From Module Breakdown to Interface Specifications

Completing the architectural design of Map Schematizer
Defining module interfaces

• The module organization document describes each module very abstractly
  – Example: The coordinate mapping module has “region iterator” functionality, but doesn’t describe how to use it
  – It’s not enough for the implementer of that module or other modules that use it

• You need to design the concrete interfaces
Example: Region iterator

• What is it?
  – Does it return a list of regions?
  – Does it apply a provided method to each region (functional programming style)
  – Does it implement the Java iterable interface?

• Any of these would be possible concrete interfaces for the region iterator (but some are better than others)
What’s the interface spec?

- It includes the actual Java method signature(s)
  - So the user of the module knows how to call it
- It includes a precise description of what the method(s) do
  - Possibly in a formal notation like JML; more likely in (careful, precise) English comments
  - So the user of the module knows what to expect, and the implementor knows what must be done
What’s an abstract interface?

• Set of assumptions that the developers of one module can make about another (not just the signature)
  – What access methods can I invoke?
    • What are their parameters?
    • What exceptions can occur?
  – What is the effect of calling a method?
    • What are legal sequences of invocations of access methods?
      – Push(a)
      – Push(a).pop
      – Push(a).top
    • What is the effect on one access method of calling another?
      – Push(a).top returns a
      – Set(x).get returns x
  – What are the externally-visible states of the module?
• Definition of interface used within context of writing interface specifications is broader than definition used by languages such as Java
Goals for Creating Module Interface Specifications

- Clearly documents the behavior of the module
  - reduces time & knowledge required to adopt module
- Clearly documents the interfaces used by the module
  - Aids in creating stubs, mock interfaces and integration test scripts
- Improves the ability to isolate errors quickly
- Supports backwards compatibility
- Defines implementers work assignment
  - Interface specification is essentially a contract for the developer that specifies the implementer’s task and the assumptions that users can make
- Enables straight-forward mapping between use case requirements and methods
  - reduces effort required for requirements traceability
Module Interface Specifications

• Are not:
  – Completed at end of development as “after-thought”
  – Typical API which focuses on describing signature of interface
  – Lengthy, 100+ page document
  – Replacement for design documents

• What they are:
  – Critical step in design interval
  – Description of minimal set of methods
    • Do not include extra methods unless clearly understand how additional functionality would be used
  – Description of how to verify behavior of module in addition to how to use module
  – Easy to read, maintain
  – Means for supporting module extensibility

• Can be documented using Javadoc or other means.
  – Interface specifications should be completed before implementation
Abstraction and Interface Specs

• We strongly prefer an interface spec that is abstract in the sense that it hides implementation details
  – Allows the implementation to change without affecting other modules
• BUT ... you can’t really design a good interface without thinking about possible implementation(s)
A method for constructing abstract interfaces

- Define services provided and services needed (assumptions)
- Decide on syntax and semantics for accessing services
- In parallel
  - Define access method effects
  - Define terms and local data types
  - Define states of the module
  - Record design decisions
  - Record implementation notes
- Define test cases and use them to verify access methods
  - Cover testing effects, parameters, exceptions
    - Test both positive and error use cases
  - Cover broader set of tests than typical unit tests
  - Support automation
  - Design test cases before implementing module
    - Who is responsible for designing test cases?
Example: Region Iterator

- Returning a list of regions would be simplest to implement
  - But it’s apt to be wasteful: A list is constructed and then thrown away (pressure on the garbage collector)
- A map function is also fairly simple, but more work for the client
- The “iterable” interface is harder to implement but nicer to use
Simple to implement or simple to use?

- How much do we really care about efficiency?
  - Right now, we care a lot more about development schedule
  - But later efficiency might be an issue.

- We want a quick implementation now, and ability to improve efficiency later (if needed) without changing the interface
Abstraction to the Rescue

- Possible solution:
  - Implement the “iterable” interface
  - But do it quick and dirty:
    - Internally, build a list
    - Implement iterable by iterating over the list
    - Not efficient, but good enough for now
  - Later, the implementation can be changed
    - Implement the iterator methods directly, without building a list (but only if efficiency is an issue)
    - The interface spec stays the same
Other things to be defined ...

- What’s a region, anyway?
  - Probably an object with four fields: x1, y1, x2, y2
  - And maybe one more: scale factor
  - Public fields or getters and setters?
    - Is there really any reason for getters and setters here?
- How are they iterated?
  - Probably in order, left-to-right and top-to-bottom. But do you want to specify that? Could it change?
- Do we iterate over all regions, or within a range?
- Is there an implicit region with scale factor 1.0 to the left of all the shrunk regions?
Possible solutions

• What’s a region, anyway?
  – Region has public x1, y1, x2, y2: int
  – ScaledRegion extends Region adds scale: float

• How are they iterated?
  – Arbitrarily (unless a need for knowing order arises

• Iterate over a specified area (min and max x,y)
  • Does this require another class in Java? Maybe a ‘set-up’ call to set the bounds?

• Iterated regions completely cover the specified area, possibly with border areas (scaled 1.0)
Lots more ...

- Each module needs a module interface spec
  - A module might be one class, or several
- Interface specs include Java classes with public method signatures (and public fields if any)
- Interface specs also include semantics
  - This is important. It’s not a spec if it doesn’t say what the methods actually do.
Describing Semantics

• Typical “contract” style:
  – Precondition: Caller responsibility – what must be true when this method is called
    • Example: shrinkRegion(x1, y1, x2, y2, sf) might have precondition: y1 < y2, x1 < x2, 0.0 < sf ≤ 1.0
  – Postcondition: What is true at return
    • In OO, this typically requires having an abstract model of the information maintained by an object (e.g., the set of regions maintained by the coordinate mapping class). Don’t confuse this with the actual data structure!
More possible semantics

• Changes at most ...
  – Promises not to change anything outside the list.

• Unchecked vs checked preconditions
  – Checked: if the client violates this, the module will notice (and throw an exception, for example)
  – Unchecked: the client better not violate this, ever

• Except ... (describe what condition triggers each exception)

• Performance characteristics
  – E.g., is the “shrink” operation constant time, linear time, or does it depend on how regions overlap in a series of calls?
Map Interface Spec to Module Service

• Example: indicate the method(s) that correspond to “Iterate over regions”
  – In general, one “service” may map to multiple methods, which must be called in some sequence (e.g., set area bounds before iterating over regions)
  – One module may be implemented by more than one class (e.g., “ScaledRegion” as a way of delivering results)
Test cases, too!

• Test cases can (and should) be defined from the interface specification
  – And interface specifications should be testable. If you can’t distinguish correct from incorrect execution, the interface should be re-thought.

• Test cases should cover legal sequences of operations, and checked exceptional cases
  – Everything the module contract promises
Should you use UML?

+ UML is widely used and understood
+ UML can capture signatures and some relations reasonably well
– UML doesn’t help much with semantics
  Unless you also use OCL. Good luck with that.
– UML doesn’t help much with judging completeness, consistency, abstractness, or almost anything else important
• So go ahead if you want, but don’t confuse “nice UML” with “well-designed interface spec”
Summary

• You need to write module interface specs
  – Enough to form an agreement between developer of a module, and users of a module

• Interface specs require lots of design decisions, and some of them are hard

• You also need to express module interface specs clearly, precisely, and completely