Design Defects and Restructuring

LECTURE 13

SAT, NOV 30, 2019

Reliability

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

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

Pre conditions

- 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
- Weak – bad news
- Strong – good news (*you only have to deal with a limited set of situations*)

A precondition violation indicates a bug in the client (caller)

- The caller did not observe the conditions imposed on correct calls

Design by Contract

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

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 – good news
- Strong – bad news (*you have to 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

- Pre conditions 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
- 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
};
```

Principles of Package Design

Granularity – The Principles of Package Cohesion: They help us allocate classes to packages

- The Reuse Release Equivalence Principle (REP)
- The Common Reuse Principle (CRP)
- The Common Closure Principle (CCP)

Stability – The Principles of Package Coupling: They help us determine how packages should be interrelated

- The Acyclic Dependency Principle (ADP)
- The Stable Dependency Principle (SDP)
- The Stable Abstraction Principle (SAP)

Granularity

In the UML, packages can be used as containers for a group of classes

By grouping classes into packages we can reason about the design at a higher level of abstraction

The goal is to partition the classes in your application according to some criteria, and then allocate those partitions to packages

The relationships between those packages expresses the high level organization of the application

Granularity

What are the best partitioning criteria?

What are the relationships that exist between packages, and what design principles govern their use?

Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?

How are packages physically represented? In C++? In the development environment?

Once created, to what purpose will we put these packages?

The Reuse Release Equivalence Principle (REP)

The granule of the reuse is the granule of the release

The granule of the reuse (package) can be no smaller than the granule of release

If a package contains software that should be reuse, then it should not also contain the software that is not designed for reuse

Either all of the classes in a package are reusable or none of them are

Concept of reusability, concept of re user

The Common Reuse Principle (CRP)

The classes in a package are reused together

If you reuse one of the classes in a package, you reuse them all

This principle helps us to decide which classes should be placed into a package

It states that classes that tend to be reused together belong in the same package

Re-distribution problems

The Common Closure Principle (CCP)

The classes in a package should be closed together against the same kind of changes

A change that affects a package affects all the classes in that package and no other packages

SRP re-stated for packages

It is closely associated with OCP

Stability

The classic definition of the word stability is “Not easily moved”

Stability is not a measure of the likelihood that a module will change; rather it is a measure of the difficulty in changing a module

Modules that are more difficult to change, are going to be less volatile

The harder the module is to change, the more stable it is, the less volatile it will be

Classes that are heavily depended upon are called “Responsible”

Responsible classes tend to be stable because any change has a large impact

The most stable classes of all are classes that are both Independent and Responsible

Such classes have no reason to change, and lots of reasons not to change

The Acyclic Dependency Principle (ADP)

Allow no cycle in the package dependency graph

Morning-after syndrome

The weekly build – Partitioning the development environment into releasable packages

Breaking the cycle

- Apply DIP
- Create a new package, move the classes that they both depend on into that new package

Top-Down design

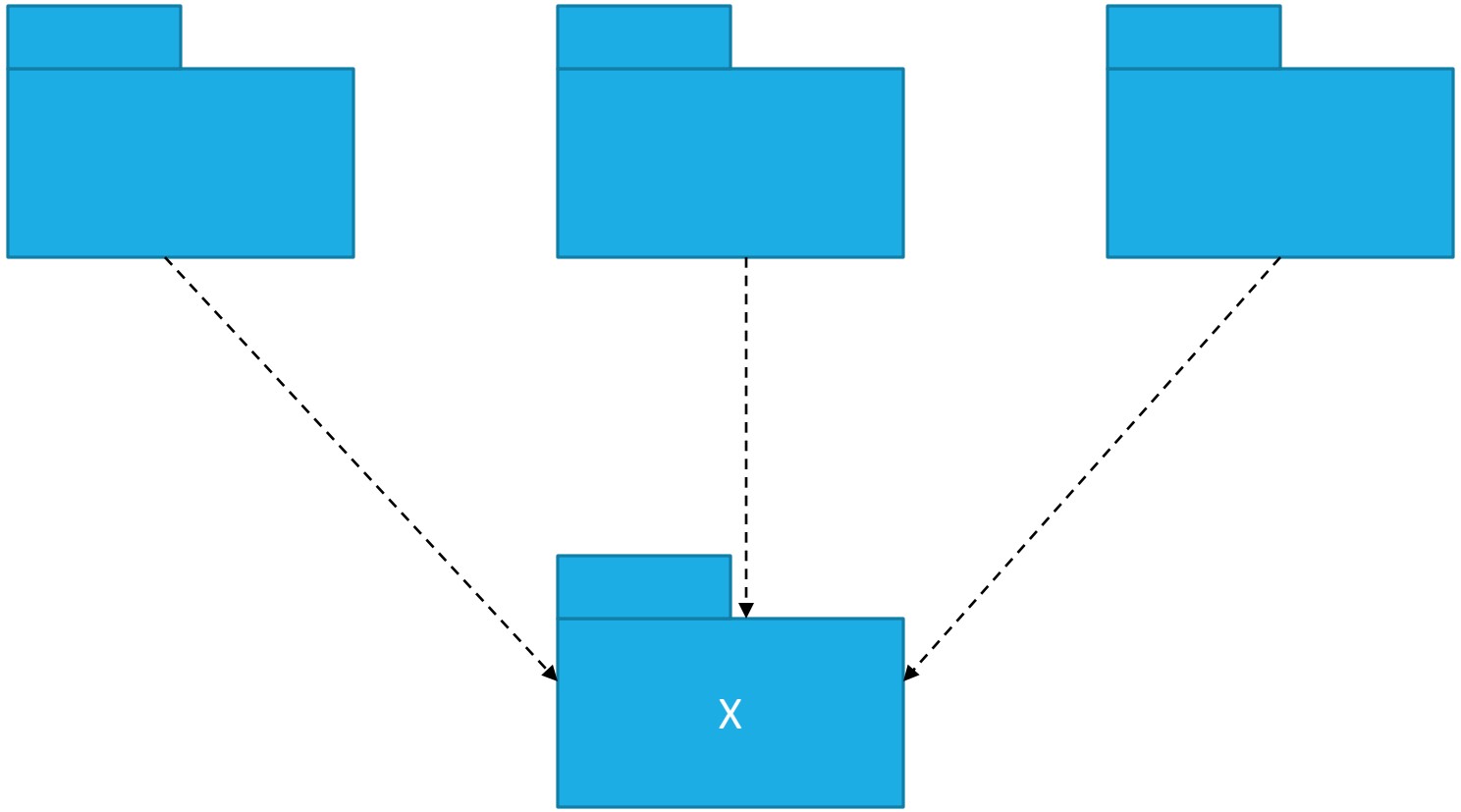
The Stable Dependency Principle (SDP)

Depends in the direction of stability

Any package that we expect to be volatile should not be depended on by a package that is difficult to change, otherwise the volatile package will also be difficult to change

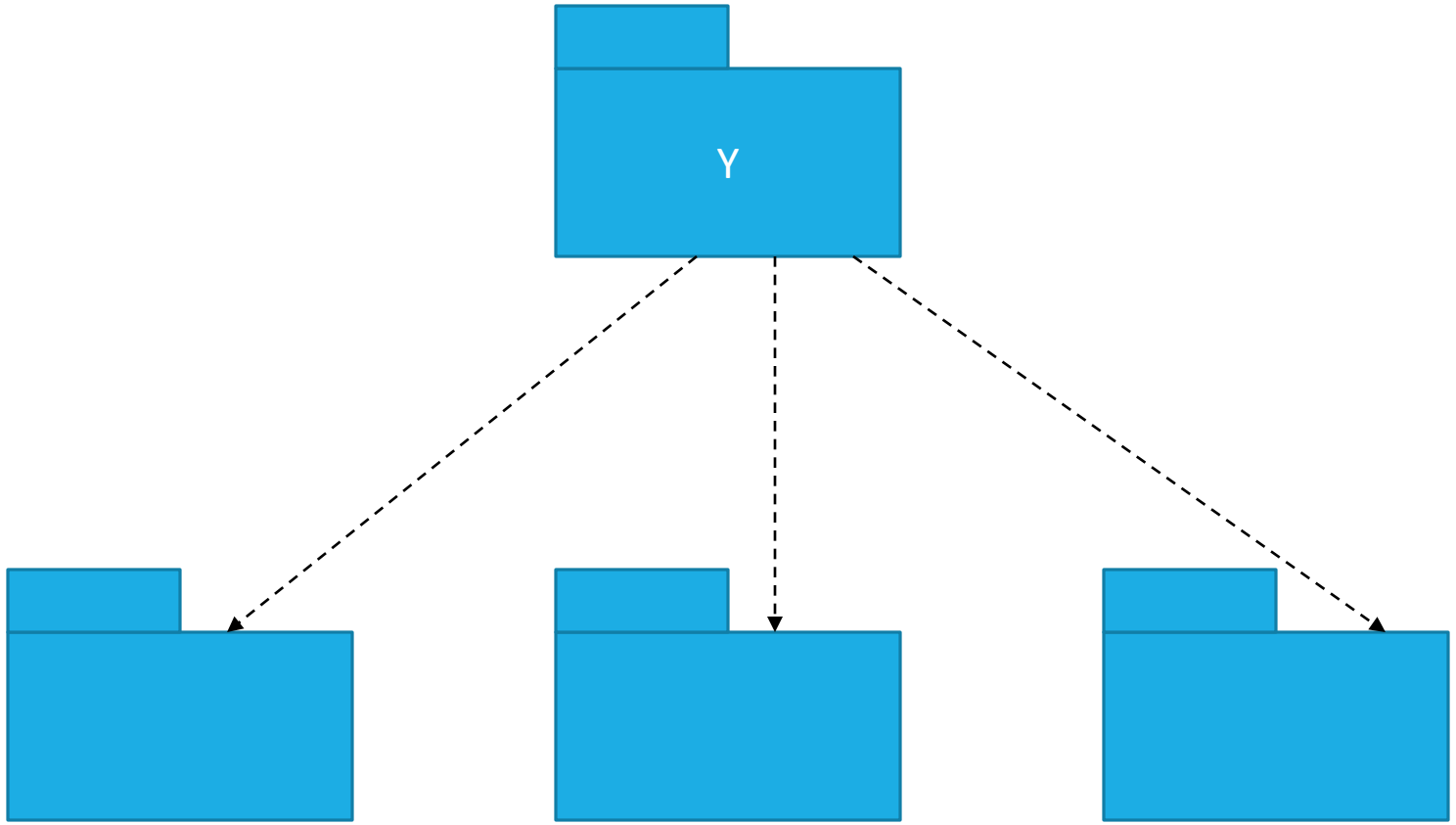
Stability : amount of work required to make a change

The Stable Dependency Principle (SDP)



X: A Stable Package

The Stable Dependency Principle (SDP)



Y: An Unstable Package

The Stable Dependency Principle (SDP)

Stability Metrics

- Afferent Couplings (C_a): The number of classes outside this package that depends on classes within this package
- Efferent Couplings (C_e): The number of classes inside this package that depend on classes outside this package
- Instability (I) : $$I = \frac{C_e}{C_a + C_e}$$

Range = [0,1]

I = 0: Maximum Stability

I = 1: Maximum Instable Package

The Stable Abstraction Principle (SAP)

The package should be as abstract as it is stable

This principle sets up the relationship between stability and abstractness

Stable package should also be abstract so that its stability does not prevent it from being extended

Instable package should also be concrete since its instability allows the concrete code within it to be easily changed

The Stable Abstraction Principle (SAP)

Abstraction Metrics

- N_c – The number of classes in the package
- N_a – The number of abstract classes in the package
- Abstractness (A) :

$$A = \frac{N_a}{N_c}$$

Range = [0,1]

A = 0: Package has no Abstract classes

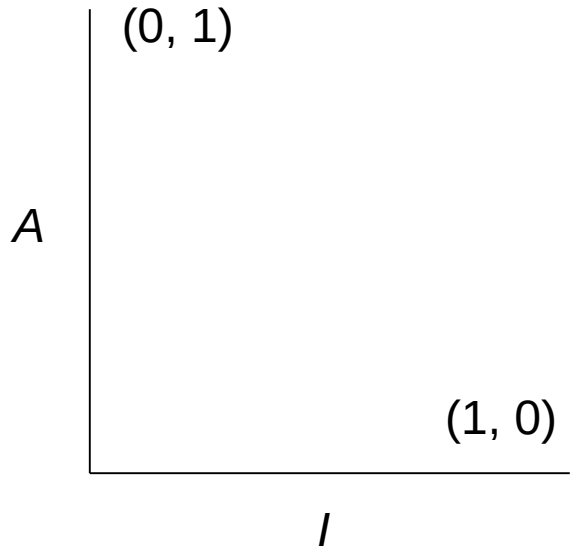
A = 1: Package contains nothing but the abstract classes

The Main Sequence

A – I Graph

Maximally stable and abstract (0, 1)

Maximally instable and concrete (1, 0)



The Main Sequence

(0, 0) : Highly stable and concrete package

It cannot be extended because it is not abstract

It is very difficult to change because of its stability

Example 1 : Database Schemas

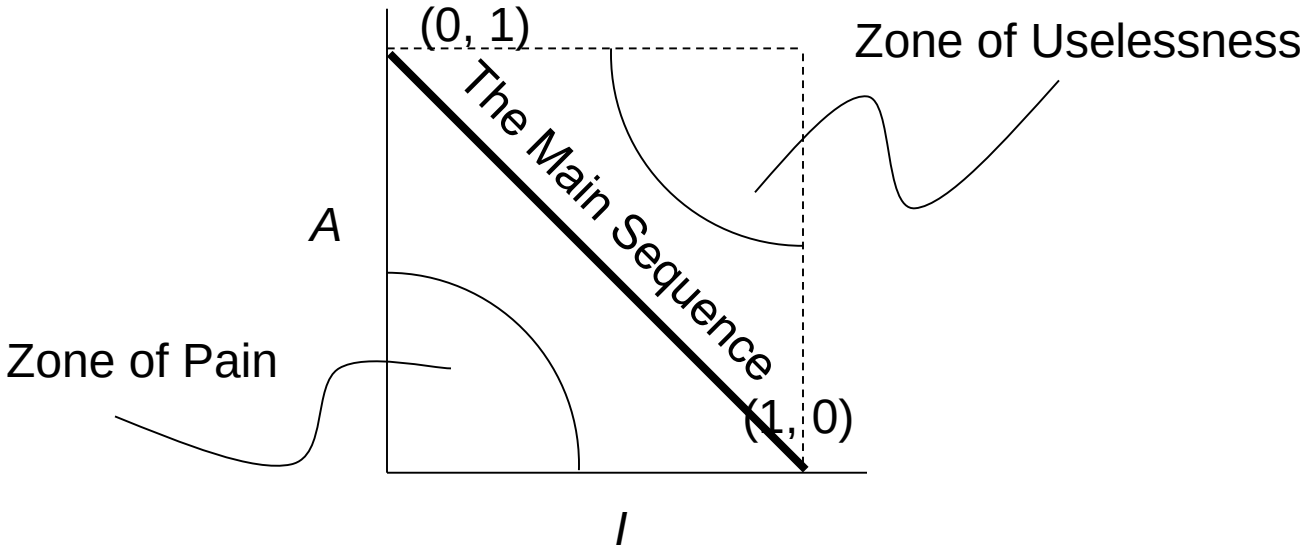
Example 2 : Concrete Utility Library

Zone of Pain

The Main Sequence

(1, 1) : Maximally abstract and no dependents

Zone of Uselessness



Distance from the Main Sequence

Distance Formula

Range [0, ~ 0.707]

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

Normalized Distance D’

Range [0, 1]

$$D' = |A + I - 1|$$