Building 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

- Requirements, expectations and realities
- On a development process
- Core programming skills
- Case study
- Conclusion
A software-centric view

domain

process

skills

people

software

tools
Requirements

- Research areas like signal processing, communication, computer vision, multimedia and etc., concepts are sometimes proven through computer programs.
- Programs are abstractions of certain aspects of the complete system
- Comprehensive system can be complex
Expectations

Software for experimental work should

- Deliver results good enough for publishing, usually before its completion.
- Support exploration of new ideas, it must be easy to add (and remove) functionalities.
- Support online analysis
- Accumulate well without constantly rebuilding
- be ready for commercialization with GUI and service layer enhancements once core functionality is working.
Realities

- Research maybe well-planned, but development is usually under-planned
- Delivered in a water-fall fashion
  - Problems are discovered too late and remedies can be costly
- Development team is unstable
- New staff with unpredictable skill level
- Programs are under-tested
Abstraction levels and strategies

- A theorem can be proved to solve the problem; prove.
- Abstractions can be made to reveal a high level model; do simulation on conjecture.
- System can be built; have the system actually solve instances of the problem.
Initial Situations

- You’re starting a new research project from scratch
  - Adopt a process
  - Build application around an existing system
  - Home-grow

- You’re starting a research project based on a existing legacy system
  - Retrofit the legacy system into a process
  - Etc.
Build application around an existing tool

- Center your system around a software tool (e.g., MatLab, Khoros, etc.)
  - **Pros:**
    - Good support at API, class library, and component levels
    - Powerful UI metaphors for modeling your experiment
    - Productivity increases once mastered
  - **Cons:**
    - Architectural impact: tool puts constraints on how a function is implemented; customization can be difficult
    - (Usually) does not directly support the idea
    - Tool use (UI and API) can be non-trivial
If you choose to build your own, you need a *software development process* that addresses requirements, expectations and realities simultaneously.

There are well-documented processes in the SE area.

In this talk, a process based on object orientation is used.
Procedure-orientation

- We view work to be done mainly as a procedure
- Implicit in the process are entities upon which the process is carries out.
- But procedure does not capture the entities well
Object-orientation

- We still view the work to be done as a procedure.
- But participants (entities) of the process are made explicit.
- The participants interact to get the procedure done.
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.
- Form is usually just whatever the outcome of the aggregation of functionalities.
- A solution: retrofit form to functions.
Process weights

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

RUP
- OMT
- Fusion
- ...

Light weight
- Few artifacts
- Hot communication
- Small team

Heavy weight
- Many artifacts
- Cold communication
- Large team
Proven process elements

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

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

<table>
<thead>
<tr>
<th>Activity</th>
<th>Responsible</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>
1. Use case modeling

- Use case: a sequence of steps describing an user-system interaction in which a well-defined service is tendered.

- Scenario: a set of use cases tied together in order to complete a user’s task in the application domain.
Use case functions (I)

- 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 functions (II)

- 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 tips

- 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
Increments and iterations (I)

- 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 improve late 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 systems in coherent chunks
  - First increment should shape the baseline architecture
  - Later increments built on top on early increments
Iteration vs. waterfall (1)

Effort/result

requirement

Analysis

Design

Implementation

Testing

Integration

time
Iteration vs. waterfall (II)

Effort/result

Iteration 1  |  Iteration 2  |  ...  |  Iteration n

Iteration 1  |  Iteration 2  |  ...  |  Iteration n
2. Domain Modeling

- Extract the static conceptual domain structure.
- Use whatever you feel most comfortable (CRC, semantic nets, conceptual diagram, header files, etc.)
  1. concentrate on the concepts
  2. find out how concepts are related
- The concepts should have a consistent medium-to-coarse granularity level
  - Fine grained concepts bug you down (too much detail)
3. Design modeling (1)

- Models the domain as a set of software classes.
  - Therefore, a design model is also a valid domain model, but it 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 function being required 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
4. Implementation and unit tests

- Can happens after the creation of design model
  - design models are drawn and recorded
- Can happen in parallel with design model creation
- Write tests for every member function of a class (more detail on unit tests later)
- Tests should be accumulative and automatic
5/6. Continual integration and functional tests

- Function tests are written for each use case.
- An increment is completed only if the functional tests pass.
- Functional tests can be written by extending unit tests.
- Function tests should cover: (e.g. circle detection)
  - Single circle without noise
  - Multiple circles with noise
  - Concentric circles
  - Mutually occluding circles
7. Review

- User evaluation and feedback
- Process observation and improvement
- Learning “what you don’t know you don’t know”
- Artifacts accumulation and synchronization
  - Use cases, domain model, design model, key interaction, tests
# Process Context (I)

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

- Language: C++
- IDE: Emacs
- Unit testing: CppUnit
- Version control: CVS
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
Additional issues

- System architecture
  - Graphical user interface
  - Domain layer
  - Service layer
  - Database/File system
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++
- “**I de”**: emacs
- **Version control**: cvs + winCvs
- **Unit testing**: CppUnit
- **UML tool**: dia
- **OS**: RedHat 6.2 Linux
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
Things not covered

- Retrofit an existing program to the development process
  - Needs refactoring
  - Patterns can useful as a guide to refactoring
- Team size >5
  - Tried the process on a 3-person team
Further reading

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

- Reactoring

- Patterns
  - E. Gamma et al., Design Patterns, Addison-Wesley. 1994.

- **Development process**

- **Process management**
C++/STL