Implementation

Dr. Fritz Solms

September 14, 2015
Responsibilities of implementation

**Inputs:**
- requirements specification
- target architecture specification
- detailed design (typically for use case)

**Outputs:**
- unit test for use case
- code for use case
- developer documentation usually as in-code comments
  - potentially some user documentation
- Build & deployment scripts usually for whole system
Quality requirements for implementation

- Completeness & correctness
- Simplicity and understandability
- Flexibility/maintainability
- Bi-directional traceability between code and design
- Performance
Simplicity

- Hoare:
  \textit{Inside every large problem is a small problem struggling to get out.}

- John Graham-Cumming’s brilliant variation:
  \textit{Inside every large, complex program is a small, elegant program that does the same thing, correctly.}

- code for simplicity/understandability
  - performance optimize later

- Follow DRY (Don’t Repeat Yourself) principle.
SOLID Principles

- Single responsibility principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle
Single responsibility principle

- Each unit has only a single responsibility
  - at some level of granularity
  - e.g. control car, control brakes, monitor brake fluid presure

- Cohesion
  - state space should not be divisible

- Mechanism:
  - Good design across levels of granularity.
  - Estensibility mechanisms except inheritance
  - chain of responsibilities, interception, decorator, . . .

- Benefits:
  - understandability
  - localized responsibility
  - increased reuse
  - simpler maintainability
Open-Closed Principle

- Should be able to extend without modifying.
  - Extensions should not affect existing code.
- Mechanisms:
  - Interception points
  - Subclassing
  - Decorator pattern
  - events and observers
  - chain of responsibilities
  - pluggability (e.g. of further spam rules)
Liskov Substitution Principle

- Derived classes must be substitutable for base classes.
- Mechanism:
  - Post-conditions should only be increased.
  - Pre-condition should only be reduced.
  - Quality attributes may only be strengthened.
Different user roles see only aspects of system elements which they should see:
- a class using another class also plays a user role

**Mechanism:**
- Define fine-grained, client-specific interfaces

**Example:**
- Separate interfaces for onto brake system
  - one for brake control system
  - one for brake monitoring system (e.g. driver console)
Dependency Inversion Principle

- Decouple high-level from low-level object (clients from service providers)
  - interfaces, adapters, messaging, service provider lookup, dependency injection
- Decouple low-level from high-level objects
  - use observer
  - use inheritance sparingly
- Abstractions should not depend on details.
- Don’t hard-code environmental parameters
Best practices around entities

- Entities should have no “business processes”.
- Do not return entities as value/result objects.
- Leave marshalling/de-marshalling to frameworks
  - O/R mapping
  - XML mapping
- Consider encoding global entity constraints
  - e.g. bean validation constraints
- Keep in mind the bigger picture
  - not hard-coding data structure
  - RDF/ontologies.
  - globally unique object identifiers.
Cross-cutting concerns

- What are cross cutting concerns?
  - Responsibilities which should be applied across services.
  - e.g. logging, authorization, . . .

- Don’t hard code cross cutting concerns into actual functionality.
  - Can use aspect-oriented frameworks to weave in functionality.
  - Other frameworks like Java-EE have limited functionality of applying interceptors across services.
  - Can leave some cross cutting concerns to service or message bus.
Errors and Exceptions

- Differentiate between exceptions and errors
  - Map each pre-condition onto separate exception
  - Publish exceptions (e.g., documentation)

- Do not hide errors/exceptions
  - At least log them

- Do not map onto codes
  - Prevents polymorphic handling of different exception types.
Do not hard-code configuration information

- Example Configuration information
  - server URLs.
  - authentication credentials.
  - system parameters.
  - ...
- Use
  - Configuration files
  - Singleton objects
  - Management Interfaces
Write self-testing code

- Use assertions liberally.
  - evaluation can be expensive, but can be switched off.
  - tests assumptions made in code.
  - implementation testing versus contract (unit) testing.
- Can also in implementation test whether contract is realized.
  - can test each post-condition before returning.
Best practices on code documentation

- Document interfaces liberally for users.
  - Document services contract.
  - Use documentation framework for your technologies (e.g. JavaDoc)
- Document classes sparingly
  - Mainly self-documenting code.
  - Avoid documenting the obvious.
  - Use in-code comments.
  - Not for class users, but for class maintainers.
  - Refer to papers/books when implementing standard algorithms.
Coding Standards

- Follow coding and naming standards for your technology
  - e.g. Camel, indentation, ..
- Avoid known bad practices.
  - e.g. GOTO statements, non-final non-virtual methods, non-virtual destructors, ...
- Avoid using deprecated features.
Debugging

- Study logs
- Analyze memory dumps
- Analyze thread dumps
- Switch on assertions
- **Unit tests** to test unit with mock objects
- **Integration tests** to test unit with real service providers
- Study commit traces to see what code could have introduced bug
  - potentially roll back.
- Try **reproduce bug** in controlled environment
- Incrementally **isolate bug**.
- **Debuggers** are over-rated, but may assist for certain applications.
Optimization

- Avoid when possible
  - leave largely to compiler and run-time environments.
- If you do it:
  - leave as last step
  - write for simplicity and understandability
  - use profiler
- Can optimize for different qualities
  - Performance
  - Code size
  - Memory footprint
Refactoring

- Code quality often detiorates under maintenance.
  - Unneccesary complexity
  - Duplicate code
  - Excessively large methods or classes
  - Switch statements
  - Unecessary strong coupling
  - deterioation of single responsibility principle

- Change code without changing what it does for user (same contract)
  - Factor out lower level responsibilities.
  - Move responsibilties to where they should be.
  - Insert decoupling via interfaces
  - Remove plumbing code and leave to architecture/frameworks
  - Move cross-cutting concerns into aspects/interceptors
  - Improve naming
  - Hide implementation details

- Continuous refactoring
  - Requires testing frameworks.