Design Defects and Restructuring

Lecture 7

Sat, Nov 6, 2021

Reliability

- It is defined as the combination of correctness and robustness or more straightforward, as the absence of bugs

Reliability

- There are three reasons devoting particular attention to reliability in the context of object-oriented development:
  - The cornerstone of object-oriented technology is **reuse**
    - For reusable components, the potential consequences of incorrect behavior are more serious than for application specific developments
  - Proponents of object-oriented methods make strong claims about their beneficial effect on software **quality**
    - Reliability is certainly a central component of any reasonable definition of quality as applied to software
  - The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing **reliability**

Defensive Programming

- Defensive programming directs developers to protect every software module against the slings and arrows of outrageous fortune
  - This encourages programmers to include as many checks as possible, even if they are redundant
  - Advice → if they do not help, at least they will not harm
  - This approach suggests that routines should be as general as possible
- A partial routine is considered dangerous because it might produce unwanted consequences if a caller does not abide by the rules
  - Adding possibly redundant code "just in case" only contributes to the software's complexity – the single worst obstacle to software quality in general, and to reliability in particular

Defensive Programming

- The result of such blind checking is simply to introduce more software
  - Hence more sources of things that could go wrong at execution time
  - Hence the need for more checks, and so on ad infinitum
- Obtaining and guaranteeing reliability requires a more systematic approach
  - Software elements should be considered as implementations (meant to satisfy well-understood specifications), not as arbitrary executable texts

Design by Contract

- Preconditions
  - It defines the conditions under which a call to the routine is legitimate
    - It is an obligation for the client and a benefit for the supplier
  - The precondition expresses requirements that any call must satisfy if it is to be correct
  - The stronger the precondition, the heavier the burden on the client and the easier for the supplier
    - The matter of who should deal with abnormal values is essentially a pragmatic decision about division of labor
    - The best solution is the one that achieves the simplest architecture
    - If every routine and caller checked for every possible call error, routines would never perform any useful work

Design by Contract

- Preconditions
  - Weak – good news
  - Strong – bad news
- A precondition violation indicates a bug in the client (caller)
  - The caller did not observe the conditions imposed on correct calls
- Who should check?
  - The rejection of defensive programming means that the client and supplier are not both held responsible for a consistency condition
  - Either the condition is part of the precondition and must be guaranteed by the client, or it is not stated in the precondition and must be handled by the supplier

Design by Contract

- Which of these two solutions should be chosen?
  - There is no absolute rule
  - Several styles of writing routines are possible, ranging
    - From “demanding” ones where the precondition is strong (putting the responsibility on clients)
    - To “tolerant” ones where it is weak (increasing the routine’s burden)
  - Choosing between them is to a certain extent a matter of personal preference
  - The key criterion is to maximize the overall simplicity of the architecture

Design by Contract

- Post conditions
  - It defines the conditions that must be ensured by the routine on return
    - It is a benefit for the client and an obligation for the supplier
  - The postcondition expresses properties that are ensured in return by the execution of the call
  - Weak – bad news
  - Strong – good news (*you must deliver more results*)
- A postcondition violation is a bug in the supplier (routine)
  - The routine failed to deliver on its promises

Design by Contract

- Class invariants
  - Preconditions and post conditions describe the properties of individual routines
  - There is also a need for expressing global properties of the instances of a class, which must be preserved by all routines
  - Such properties will make up the class invariant, capturing the deeper semantic properties and integrity constraints characterizing a class
  - A class invariant is a property that applies to all instances of the class, transcending particular routines

Design by Contract

- Class invariants
  - Two properties characterize a class invariant:
    - The invariant must be satisfied after the creation of every instance of the class
      - This means that every creation procedure of the class must yield an object satisfying the invariant
    - The invariant must be preserved by every exported routine of the class
      - Any such routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry

Example Contract

Party	Obligations	Benefits
Client	Provide letter or package of no more than 5 kgs. each dimension no more than 2 meters. Pay 100 francs.	Get package delivered to recipient in four hours or less.
Supplier	Deliver package to recipient in four hours or less.	No need to deal with deliveries too big, too heavy or unpaid

A routine equipped with assertions

- *routine-name* (*argument declarations*) **is**
- -- Header comment
- **require**
- *Precondition*
- **do**
- *Routine body, i.e. instructions*
- **ensure**
- *Postcondition*
- **end**

Example

- *put_child* (*new*: *NODE*) **is**
- 	-- Add *new* to the children of current node
- **require**
- 	*new* /= *Void*
- **do**
- 	. . . Insertion algorithm . . .
- **ensure**
- 	*new.parent* = *Current*;
	child_count = **old** *child_count* + 1
- **end** – *put_child*

The *put_child* Contract

Party	Obligations	Benefits
Client	Use as argument a reference, say new, to an existing node object.	Get updated tree where the Current node has one more child than before; *new* now has Current as its parent.
Supplier	Insert new node as required.	No need to do anything if the argument is not attached to an object.

Substitutability: Require no more, promise no less

- ```
  class Version1 {
    public:
      int f(int x);
      // REQUIRE: Parameter x must be odd
      // PROMISE: Return value will be some even number
  };
  ```
- ```
  class Version2 extends Version1 {
    public:
      int f(int x);
      // REQUIRE: Parameter x can be anything
      // PROMISE: Return value will be 8
  };
  ```

Design Patterns – GoF

- There are 23 design patterns mentioned in the book
- These patterns are grouped as
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

Design Patterns

A design pattern is a general reusable solution to a commonly occurring problem in software design

Design Patterns

A design pattern is not a finished design that can be transformed directly into code

It is a description or template for how to solve a problem that can be used in many different situations

Design Patterns

Not all software patterns are design patterns

For instance, algorithms solve computational problems rather than software design problems

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Abstract Factory

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Applicability
  - A system should be independent of how its products are created, composed, and represented
  - A system should be configured with one of multiple families of products
  - A family of related product objects is designed to be used together, and you need to enforce this constraint
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory