Programming in the Small

David B. Gersten¹ and Steve G. Langer²

Academic medical centers, in general, and radiation oncology research, in particular, rely heavily on custom software tools and applications. The code development is typically the responsibility of a single individual or at most a small team. Often these individuals are not professional programmers but physicists, students, and physicians. While they possess domain expertise and algorithm knowledge, they often are not fully aware of general “safe coding” practices—nor do they need the full complexity familiar in large commercial software projects to succeed. Rather, some simple guidelines we refer to as “programming in the small” can be used.

KEY WORDS: Quality assurance, software design, medical informatics applications

BACKGROUND

Large institutions with dedicated software development staff develop and follow rigid methodologies for software development. Regardless of the methodology chosen, however, the details of its implementation are important. The methods used aid in defining the problem domain, discovering the steps required for achieving a solution and imposing sound programming principles, structure, and discipline on the development process. The benefits are a robust piece of software that is well documented, easily understood, and maintainable.

A large number of methodologies have been developed to address the difficulties associated with producing robust software systems. A simplified list could consist of:

1. Object oriented. Programming objects consist of a mixture of both data elements and the methods (aka functions) that operate on them.
2. Data driven design concentrates on the data to be used by objects. This is a relatively early approach and has been shown to produce code that is difficult to reuse.¹

3. Use case design defines actors in a system and then specifies what each actor does in response to an external input to the system.² The approach maps easily to objects.

Although they all may take different approaches and use different notations, there are common phases. These include (1) analysis, (2) design, (3) implementation, and (4) testing. Aside from design-focused differences, the methods also differ in how they move through the four stages; in the Waterfall (or linear) method, each of the steps is completed sequentially whereas in other methods (e.g., Agile and Extreme), an iterative process is used.³

The phrase “programming in the small” was coined by earlier investigators to make explicit the different paths used to create a program based on its scope.⁴ When programming “in the small,” the developer’s first inclination is to jump in and start writing code immediately. This ad hoc development process tends to result in unreliable, poorly written, nonscalable, undocumented, and limited use code. It would benefit the solo developer or small team to apply some kind of software development methodology.

Two problems stand in the way. First, while many people may perform software development, not all of them have had formal training in computer science and software development methodologies. Second, effectively applying the methods of a complex

¹From the William Beaumont Hospital, Royal Oak, MI, USA.

²From the Mayo Clinic, Rochester, MN, USA.

Correspondence to: Steve G. Langer, Mayo Clinic, Rochester, MN, USA; e-mail: langer.steve@mayo.edu

Copyright © 2010 by Society for Imaging Informatics in Medicine

Online publication 17 February 2010

doi: 10.1007/s10278-009-9271-z

design methodology is difficult. The solution is to define a simple set of methods and best practices that can be easily implemented on a small scale based on the four commonalities listed above.

METHODS

A useful way to start a software project is to state the problem and write a description of what the system is expected to accomplish. The benefits of this simple step are legion: It provides an outline of objectives to be accomplished, begins the documentation process, facilitates later maintenance, and updates either by the original author or a new person. This simple step can avoid “reinventing the wheel” because subsequent developers cannot understand the original code and have to start a project over from scratch.

The individual writing the application may be an expert in the problem domain and believe that they have a thorough understanding of the problem, but from what point of view? A medical physicist will understand external beam radiation therapy but are they also expert in capture and display of the entire treatment in real time, performing real-time quality assurance, and storing the resulting data to permanent archive for retrospective analysis using standards such as DICOM-RT? Such operations require a more granular level of understanding, such that the problem can be expressed in a computational language.

The authors have found the following object-oriented strategy for performing development in small project teams to be a fruitful approach. The method has several steps:

Analysis During analysis, the objective is to capture the requirements of the problem to be solved by developing a complete description. It begins with a layperson perspective and iterates through successive refinement to a detailed perspective. Analysis expresses what the system will do, not how it will do it. Commonly, the requirements are expressed in written form.

An effective method for discovering system requirements is by employing use cases and scenarios.⁵

Design The design phase is the “how” phase. How will requirements set out in the analysis phase be accomplished? This relates to the architecture of the system. From an object-oriented point of view,

this is where classes, their operations, and the interactions among them are outlined.⁶

Diagrams aid greatly in the design stage; diagrams convey a lot of information in a compact format. Many diagram standards have been developed to capture the structure, behavior, and interactions of the system as well as sophisticated software development tools to generate these diagrams.⁷ However, the objective of programming in the small is not to get bogged down in the details of how to draw the appropriate diagram or even what diagram is appropriate to use in which phase of development. Rather, diagrams are a powerful tool in discovering and expressing the content and intent of a software system. Two very basic types that are easily used and constructed are:

1. Object Diagrams—used to model the relationship between ‘things’ in the system
2. Domain Level Diagram—a macro view of the system

The value of spending time on the analysis and design phase cannot be over emphasized. The benefits are catching design flaws early in the process where they can be easily corrected, discovery of potential bugs, and conflicts before one line of code is written and smoother quicker implementation.

Implementation The implementation phase is where artifacts (requirements, diagrams, etc) from the analysis and design phases are turned into actual code. This should flow naturally from the objects identified from modeling. It is here where the programmer makes key decisions such as the choice of programming language. Assuming an object-oriented language is used, the programmer then must consider the class hierarchy that will be implemented, the data structures (hash table, linked list, array, etc.) and methods within them, and possible parallel processing via threading.

For any complex system, it is unlikely that all requirements and design parameters will be captured the first time through. Usually, an iterative process is needed where new requirements are discovered (e.g., based on hardware constraints or the need for added functionality). Classes may need to be altered, added, or discarded as behaviors and interactions are identified.⁸

Testing Testing is also an iterative process and should be used and updated throughout the

development process. An easy way to approach testing is to break it up into three levels.⁸

1. Unit Testing—Unit testing usually means writing and running tests to exercise a single class or even smaller unit, such as a function (e.g., a class that opens and reads a file from disk).⁹
2. Subsystem (or Module) Testing—Testing several units that collectively work together for a common purpose (e.g., opening and reading a file, inserting text, and writing the file back out to disk).
3. System Testing—End to send testing the system as a whole

Proper testing requires a plan using the software requirements from the analysis and design phase to create a test procedure. This should include test inputs with known correct outputs for each unit, module, and system level; error handlers should also be considered at this point (normal and abnormal exit conditions).¹⁰

RESULTS: PROGRAMMING A LINAC DATA CAPTURE SYSTEM

To illustrate the use of the model in practice, we will apply some object-oriented design principles used in the construction of a data capture system on a clinical linear accelerator (LINAC) in the radiation oncology department. The process begins with listing the requirements of the system.

Analysis The system is required to make a complete record of an external beam radiation therapy

treatment and write the resulting data to permanent storage. The data to be captured includes:

- Gantry angle
- Collimator information
 - Rotation
 - Jaw position
 - Leaves
 - Wedge
- Couch position
 - Height
 - Iso Rotation
 - Lateral
 - Longitudinal
 - Rotation
- Dose rate
- Energy
- Beam information
 - Name
 - Beam monitor units
- Patient information

The system will also:

1. Provide a user interface for starting, pausing, and stopping the recording process
2. Display in real time the state of the treatment
3. Be able to compare the current state of the treatment against the patient's treatment plan and visually indicate parameters that are out of tolerance
4. Be able to replay a previously captured treatment

Fig 1. A very simple view of the actors involved in the application so far.

Fig 2. Notice that *Patient* in Figure 1 has been replaced with the *RT planning computer*; the *LINAC Control* now contains an *External Communication Interface*, and *connections* between actors have been added showing potential transaction flow.

Design Domain level diagrams are used to discover the key *actors* and their potential interactions. Figure 1 illustrates the major actors of the system; each one is given a clearly defined label that conveys the item's purpose unambiguously. This diagram will be refined as more details emerge and subsystems are revealed.

Given that we desire to model not the patient but actually the patient's treatment, we see that Figure 1 is inaccurate; it is also incomplete. For example, some means is required to actually interact with the various components. Figure 2 adds some additional

details, updates the actor labels, and begins to chart the interconnecting *transactions*.

From Figure 2, it is clear that we are not going to directly communicate with the LINAC but instead use the External Communication Interface subsystem. As we iterate the design process, we learn successively greater detail about each actor; for instance, the LINAC can be modeled as shown in Figure 3, and the subcomponents of the collimator can also be seen. This “divide and conquer” approach facilitates detail when it is needed but permits abstraction to a simpler form

Fig 3. A cascading diagram showing the LINAC's subcomponents; a further cascade details the collimator subcomponents.

Fig. 4. This figure encapsulates the high level transactions needed for several use cases: 1a, send radiation therapy plan to archive, 1b, treatment capture computer retrieves plan from archive, 1c, plan is sent to LINAC controller, 1d, LINAC controller initiates plan execution. 2a, LINAC sends beam's eye view and dose to LINAC control. 2b, LINAC control system sends data to treatment capture system where operator notes plan deviation. 3a, treatment capture system sends plan correction to LINAC control system; 3b, LINAC responds. 4a, LINAC continues to report to LINAC control system; 4b, LINAC control sends data to treatment capture system; 4c, treatment capture system archives updated plan. 5a, at some time later, the physicist reviews the "as delivered" radiation therapy plan.

when looking at the higher level system as a whole.

After successive iterations, we see that the base level actors consist of the LINAC, radiation therapy (RT) planning computer, archive, real-time control system, and LINAC control systems. Specific use cases can now be discussed in the context of variations on Figure 2 where events and their resulting transactions are shown. For example, consider the use cases in Figure 4. Encoding use cases on actor-dataflow diagrams in this way reveals events that need to be addressed and transactions that flow as a result of those events. Naturally as detail increases, a specific diagram can be used per use case.

Implementation As a result of the preceding design work, we have already discovered many of the details that will need to be addressed in the program, uncovered potential bugs, and have a road map of how to proceed. We can now proceed to map actors in the diagram to classes in an object-oriented programming language. A useful approach is to map the actor *in the class* in the same manner it is mapped in the diagram. For example, the class LINAC would also contain subclasses for gantry, collimator, and couch. Extending this, the class collimator would also contain subclasses for jaw,

leaf, and wedge. The required physical parameters enumerated in the *Analysis* stage can now be added to the LINAC class.

As one proceeds to code, we have found that the following "best practices" listed below have served us well (at least in C++ and C#).

Classes

Name

Class names should reflect what the class represents so that someone reading the code will understand the intent immediately. Often the name of the class can be taken directly from the diagrams produced during the design phase. Typically, classes use nouns for their names.

During the analysis and design phase of our sample problem, it was discovered that a treatment session has at least one Beam and that this will need to be translated in an object in our implementation. A logical choice for the class name would therefore be CBeam.

Data Hiding

All class data should be designated as private using the access specifier private. Access to member variables is given through accessors only (member functions that set or get variables). This is referred to as "data hiding." In this way, other classes cannot mutate the data in unexpected ways.

Control is kept entirely within the class. Furthermore, when unexpected behavior does show up, it is much easier to debug due to the limited scope of the variable.

Atomicity

The class should be responsible for one thing and one thing only. For example, our CBeam class should not be directly involved in anything related to visualization. If the CBeam data needs to be visualized, another class should be responsible for this and that class should only query the CBeam class for data.

Coupling

Coupling or the dependency of one class on other classes should be avoided if possible or at least kept to minimum. The reason is that a change in one class can have a cascading effect and require changes in multiple other classes. Reuse of the class also becomes more difficult.

Header

Each class will have an associated header; an explanation of what the intended purpose of the class should be documented here, as well as the author(s), date, and revision number. We also like to include which development environment the code is being developed in, for example Visual Studio .NET 2005, as well as any external dependencies. Our classes always have the following template at the top:

```
Name: LINAC_control
Purpose: top level module for LINAC project
Author:
Creation Date:
Last Revised date:
Dev Environ: Vis Studio .NET 2005
External Dependencies:
```

Variables

Variables should be given names that are meaningful so their intended purpose is clear. The variable name “res” conveys no information but

the variable name `residual_error` clearly does. In addition, a comment should be placed next to the variable where it is initialized.

Private: float m_residual_error; // Residual error measured in millimeters

Initialize Variables

All variables must be given an initial value. If possible, they should be initialized with a value that would indicate an invalid condition. By doing this if the value is never set but the application tries to use it at a later time, it will be recognized as invalid, and appropriate action can be taken. An example would be validating a parameter, during the check, if the value must be ≤ 10 and it is set to 1,000, it will immediately be picked up.

Validation

Do not assume users will enter the data expected or users of a class will pass in the correct arguments. Always validate the input. Valid values for the data go back to the analysis and design phase of the project. For example, if the maximum allowable tolerance for a gantry angle is specified as plus or minus 1° and the user enters 55, unintended consequences will result.

Input arguments to all class methods must also be checked. A common problem is the passing in of a null pointer. At best, it will cause your program to crash; at worst, it will continue with unknown consequences.

Exception Handling

Use exception handling liberally. An exception is a special condition caused when something out of the ordinary occurred. Exceptions can be raised by the system such as a memory exception or raised by the application itself.

Exception handling provides a chance to correct the problem, notify the user of the problem, and continue or gracefully exit from the application. Unhandled exception typically result in catastrophic failure; however, if the application does not crash, unintended behavior will occur. What follows is an example of using exception handling:

```

long textinput_reader::open_source(CString filename,CString&
errmsg)
{
    // Purpose:Open the text file specified in input arg
    filename for reading.
    // Return: If file open is successful 0 otherwise a long
    value indicating the OS's error code and fills in the input
    argument errnmsg with the appropriate error message.

    char  cause[256];
    BOOL rc = FALSE;

    TRY
    {
        rc = file.Open(
            filename,CFile::modeRead|CFile::typeText,&e );
        file_open = TRUE;
    }
    CATCH(CFileException e)
    {
        e.GetErrorMessage(cause, 255);
        errmsg.Format("Open File %s
Failed.\r\nCAUSE:%s",filename,cause);
        return(e.m_lOsError);
    }
    return(0);
}

```

Return Values

If the code executed in a method can fail, it must return a value to the caller indicating success or failure. The return value of all method calls must be checked. Do not assume that a method has completed successfully.

Hard Set Values

Do not hard set values. This takes the responsibility of system configuration away from the user. For example, in the case of the linear accelerator collimator, it has a collection of leaves. The leaf positions can be stored in an array. If the array size is hard set and

another linear accelerator collimator has a different number of leafs, an array overrun will occur resulting in unexpected behavior. It is better to read the value in from a configuration file or query the user and then dynamically allocate the required storage space.

Method Header

Each class method should contain a header that explains the methods purpose, an explanation of input arguments, and outputs and return values. It is also handy to include the name of the calling method or methods for tracing through the code and for debugging.

Example

```

int CTxSession::get_beam_count( bool exclude_qa_beam)
{
    // Purpose: A TxSession contains a collection of CBeam object. Loop
    // through all the beams and if the beam is not a QA beam (quality
    // assurance) then add it to the count.
    // Return: number of beams or -1 on error
}

long CBeam::read_data(textinput_reader* ir)
{
    // Purpose: Read in the data associated with the beam
    // Caller : CTxSession::process_beam_data
    // Arg    : pointer to text file reader object
    // Return : 0 OK else error
}

```

Comments

Use comments though out your code. By reading the comment, someone unfamiliar with your code should be able to have a basic understanding of what each class and method does. It also aids in debugging, updating, and maintenance of code; do not assume that you will remember what was intended of code that was written a year ago or even a month ago.

Testing Test cases derived directly from the use cases developed in the analysis and design phases were constructed and run throughout the development process. For example, use case 1 in Figure 4 depicts the flow of the RT plan throughout the system. A unit test was performed to verify that the correct RT plan was retrieved from the archive, without error and with acceptable performance. A subsystem test went further to verify the plan was valid and was sent to all components in the system. Finally, an end-to-end system test verified the correct behavior on the LINAC and confirmed that the reported output agreed with real-world measurements (e.g., phantom dosimetry). For each test (unit, module, and system) that was performed with “validated” data, the tests were also repeated with plans where one or more variables were set to invalid values; this approach was used to validate the error handlers in the developed software. In at least one instance, this revealed a condition in an error handler that would have had significant impacts

(an impossible gantry rotation angle that would have been sent to the LINAC).

CONCLUSIONS

Scientific investigators can come from various backgrounds (medical, physics, biology, etc), but the one thing they often have in common is the need to conduct experiments and collect and analyze data.^{11, 12} This often involves computers and software, the latter of which may not always be commercially available for the task at hand. Hence, regardless of their formal training, the aforementioned investigators often find a common need to write their own software, without the benefit of formalized training in that area. We hope the guidelines shared in this work will aid those efforts to become more structured and supportable, without unduly complicating the primary goal of writing software to advance science.

REFERENCES

1. Wirfs-Brock R, Wilkerson B: Object-oriented design: a responsibility-driven approach. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (New Orleans, Louisiana, USA, October 2–6, 1989). OOPSLA '89. ACM Press, New York, 1989, pp 71–75
2. Cockburn A. Writing Effective Use Cases. New York: Addison-Wesley Longman Publishing Co., 2001
3. <http://www.agilemodeling.com/essays/umlDiagrams.htm>. Last viewed: October 2009

4. DeRemer, F, Kron H: Programming-in-the large versus programming-in-the-small. In: Proceedings of the International Conference on Reliable Software. ACM, Los Angeles, CA, 1975, pp 114–121
5. <http://alistair.cockburn.us/Why+I+still+use+use+cases>, Last viewed October 2009.
6. Horstman C: Mastering Object-Oriented Design in C++. New York: Wiley, 1995
7. Alhir S: UML in a Nutshell. Sebastopol, CA: O'Reilly & Associates, Inc., 1998
8. Booch G: Object-Oriented Analysis and Design with Applications, 2nd edition. New York: Addison-Wesley, 1994
9. Martelli, A: Python in a Nutshell, 2nd edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2006
10. http://standards.ieee.org/reading/ieee/std_public/description/se/1008-1987_desc.html. IEEE *Standard for Software Unit Testing*, Last viewed October 2009.
11. Langer S, Kanal K: Spreadsheets for automated data collection, analysis, and report generation for diagnostic medical physics: publicly available on the world wide web. *J Digit Imaging* 15(2):98–105, 2002
12. Langer S: OpenRIMS: an open architecture radiology informatics management system. *J Digit Imaging* 15(2):91–97, 2002