criteria

- Improve the usability
  - Multiple tries on randomly generated images
  - Further refine detection profile
  - Summary of detection results
Increment 2 and on

- Build on existing MVC+filter architecture
- Compare two methods
  - Two methods identical except in the intended component (e.g., sampling method or transformation method)
- Template method (GoF)
  - Flow modeled with a template method
  - Variations hooked to the template method at the extension points
Concentrate on the Domain

- Add GUI only if domain layer is relatively complete
- Avoid implementing service layer (e.g., distribution, persistence framework)
  - They are technically challenging
  - Could easily distract your effort (domain)
  - Find or buy service layer class libraries/framework
Language and tools

- Language: gnu C++
- “Ide”: Emacs, Eclipse
- Version control: cvs + winCvs, Eclipse
- Unit testing: CppUnit
- UML tool: dia, visio
- OS: Linux RedHat 6.2, 7.0, 9.0
5. Conclusion
Summary

The process improves on the following aspects of the project:

- Productivity: continual deliveries
- Quality: through unit and acceptance testing
- Communication: in review sessions
- Documentation: self documenting code, tests, and additional artifacts
- Staff transition
Further reading

- Testing
  - E. Gamma and K. Beck, “J Unit cookbook”.

- Reactoring

- Patterns

Development process


Blaha and Rumbaugh, Object-oriented Modeling and Design, 2nd ed, Prentice Hall, 2005

C++/STL