Software Engineering with Fusion and UML

Prof.Dr. Bruce W. Watson
bruce@bruce-watson.com

Fusion: Requirements

This is arguably the most important phase: it will drive the rest of the process.

General description

- Provide a general description in the form of a paragraph or two regarding the system.

- Identify the *stakeholders*: those groups/people who have a vested interest in the success (or failure!) of the system. Assign some kind of mnemonic to the stakeholders.

- Write a *value proposition*: what are the critical aspects which will define the system's success.

[CF1] Define high level requirements

- Functional and nonfunctional requirements.
- Brainstorm, with client.
- Name each requirement.
- Each should provide an *external* view of the system.
- Identify functional areas:
  - Classes of users.
  - Product feature.
  - Mode of operation.

  - Lifecycle phase.

- Nonfunctional requirements:
  - Constraints: t-to-m, platforms, interoperatibility.
  - Qualities: q-of-s (performance, MTBF), usability.

- Output:
  - Natural language descriptions.
  - Cross references back to key people (auditability).

Example
(we'll probably us another in class.) FIRE Station:

1. Description: A graphical environment for constructing, manipulating and testing finite state machines, including running them on some input text.

2. Stakeholders:

   (a) Me (to make money and have a successful product).

   (b) Finite state machine (FSM) developers (such as linguists, compiler writers and hardware designers, etc.).

   (c) FSM end-users (spell-checker users and compiler users, etc.).

3. Value proposition: The environment will:

   (a) Offer more choices of FSM types.

   (b) Support domain-specific FSMs.

   (c) Provide massive scalability up to FSMs with millions of states.

   (d) Provide an intuitive user interface, along with the traditional symbology for FSMs.

   (e) Provide the best graph drawing and lay-out of FSMs.

4. Requirements:

   (a) Functional:

      i. Build a regular expression (RE).

      ii. Build an FSM inductively.

iii. Convert between FSMs from an REs.

iv. Test an FSM on some input string.

v. Save the workspace for later restart.

vi. Load and save FSMs and REs in standard format.

(b) Nonfunctional:

i. Deal with large-scale FSMs.

ii. Scale linearly.

iii. Multi-platform support (Java).

iv. Fail-safe during a crash (do not destroy a half-built FSM).

v. Acceptable performance on a P100/Win32 machine.

vi. Use FIRE Engine.

Coleman 6

[CF2] System functionality and scale

- Actors: external entity that uses/interacts with system.
- Use case: a set of interactions between the system/actors to achieve some specific goal.
- Finding new ones: consider both types of requirements, actors, use cases.
- Output:
  - Use cases.
  - Scenarios (CRC cards).
  - Scale: simultaneity (and priorities), actor population, geographical separation.

Coleman 7–11

[CF 3] Relating functional and nonfunctional requirements

This will be heavily used by EVO Fusion.

1. Output: a matrix relating the functional and nonfunctional requirements.

2. Each box (which, of course, corresponds to one functional and one nonfunctional requirement) in the matrix should contain three key indicators :

   (a) The level (at a minimum) of the nonfunctional requirement which is to be achieved for the given functional requirement.

   Some nonfunctional requirements may be things which cannot be expressed in levels, meaning that this would be written as "total" or something similar.

(b) The difficulty (risk) of achieving the two requirements at the same time; i.e. is the nonfunctional one achievable while implementing the functional one?

(c) The priority of the combination of the two. Keep track of whether the priority was determined by the client or by you.

3. You can already make some notes about boxes which have high risk and high priority — they could become problems.

   You will be scheduling the development process in the following order (will be done later in the evolutionary cycle):

   (a) High risk and high priority.

   (b) Low risk and high priorty.

   (c) High risk and low priority.

   (d) Low risk and low priority.

[CF4] Define use case specifications

- Strategies:
  - Generalize from scenarios.
  - Don't forget to use conditionals and iterations.
  - Use the requirements matrix.
  - Maintain consistent level of abstraction.
- Output: detailed use cases:
  - Goal.
  - Assumptions.
  - Actors involved.

– Sequence of steps.

– Information sources.

– Nonfunctional requirements.

– Variants.

[CF5] Structure use case specifications

- Restructuring step.
- Shared behaviour: introduce sub use cases.
- Variations: introduce extensions.
- Output: detailed use cases with structuring diagrams.

Coleman 12–20

6 Review and refine requirements models

- Review all requirements with clients.
- Track all requirements.
- Finishing?
  - Track time distribution.

Coleman 21–23

Fusion: Analysis

About *what* the system does, not how it does it.

[CF1] Domain class diagram

- Stick to high level abstractions.
- Involve domain expert.
- Strategies:
  1. Model the actors themselves as classes too.
  2. Use any pre-existing classes for the domain.
  3. Examine use cases for classes and associations.
  4. Introduce *generalizations* (is-a) and *specializations* between classes.

5. Introduce *aggregations* (has-a) to internally structure a class.

6. Other relations: *associations*, *navigability*.

7. Stay away from computing notions.

8. Cardinalities of associations: 0, 1, *, $i..j$.

- Output: *domain class diagram*, a type of UML class diagram — typically limited to classes (names only), generalization relationships, navigability, aggregation, dependencies and cardinalities.

Example of UML class diagram

[CF2] Analyze use cases: system operations and interface

1. Review each use case and make the use case steps more precise.

2. Determine responsibilities for each use case:

   (a) A piece of functionality.

   (b) Find them using CRC cards.

3. Find *system operations*: the set of interactions between the actors and the system, and between use cases and sub-use cases.

4. Tactics:

(a) Identify or distinguish similar responsibilities.

(b) Actions on the system: give parameters; record responsibilities.

(c) Make sequentiality/concurrency explicit.

5. Output: *system interface* (set of system operations and output events between system and actors — not easily depicted directly in UML).

6. Along with the use case scenarios, these will form part of the testing document.

[CF3] Analysis class diagram

1. Start with domain class diagram.

2. Drop all classes which fall outside the system boundary.

3. Examine use cases and system operations to find new classes.

4. Introduce new classes as required (without doing algorithmics) — applying knowledge of what would be computationally required (something the domain expert couldn't do).

5. Output: *analysis class diagram*, a type of UML class diagram.

6. This will become the basis for the architecture.

[CF4] System operations and event specifications

1. Proceed through the responsibilities and find:
   - Preconditions.
   - Postconditions.
   - Invariants.
   - Detailed sequence of actions/events.
2. Concurrency/atomicity are not an issue.
3. Output: text annotations to the use cases.
4. This will be part of the component-wise testing.

[CF5] Review analysis models
Check consistency:

- Use cases/analysis models.

- System operations/analysis classes.

Fusion: Architecture

Architecture:

- System is specified in terms of components and interactions.

- Two levels:
  - *Conceptual*: interaction specified informally at high level.
  - *Logical*: interaction in terms of messages.

- Can be applied recursively (flexible granularity).

[CF1] Review/select architectural style

- Largely guesswork at this point, but let nonfunctional aspects drive it: performance-based.

- Main styles:
  - Layered.
  - Pipe and filter.
  - Blackboard.
  - Microkernel.
  - Interpreter/virtual machine.

- May actually involve mixing them.

[CF2] Informal design of architecture

- Subdivide the analysis class diagram into components, according to the chosen architecture.

- Hints:
  - Focus on *cohesiveness*, *loose coupling*, etc.
  - Support of nonfunctional requirements.
  - Legacy components.
  - GUI components.
  - DB components.
  - Artificial components for grouping purposes.

  - A view to building your portfolio.

- Document:
  - Components.
  - Responsibilities.
  - Sketch collaboration diagrams: these are usually done at the object level, but can be done between components.

Examples of UML collaboration diagrams

[CF3] Develop conceptual architecture

- Focus on risky areas from step 2.
- Use the scenarios for each use case to validate the collaboration diagrams.
- Within the collaboration diagrams (one for each use case scenario), verify:
  - Sequencing.
  - Concurrency.
  - Parameters, returns.
  - Data-flow.
  - Creation.

- Use CRC role-playing to verify interfaces to components.

- Output: collaboration diagrams (sequence diagrams could be used if you want).

[CF4] Develop logical architecture

- Refine collaborations into messages (methods).
- Order them according to risk.
- Determine architectural mechanisms and *patterns*.
- Explore timing effects here.
- Output:
  - Collaboration diagrams of messages flowing between components.
  - For each component:

* Interface: pre- and post-conditions.
* Do an analysis class diagram for the internals.
* For each message: is it (a)synchronous?
* Interface opaqueness.

[CF5] Rationalize (justify) architecture

- Are there clear responsibilities for components?
- Are interactions distributed among components?
- Are quality requirements satisfied?
- Can the architecture be allocated to a physical architecture?

[CF6] Create design guidelines

Designer principles:

- Security.
- Mechanisms:
  - C/S.
  - TP.
  - Load balancing.
- Signatures.

Fusion: Design

Outputs:

- Design class diagram.
- Object collaboration diagrams.

[CF1] Initial class diagram

Copy the analysis class diagram as a starter.

[CF2] Object collaboration diagram

- The essentially algorithmic.
- *Intra*-component collaborations.
- Threading and concurrency should be explicit — introduce critical regions/mutual exclusions.
- Analysis classes will become one or more design classes.
- Within a component:
  - First object to receive the message is the *controller*.

  - Subsequent ones are *collaborators*.

- Use CRC roleplaying to verify.

- Output: object collaboration diagrams.

[CF3] Object aggregation and visibility

- Five classes of visibilities: *association*, *parameter*, *local*, *global*, *self*.
- Match lifetimes of objects.
- *Aggregate* things with similar lifetimes.

[CF4] Rationalize design class diagram
Consider objects, classes, behaviours:

- Similar operations? Unify.

- Similar behaviours for different classes? Factor a common parent (generalize) and specialize (derive).

- Seperable behaviours in a class?
  - Split class completely,
  - Create aggregate class, or
  - Generalize and inherit multiply.

[CF5] Review design

- Verify all system operations against collaboration diagrams.

- Verify timing requirements (nonfunctional) using sequence diagrams.

Fusion: Implementation

Implementation

- Most decisions already made.
- Should be straightforward.
- Mainly uses the design class diagrams, fully annotated with:
  - Operations with parameters and returns.
  - Data attributes.
  - Parent classes.

[CF1] Resource management strategy

- Create policy.
- Resources:
  - Files and handles.
  - Memory.
  - Windows descriptors.
  - Threads.
- GC is a solution.
- Chosen solution depends on programming language, quality criteria.

[CF2] Code arising from the data dictionary

The data dictionary is largely inapplicable.

[CF3] Code the class descriptions

- This may be generated automatically (e.g. by Rose).
- Design class descriptions give rise to the interfaces:
  - Visibilities.
  - Method signatures.
  - Mutability and Mutex issues.
  - Visible data members.
  - Inheritance structures.
  - Class invariants, pre- and post-conditions.
- Output: e.g. header files in C++.

[CF4] Code the method bodies

- Code can be lifted directly from the object collaboration diagrams.

- Check that invariants, pre- and post-conditions are respected.

[CF5] Performance analysis

- Use good profilers (instrumenting vs. sampling).
- Cross-check results with expectations (derived throughout the process).
- Cross-check with nonfunctional requirements.

[CF6] Code reviews

Inspections:

- Human inspection.
- Limited value.

Testing:

- Idiomatic (language specific).
- Low level.

Evolutionary Fusion

Key attributes:

- Multiobjective driven.
- Early, frequent iteration.
- Analysis, design, build, test in each cycle.
- User orientation.
- Fully systems-oriented approach (like Fusion).
- Result orientation, not purely SEP oriented.

Benefits

- Better match to customer need — explicit feedback loop.

- Hitting market windows:
  - Short cycles.
  - Risk management.
  - Divisibility into subteams.

- Engineer motivation and productivity.

- Quality control: ISO9K, TQM, etc. are applied more easily.

- Reduced risk in transition: move to OO can be done at the same time as the SEP change.

Costs

- Forces focused/efficient decision making.
- Good SEP is a must.
- Overheads are non-trivial.

[EVO1] Definition phase

- Fundamentally focused on communication and thought.
- Estimation of viability, cost, etc.
- Good architecture is critical.

Requirements

- Follow Fusion approach.
- Define the *value proposition*: articulation of why the customer will choose the system (over alternatives).
- Outputs:
  * Functional/nonfunctional requirements.
  * Requirements matrix.
  * Use-cases.

Analysis (first pass)

- Elaborate use cases.
- Domain class diagrams (domain experts).
- Analysis class diagrams.
- Expand scenarios so they correspond to the analysis class diagram.
- Analysis paralysis: change-density tracking.

Architecture

- Crucial to rapid cycles/releases, without redesigning.
- Use the standard Fusion methodology.
- Do not focus excessively on details
  - of class interaction, or
  - of component grouping.

Planning

Define key roles (depending on the project size, some could be assigned to more than one person or more than one person doing a single role).

- Project manager:
  - Work with marketing and client.
  - Co-ordinate creation of value proposition (focal point for key decisions).
  - Main decision making/prioritization.
  - Overall risk management.
  - Sequencing and insertion of cycles.
- Technical lead:
  - Architectural decisions.
  - Definition of cycles.

    * Deliverables, etc.
    * Insertion of new cycles.
    * Especially for architectural repair.
- User liaison:
    * Manages release distribution.
    * Collects information in the feedback cycle.

Define the standard EVO cycle.

- Define length: 1–4 weeks.
- Factors:
  - Management insight.
  - Adjustment cycle.
- Plan milestones.

Group and prioritize functionalities.

- Create 4-5 chunks.
- Prioritize use cases and group into the chunks (similar in size).
  - Risk.
  - Must-have, want,
  - Infrastructure.
- Prioritize within the chunks as well.
- Elaborate on the system operations for first chunk.
- Group them into some initial cycles.

  - HP: size of cycle should be half of team's estimate.

  - Initial success is crucial.

- Prepare task list for initial cycles (technical lead).

- Output: implementation schedule.

[EVO2] Development phase

This part is iterated.

Refining analysis

- Review existing analysis models.
- System operations to be implemented are checked against the analysis class diagrams.
- Architectural compromises are logged as defects for (architectural) repair later.

Design

- Updated according to Fusion.
- May lead to: new methods in pre-existing classes, or new classes.

Coding/validation

- Create test cases (on a local level) simultaneously (from the use cases).
- Use a test harness for early testing. Feedback
- Should operate simultaneously with other tasks.
- May use surrogate users in early phases.
- Managed by user liaison.
- Allocate time to review and strategize.

System test

- Apply system-wide tests — derive from the use cases.

- Maintain set of regression tests.