CS620 Midterm Study Guide

Author: Elais Jackson

Created: 10/03/2013

Version: 1.0

Table of Contents

No Silver Bullet: Essence and Accidents of Software Engineering

Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past profess has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Does it have to be hard? Essential difficulties

- Not only are there no silver bullets in view, the nature of software maeks it unlikely that any will ever come up.
- The difficulties of software:
    - *Essence*: the difficulties inherent in the nature of software
    - *Accidents*: those difficulties that today attend its production but are not inherent.

> *The hard part of building software is the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.*

- Inherent properties of irreducible essence of modern software systems: complexity, conformity, changeability, and invisibility.
    - *Complexity*. Software entities are more complex for their size that perhaps any other human construct.
        - Because no two parts are alike. Software systems differ from systems where repeated elements abound.
        - Descriptions of a software entity that abstract away its complexity often abstract away its essence.
    - *Conformity*. The software must conform because it is the most recent arrival on the scene.
        - Much of the complexity one has to master is arbitrary complexity.
    - *Changeability*. The software system embodies its function, and the function is the part that most feels the pressures of change.
        - The software entity is constantly subject to pressures for change.
    - *Invisibility*. Software is invisible and unvisualizable.
        - The reality of software is not inherently embedded ins pace.

> *Despite progess in restricting and simplifying software structures, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools.*

Past breakthroughs solved accidental difficulties.

- *High-level Languages*. Surely the most powerful stroke for software productivity has been the progessive use of high-level languages for programming.
    - The most a high-level language can do is to furnish all the constructs that the programmer imagines in the abstract program.
- *Time-sharing*. Time-sharing brought a major improvement in the productivity of programmers and in the quality of their product.
    - Time sharing preserves immediacy, and hence enables one oto maintain an overview of complexity.
- *Unified programming environments*. They attack the accidental difficulties that result from using individual programs *together* by providing integrated libraries, unified file formats, and pipes and filters.

    - Conceptual structures that in principle could always call, feed, and use one another can indeed easily do so in practice.

Promising attacks on the conceptual essence

- All attacks on the accidents of the software process are fundamentally limited by the productivity equation
    - *time of task* = the sum of all frequencies multiplied by time
- If the conceptual components of the task are taking most of the time, then no amount of activity on the task components that are memely the expression of the concepts can give large productivity gains.
- *Buy versus build*. The most radical possible solution for constructing software is not to construct it at all.
    - It will ALWAYS be cheaper to buy software rather than build it. The key issue being applicability.

Some Software Engineering Principles

David L. Parnas

- Software *engineering* is the design of useful programs under one or both of the following conditions:
    1. More than one person is involved in the construction and/or use the program, and
    2. More than one version o fthe program will be produced.
- In multiperson programming we find six problems that are not significant in the solo-programming situation:
    1. How to divide the job of producing the software among the programmers.
    2. How to divide the job of producing the software among the programmers.
    3. How to communicate to all the people involved information about the occurrence of run-time errors among the system components and to the user.
    4. How to write programs that are easily modified. Programs in which a change of one design decision does not require changes in many parts of the program.
    5. How to write programs with useful subsets. If we only need a subset of the services performed by a a program we should be able to quickly remove unneeded parts without having to rewrite the remainder.
    6. How to write programs that are easily extended. We should like to be able to add new capabilites to programs without rewriting the programs that are already present. This, too, is a fail soft goal; build a subset to meet a deadline, then

extend as time permits.

What is a Well-Structured Program

- *Structure* refers to to a partial description of a system.
- *Connection*. The connection between program parts are the assumptions that the parts make about each other.
    - This definition of connection can be clarified by considering two situations in which the structure of a system is terribly important:
        1. Making of changes in a system, and
        2. proving system correctness.

What Is a Module?

- During the preparation of software, there are several tiems at which parts are joined together. For example:
    1. The products of varous programmers are put together.
    2. Subroutines are "linked" together.
    3. Segments of a system are loaded into memory.
    4. Programs to perform various functions are combined.

Controlling the Structure of System Programs

There are two basic functions that a designer must perform very carefully: *decomposition* and *specification*.

- One conclusion of the previous sections was that a well-structured program was one with minimal interconnections between its module.
    - Systems that are well interconnected are difficult to change.
    - If we succeed in defining interfaces that are sufficiently "solid" (i.e, they will not change), changes in the system can be confined to one module.
    - Two common approaches to software module specification is,
        1. to reveal a rough description of the internal structure of the module,
        2. to reveal a description of a "hypothetical" implementation of the module.
    - Both approaches are fraught with danger.
    - It is extremely difficult to reveal some part of a n implementation (in order to be precise) and then instrucxt the reader as to precisely which parts of the revealed information he must not use.

Hierarchical Structure and Subsetable Systems

- It is important to design a "uses" hierarchy before coding actually begins.

Designing Abstract Interfaces

- Best practices procedure for interface design
    1. The development of a list of assumptions believed unlikely to chagne during the life cycle of the product, and
    2. The specification of a set of interface functions whose implementability is guaranteed by those assumptions.

On the Criteria To Be Used In Decomposing Systems into Modules

> *Usually nothing is said about the criteria to be used in dividing the system into modules. this paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.*

- The benefits expected of modular programming are,
    1. Managerial development time should be shortened
        - because separate groups should work on each module with little need for communication.
    2. product flexibility
        - it should be possible to make drastic changes to one module without a need ot change others.
    3. comprehensibility
        - it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

What is Modularization?

- In some cases a module is considered to be a responsibility assignment rather than a subprogram.
- Each module should hide some design decision from the rest of the system.

Use of the Concept of Transparency in the Design of Hierarchically Structured Systems

- Each level in a hierarchically structured system provides a virtual machine which hides (or abstracts from) some aspects of the machine below it.
- There is risk that we may eliminate some essential capability of the system if we do not abstract some parts of the machine.

The "Top Down" or "Outside in" Approach

- There are a number of difficulties with the outside-in approach.
    1. The necessary specification of the "outside" is often difficult to obtain.
        - It is difficult to express such design decisions precicsely without implying addituional internal, design decisions.
    2. The derivation of a design from such a specification is often not feasibile.
        - The set of possible internal structures for a given external specification is so large that one needs some additional constrainst before asearch can be begun.
        - These constraints are usually information about the "inside" (e.g the hardware).
    3. In attempting to follow the "outside in" procedure it is quite easy to specify internal mechanisms which would simplify implementation of the desired outside but would themselves be impractical to implement.
    4. It is difficult to apply this method if one is actually designing a set of systems whose only description is "general purpose"
    5. The application of this method may result in a piece of software which is unnecessarily inflexible.
    6. It is quite comon to design software in a situation where the inside is already fixed.
- For these reasons it is necessary to abandon the pure "outside-in" approach and adopt some additional procedures which are actually of an "inside-out" or "bottom-up" nature.

Structured Design

- Structured design is a set of proposed general program design considerations and techniques for making coding, debugging, and modification easier, faster, and less

expensive by reducing complexity.

General considerations of Structured Design

- Simplicity is the primary measurement recommended for evaluating alternative designs relative to reduced debugging and modification time.
    - Simplicity can be enhanced by dividing the system into separate pieces in shuch a way that pieces can be considered, implemented, fixed, and changed with minimal consideration or effect on the other pieces of the system.
- Observability (the ability to easily perceive how and why actions occur) is another useful consideration that can help in designing programs that can be changed easily.
- Consideration of the effect of reasonabile changes is also valuabe for evaluating alternative designs.
- The term *module* is used to refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names.

> *Considerations are always with relation to the program statements as coded, since it is the programmer's ability to understand and change the source program that is under consideration.*

Coupling and Communications

- To evaulate alternatives for dividing programs into modules, it becomes useful to examine and evaluate types of "connections" between modules.
    - A *connection* is a reference to some label or address defined (or also defined) elsewhere.
        - The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other modules.
        - Minimizing connections between modules also minimizes the chance of ripple effects.
- *Coupling* is the measure of strength of association established by a connection from one module to the other.
    - Strong coupling complicates a system since a module is harder to understand, change, or correct by iteself if it is highly interrelated with other modules.

    - Complexity can be reduced by designing systems with the weakest possible coupling between modules.
    - Coupling depends on
        1. how complicated the connection is,
        2. on whether the connection refers to the module itself or something inside of it, and
        3. what is being sent or received.

The Open-Closed Principle

“

> *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

- When a single change to a program results in a cascade of changes to dependent modules, that program exhitbits the undesirable attributes that we have come to associate with "bad" design.
    - The open-closed principle attacks this in a very straightforward way.
        - It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Description

- Modules that conform to the open-closed principle have two primary attributes.
    1. They are "Open For Extension". This means that the behavior of the module can be extended. We can make the module behave in new and different way sas the requirements of the application change, or to meet the needs of new aplications.
    2. They are "Closed for Modification". THe source code of such a module is inviolate. No one is allowed to make source code changes to it.

The Liskov Substitution Principle

- The primary mechanisms behind the Open-Closed principle are abstraction and ploymorphism.
    - In statically typed languages one of the key mechanisms that supports abstraction and polymorphism is inheritance.
    - It is by using inheritance that we can create derived clases that conform to the

abstract polymorphic interfaced defined by pure virtual functions in abstract base classes.

> *Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.*

- If there is a function that does not conform to the LSP, then that function uses a pointer or reference to a base class, but must know about all the derivatives of that base class.
    - Such a function violates the Open-Closed Principle because it must be modified whenever a new derivative of the base class is created.
- A model, viewed in isolation, can not be meaningfully validated.
    - The validity of a model can only be expressed in terms of its clients.
- When considering whether a particular design is appropriate or not, one must not simply view the solution in isolation. One must view it in terms of the reasonable assumptions that will be made by the users of that design.
- LSP makes clear that in OOD the ISA relationshp pertains to *behavior*.
    - Not intrinsic private behaviour, but extrinsic public behavior; behavior that clients depend on.
- In order for the LSP to hold, and with it the Open-Closed principle, all derivatives must conform to the behavior that clients expect of the base classes that they use.

Design By Contract

- There is a strong relationship between the LSp and the concept of Design by Contract as expounded by Bertrand Meyer.
    - Using this scheme, methods of classes declare preconditions and postconditions.
        - The precondtions must be true in order for the method to execute. Upon completion, the method guarantees that the postcondition will be true.

> *... When redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.*

- When using an object through its base class interface, the user knows only the

preconditions and postconditions of the base class.

- Derived objects must not expect users to obey preconditions that are stronger than those required by the base class.
    - They must accept anything that the base class could accept.
- Derived classes most conform to all the postcondtions of the base.
    - Their behaviors and outputs must not violate any of the constraints established for the base class. Users of the base class must not be confused by the output of the derived class.