

Reengineering a Mobile Nursing Information System

Larry LeFever

Abstract

This is a case-study of the value of the principles of object-orientation. The author was given the task of enhancing the substantial version 1.0 codebase of a certain mobile application, to enable that application to satisfy a crucial usage-requirement evidently not discovered or anticipated during the version 1.0 design-phase. The author attempts to demonstrate that the desired enhancement cannot be robustly and scalably achieved without a substantial re-design of the entire project, thanks largely to failures of the version 1.0 design to avail itself effectively of all of the benefits of the principles and techniques of object-orientation relevant to the project's requirements.

I. NIS Version 1.0 Overview

This project concerns the reengineering of the Nursing Information System (NIS), a mobile application developed for the Palm Pilot using Java's J2ME/MIDP platform (Java 2 Mobile Edition / Mobile Information Device Profile). The first version of the project was developed by a prior development-team.

This system was to be designed to enable nursing students to store on the mobile device data regarding one or more physical assessments of patients and then transfer those assessments to a desktop application for review and optional editing, optionally by others, such as the nursing students' professors.

The version 1.0 system, once initialized, presents the user with an image-map consisting of a reproduction of Leonardo Da Vinci's sketch of a man spread-eagled within a perfect circle. Relevant parts of the man's body are selectable with the Palm Pilot stylus. There's also an enlargement of just the head of the body. That enlargement enables the user to select a more specific part of the head, such as the eyes, nose, mouth or ears. Selection of any such body-part causes a menu to appear, allowing the user to select a physiological system relevant to the selected body-part. Such system-menus lead, in turn, to a set of option-lists, and, in some cases, also to one or more editable fields or text-areas. By the time the user has exited the system, the system has persisted the user's option-selections and, in relevant cases, his or her text-entries.

The user is then meant to use the Palm system's "Sync-Manager" to transfer the physical assessment data-sets to the desktop, to be viewed and optionally edited using the desktop version of NIS. A custom-developed "conduit" manages this transfer, in an NIS-proprietary file-format-specific way. [NIS1]

II. New Features Requested in version 2.0

The client indicated that, for this version of NIS, the most important, if not sole, requirement is that the system be enhanced to enable the user to create and store on the device multiple physical assessments. NIS version 1.0 did not, in fact, satisfy that requirement. Evidently, only late-stage usability-testing made that problem known.

The issue given secondary importance was the slow start-up process of the application.

III. New Features Implemented in version 2.0

The multiple physical assessment requirement was satisfied in version 2.0, the version developed by the author. This requirement was implemented by implementing the following new use-cases:

- “New Patient”
- “List All Patients”
- “Select Patient By ID”
- “Delete Patient By ID”

The design-issues to be discussed later in this paper forced the author to resort to an undesirable strategy for implementing this new functionality. That strategy could be considered “kludgy” or a work-around.

The principal motivation for resorting to such a work-around is a certain monolithic character of the version 1.0 design that forced the author to simply wrap, indirectly, a great deal of version 1.0 initialization-code, in such a way as to cause the application to nearly completely re-initialize itself whenever another patient data-set, new or not, is to be displayed.

Such repeated re-initialization is evidently causing or exacerbating performance-problems and sporadic memory-errors.

Some of the memory-errors are clearly stack-related. These are evidently caused at least partly by use of recursion that, in certain cases, could be avoided, provided there is a major re-design of the system, one that emphasizes “Lazy Initialization”, more about which follows.

IV. Performance-Problems

Certain performance-problems had been expected, but, it turned out, the causes of such problems were not the ones expected.

It was expected that, at least eventually, during the course of scaling upward towards greater and greater numbers of simultaneously stored physical assessments, there would be a “data-packing” issue. That is, the mobile device’s memory-constraints were expected to limit problematically the number and size of physical assessment data-sets.

The originally proposed solution to that problem was to finish implementing the wireless-connectivity feature prototyped as part of NIS version 1.0 (which, incidently, one should probably call version 1.1). However, the client decided that such connectivity could not be relied upon to be sufficiently available to the intended user-community. So, a “lowest common denominator” solution was deemed preferable, and that meant solving the multiple data-set problem without resorting to any means of network-connectivity.

So, the solution implemented in version 2.0 avoids network-connectivity and, instead, essentially builds around the principal part of the version 1.0 codebase – that is, that part of the codebase responsible for displaying a single editable data-set -- a wrapper that enables the user to specify which data-set to display. Once the user has selected a data-set, the system, unfortunately (as regards performance and memory-demand issues), proceeds to almost entirely re-initialize itself.

An alternative strategy was attempted, in the interest of avoiding such re-initialization. However, that strategy proved to be, though less of a performance-problem, more of a maintenance-problem. There was to be need for much tedious (and therefore error-prone) code-revision to implement a recursive data-reset (or field-clearing) feature.

So, the re-initialization-heavy solution seemed to be the best compromise, albeit a kludgy one. However, while testing this new version of NIS, certain memory-errors appeared occasionally.

Occasionally, a native-code error-message alert-box appeared, indicating that a stack-overflow was about to happen. Such errors required a “soft-reset”. At other times, there appeared in the standard-output log an OutOfMemoryError message.

Furthermore, a performance-problem that had existed already in version 1.0 continued to exist: that of the start-up process taking too long.

The version 1.0 solution to the long start-up process problem involves showing a progress-bar during the start-up process. However, the application seems to hang for at least a few seconds after the progress-bar has reached its full size, especially in the case of re-initialization (as opposed to the very first initialization).

One likely cause for this delay is the need for quite a lot of garbage-collection, at least in cases of re-initialization (which, incidently, was not an issue in version 1.0, because re-initialization never occurred, since only one data-set was supported). That is, the entire object-model loaded during any given prior initialization needs to be garbage-collected at the same time as the next instance of the object-model is being loaded or allocated. Furthermore, the monolithic aspect of this object-model loading-strategy prevents any objects of the current object-graph from being garbage-collected until its root-object is no longer referenced, which doesn't occur until the next initialization has begun. This apparently tends to cause garbage-collection to be frequently interleaved with the new allocations, which has the effect of slowing down the new initialization. There's also apparently another reason for the “hanging” after (re-)initialization.

It turns out that this hanging of the progress-bar is also partly caused by the fact that the progress it indicates is not *bona fide*. That is, the progress-bar grows at a fixed rate for a fixed span of time (thread-scheduling notwithstanding), regardless of the actual degree of progress of the process whose progress it claims to be indicating.

Closer inspection of the version 1.0 codebase indicates that several crucial design-errors were made. The design-errors in question seem to cause the aforementioned memory-errors and excessively long start-up process. The errors also cause major maintenance and extensibility problems.

The remainder of this paper consists of descriptions of certain of the details of the aforementioned design-errors, together with recommendations for eliminating those errors, largely through the implementation of certain of the “Gang-of-Four” (GoF) Design Patterns. [GHJV]

Finally, the author's examination of the version 1.0 codebase indicates that the robustness, performance and scalability required of a truly production-ready application cannot be achieved for NIS without a thorough re-design and re-implementation of the application.

V. Re-Design Recommendations

The following items summarize the re-design recommendations.

1.) Lazy Initialization:

The principal design-error of the version 1.0 codebase is, in the opinion of the author, the insistence of the design upon loading and initializing the entire data-set object-model, rather than loading and initializing only what's needed at any given time, much as is done in operating systems by way of virtual-memory paging. This preferred delayed, conditional, and temporary initialization is what's referred to, here, as “Lazy Initialization”.

There is, actually, a degree of Lazy Initialization implemented in version 1.0. However, it's implemented in such a way as to unnecessarily re-instantiate numerous UI-objects **when**

referencing one or another part of what's being called, here, the "object-model", which has, indeed, been fully constructed by then. That is, what happens lazily is the construction of, e.g, a menu. However, that menu's contents are drawn from the complete "object-model", which is larger, in terms of memory-demand, than any single menu and is, therefore, more in need of Lazy Initialization.

The GoF Flyweight Design Pattern [GHJV] is recommended as part of a solution to this problem. A "Flyweight", in GoF terminology, is a type of shared object with two types of state: "intrinsic" and "extrinsic". The intrinsic state is independent of context (e.g., a GUI-widget's general appearance); the extrinsic state is, essentially, that part of the context, at any given time, which the shared object needs to know about (e.g., which data the GUI-widget should be currently displaying). The "Flyweight" gets repeatedly re-initialized (but not re-instantiated), as appropriate for its current context.

The Flyweight Pattern would reduce the number of UI objects in memory at any given time. It would also de-couple domain-objects from UI-objects, by causing a given UI-object to be re-initialized, as needed, with the relevant domain-object's data, as opposed to assuming a one-to-one relationship between domain-objects and UI-objects, as is currently generally the case, in the version 1.0 codebase.

2.) Visitor Pattern:

Once Lazy Initialization has been implemented (or, better said, more fully implemented), there would be need to solve a new problem: that of updating data-sets in secondary storage without any longer being able to recursively traverse the entire object-model, which could no longer be relied upon to be in memory in its entirety at any given time.

The Visitor Pattern is recommended as part of a solution to this problem. The Writer object would "visit" each "dirty" object (i.e., each domain-object edited since last persisted), and each such "dirty" object would then call back the "visitor", calling that Writer-method of the "visitor" which is appropriate for persisting the calling "dirty" object. Incidentally, there might be, in any given case, only one such "dirty" object when the persistence-operation is to be carried out. The non-dirty fields of the raw data-set (cached since last read from secondary storage) would simply be re-written directly in their raw form, as part of the new stream of data being written to the relevant revised patient data-set file. The fields' IDs would be used as keys into a Hashtable. The Writer, while traversing the raw data-set and detecting field-IDs therein, would check the Hashtable for each such ID, to see if there's a "dirty" domain-object that needs to intervene in the Writer's default progress, to write itself to the file, as only it need know how to do, recursing through itself, if necessary. If there's no raw data-set yet (because this is a "New" patient), then the RecordStore-file will end up containing the persisted form of only the "dirty" domain-objects. This means that, in read-only usage-scenarios (i.e., those in which the user is not adding or updating any of a patient's data), there needn't be any file I/O at all. This also has a favorable side-effect: namely, that the file-format becomes more flexible and efficient, storing only what needs to be stored.

This part of the proposed solution would also involve storing each patient's data-set in its own "RecordStore"-file, rather than storing each in a Record stored in one "RecordStore"-file shared by all patient data-sets, as is the case in the NIS version 1.0 codebase.

Actually, there are two RecordStore-files: one for textual data; and one for option-selection data. Each patient, at any given time, may have data in either or both of these files. This further complicates efforts to enable the application to support multiple patient data-sets. This, furthermore, adds to the performance-problems, even though, on the Palm Pilot, the file-system is volatile and therefore accessible just about as fast as main memory.

This one-to-one strategy involving RecordStores and patient data-sets is recommended so as to avoid the need to re-write all patient data-sets whenever any of them needs to be updated,

which is currently required while all patient data-sets are stored in the same file. This is a scalability-issue: as the number of patient data-sets increases, so will the time required to update the entire shared RecordStore. That is, even if the user changes just one option-selection, all the data for all the patient data-sets will be re-written.

Incidentally, each file, while being updated, needs to be completely re-written, because its data-file format necessarily does not use fixed-sized text-fields – that is, in the case of that one of the two aforementioned files which stores textual data.

Also, though elimination of recursion is perhaps not necessary, since there would usually be only substantially fewer objects to recurse through than is currently the case, this recommendation could optionally enable the elimination of recursion, which is presumably the principal cause of the stack-overflow problem.

3.) Revision of the Progress-Indicator:

First of all, proper implementation of Lazy Initialization might prevent the need for any progress-indicator. However, if the need for one were to remain, the simplest solution to the progress-indicator problem (i.e., that of its failure to indicate actual application-initialization progress) is to cease attempting to enable it to indicate actual progress. Instead, one could use a “busy-bar” (i.e., a bar, or a spot of some kind, that oscillates back and forth, playing the role played sometimes, on the desktop, by a sand-clock type of cursor). If a progress-indicator is, instead, deemed required, then the author recommends a solution that, though fairly complicated, would, indeed, indicate actual progress.

Such a solution would involve an implementation of a kind of Observer Pattern (another GoF Pattern) [GHJV], such that the progress-indicator would play the role of Observer while each object that needs to initialize itself (in particular, during application-initialization) would register itself with the progress-indicator and then, probably while executing in its own thread, send events of some application-specific type to the progress-indicator so that, periodically, the progress-indicator could calculate the percentage of such registered objects that have so far completed their initialization and then display a progress-bar of correspondingly proportional width.

4.) Addressing the Maintenance Issues Involving the Domain-Data:

Largely to facilitate the revision and/or enlargement of the domain’s data-model, the author recommends storing the domain-data in a relational database and using an “ORM” (i.e., Object-Relational Mapper Utility) such as “Jet” [JET], to auto-generate source-code for domain data-classes. (“Domain-data”, in this case, refers to the categories of, and/or the criteria for, the physical assessment of patients.)

Jet, the ORM, uses the metadata of a relational database in order to generate an XML-representation of that database’s relational model and, through XSLT (Extensible Stylesheet Language Transformation), merges those data with an XSL-representation of the format of the Java source-code of the data-access-class appropriate for the relevant database table – and this for each of the relevant tables. For NIS, which does not itself use a relational database, each such generated class would presumably be a simple “value-object” (a J2EE Pattern), a class without methods except accessors and mutators (i.e., “getters” and “setters”).

This recommendation could be referred to as the DAO recommendation, i.e., the Data Access Object Pattern Recommendation (another J2EE pattern). DAO is not a GoF pattern, but it is, nevertheless, an important one, even for applications that do not directly access a relational database. Without it, in the case of NIS, changes to, and/or enlargements of, the data that define the categories or criteria of patient physical assessments would be markedly more error-prone, since such changes would require manual source-code editing.

5.) Using the Composite Pattern for Representation of Systems and Sub-Systems:

Two classes in the NIS version 1.0 object-model indicate the need for the Composite Pattern [GHJV]: AnatomySystem and AnatomySubsystem. The Composite Pattern conceptualizes an entity as consisting recursively of several instances of itself, in an abstract sense.

For instance, any file-system could be thought of as a Composite, in this sense: the root directory is a file that contains other files that are directories that contain other files, and so on. So, the root directory is really just a directory like any other, except that it has no parent. In other words, the distinction between “directory” and “sub-directory” may be thought of as a matter of state, i.e., whether or not the directory has a parent directory. If so, then it may be thought of as being in the “sub-directory” state.

The object-orientation principle of abstraction calls for the system-subsystem relationship to be generalized as a relationship between systems, such that a “subsystem” should be conceptualized as a system that has a parent system (or is in the “subsystem state”), while a “system”, simply, should be conceptualized as a system without a parent system (or that’s in the “parent system state”).

So, in a re-design involving an application of the Composite Pattern, there would be no class named “AnatomySubsystem”. However, the domain model would presumably need to continue storing, in some form, information about the relationships amongst systems.

In a relational model, there would presumably be a relation named System with a reflexive relationship between, say, “parent_sys_id”, as foreign-key, and “sys_id”, as primary key. The Object-Relational Mapper utility, e.g., “Jet”, would generate classes such as AnatomySystem, containing a member named “parentSysId”.

At NIS mobile-application runtime (as opposed to beforehand, when the ORM runs), a Builder object (“Builder”, another GoF Design Pattern) [GHJV], such as NIS version 1.0’s “BodyBuilder” class, would build (though only “on-demand”, as per the rules of “Lazy Initialization”) the requested System, as a Composite, having each such System-instance recurse, as needed, to load and initialize, from the selected patient’s RecordStore, all of the “subsystems” of the given system.

A custom utility (perhaps to be written in Perl, for instance) would generate a class of static constant arrays of “sysIds”, one for each “parent” System. This would involve the writing of an appropriate aggregating SELF-JOIN SQL-query within this utility.

VI. Conclusion

The design-issues cited in this paper, together with the recommendations for how best to resolve them, are quite well known and have been so for years – at least for about seven years, since the “Gang of Four” published their book of twenty-three design-patterns [GHJV].

In NIS version 1.0, several “broken wheels”, so to speak, were “re-invented”. The developers of version 1.0 had, as it were, “the shoulders of giants to stand upon” but either were not aware of such “giants” (i.e., the “Gang of Four” of Design Pattern fame) or they apparently failed to appreciate the value provided by those “giants”.

Perhaps the most problematical consequence of this lack of “pattern-awareness” is that the NIS project is now not likely to achieve production-quality without a very time-consuming major re-design and re-implementation that would not be necessary if the version 1.0 designers had been “pattern-aware”.

That is, it was discovered only very late in the prior development-iteration that a crucial requirement had been overlooked: that of the ability to store and manage multiple patient

data-sets on the mobile-device at once. This, by itself, assuming effective design, needn't have been a deal-breaker. However, with a design lacking pattern-awareness, the overlooked requirement cannot be implemented properly (i.e., robustly and scalably).

What more need be said about the value of the teaching of Design Patterns in computer science – or at least software engineering – curricula?

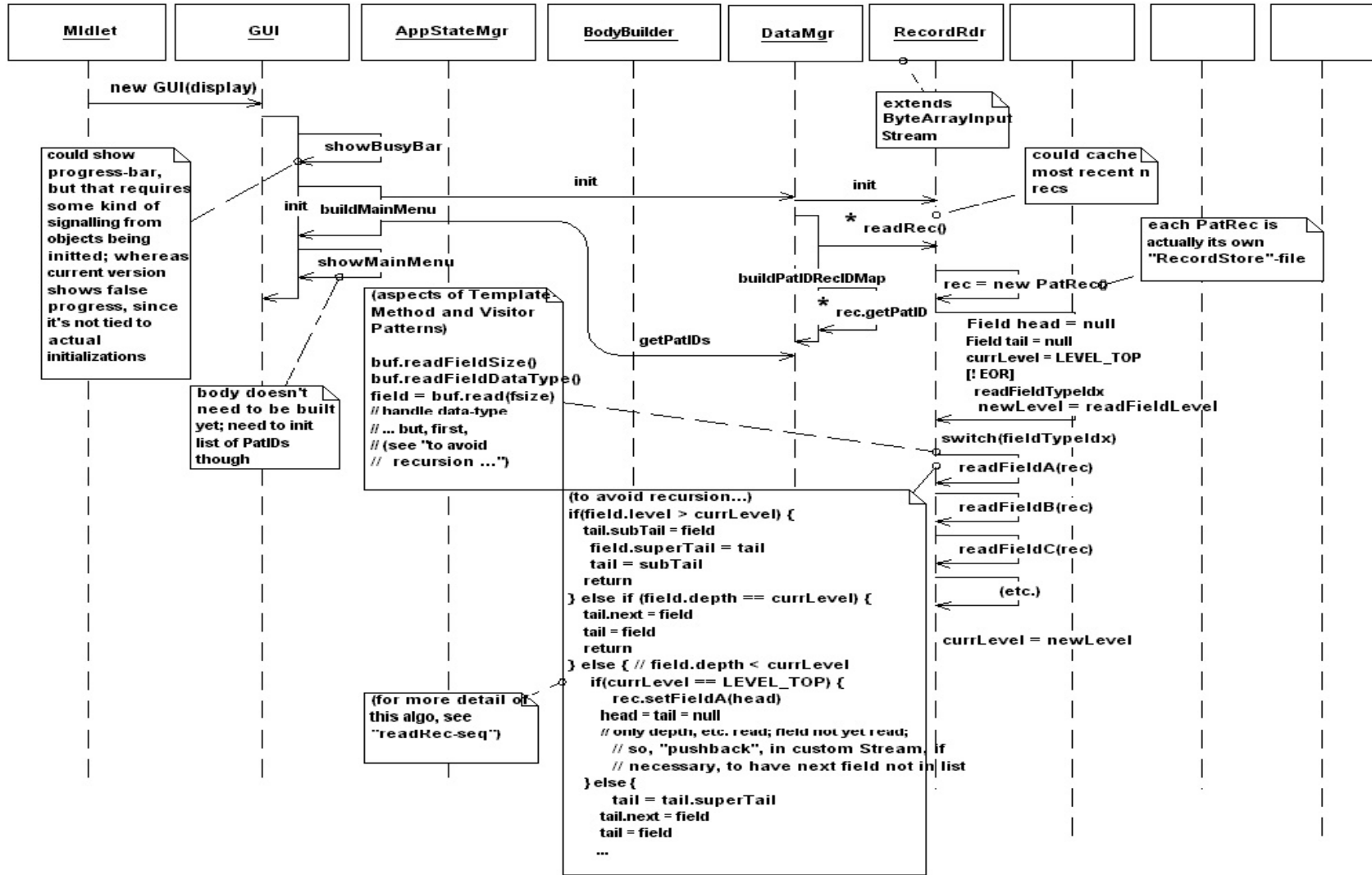
REFERENCES:

[GHJV] Design Patterns: Elements of Reusable Object-Oriented Software; Gamm, Helm, Johnson, Vlissides; Addison-Wesley Professional Computing Series, New York, NY, 1995.

[JET] <http://www.jetools.com/products/jetgen>

[NIS1] "Migrating an Application to Java2 Micro Edition: From Port to Portability"; Proceedings of MASPLAS 2002; Pace University, New York, NY; April 19, 2002.

NIS v3.0 Proposed Application-Initialization



NIS v3.0 Proposed Body-Loading

