Software Engineering - Lecture 7

Software implementation

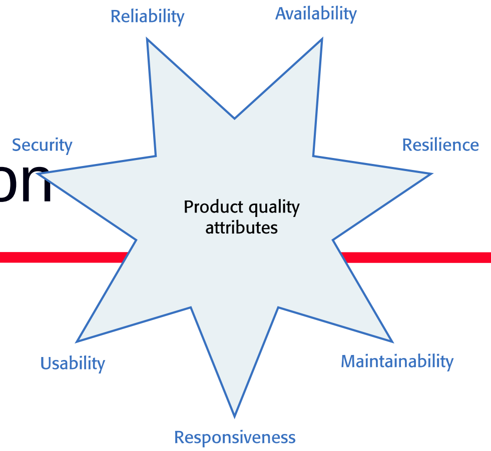
Implementation

- Implementation = the act of *transforming a design* in a valid *program* written in some programming language, together with *all its supporting activities*.

- Implementation – more just *writing code*, involves *code testing* and *debugging*, as well as *compiling* and *building* a complete executable product.

Figure 8.1 Software product quality attributes
Ian Sommerville – Engineering Software Products

Characteristics of a good implementation



- ***Readability*** – the code can be easily read and understood by other programmers.
- ***Mentenability*** – the code can be easily modified and maintained.
- ***Performance*** – the code should perform as fast as possible.
- ***Traceability*** – all code elements should correspond to a design element. Code can be traced back to design (and design to requirements).
- ***Correctness*** – the implementation should do what it is intended to do (as defined in the requirements and detailed design).
- ***Completness*** – all the system requirements are met.

Concerns and trade-offs

Concerns - examples

- Programmer – *correctness, performance.*
- Software engineer implied in projects with multiple releases – *correctness, mentenability*.

Trade-offs - examples

- Clarity helps mentenability and both help correctness.
- Performance optimizations reduce clarity and mentenability (sometimes even performance).

Topics covered

- **Programming for mentenability**
- Programming for reliability
- Performance optimization
- Testing vs. debugging

Programming style and coding guide

Keeping the consistency of the used notation.

Highlighting code ***semantics***.

- Coding guide (specific to software company) expresses :
  - Naming
  - Identation
  - Comment styles
  - Capitalization
  - Banning language features and practices that have proved error-prone (ex. pointers, multiple inheritance, ignoring warnings, deprecated language characteristics).

Solutions – in most of the specific software tools.

Programming style and coding guide

Recommendations

- **Naming**
  - Long names conveing semantics for global entities
  - Consistent semantics both for long names and abreviations
  - Consistent with external standards

- **Separating words and capitalization**
  - Consistent use of the standard conventions of the programming language.

Programming style and coding guide

Recommendations

- **Identation and spacing**
  - Identation according to the programming structures
  - Consistency in using a predefined identation style

- **Function / method size**
  - Big size ⇒ high probability to contain errors
  - Big size ⇒ difficult to read and understand
  - Recommendation : max. 50 lines of code

Programming style and coding guide

Recommendations

- **File naming**
  - Create and folow a file naming convention that allows
    - identify the files which must be generated for each module
    - localize a file in a given module

  Ex. Use prefixes and sufixes.

  Obs. The same prefix can be also used in error messages in order to localize the module that generates the error.

OR

  - Create a document that specifies the mapping between module and file.

- **Dangerous programming features** (language specific)
  - Implicit : Banning dangerous language features (ex. GOTO, multiple inheritance)
  - Authorize their usage in well motivated situations.

Comments

Extremely important

BUT

Possible problems :

- the programe becomes less readable
- wrong comments
- comments not synchronized with code changes

Comments

Advantages:

- Clarifying the code
- Relating the code with other sources

Disadvantages:

- Introduce a certain level of code duplication
- Need supplementary effort for creation and updating
- Sometimes used trying to justify a too complex code

Tendency –”self-documented code” – written so well that it must not be documented with comments.

Comments - still needed at least to describe the *programmer intention*.

Comments

Categories of comments (1):

```
Ex.
// increment by one
++i;
```

- Repeat code – to avoid
- Explanation of the code – if the code is so complex that it requires an explanation, consider to rewrite it.
- **Marker** – used to indicate incomplete elements, improving oportunities, other similar functions.
  - Use a consistent notation
  - Eliminate at the right moment
  - Instead of using markers to keep track of changes and who made them, it is recommended to use a version management software.

Comments

Categories of comments (2):

- **Summary of the code** – very helpful in understanding the code, but must be synchronized with code changes; can be replaced with a well chosen name for the implemented function.

- **Description of the code intent** – *the most useful comments*; override the code: if the code does not fulfill its intent, then the code is wrong.

- **External references** – link the code to external entities (books, other programs, ...) or to external prerequisites and corequisites for the code (ex. existence of initialized data in the database tables).

Programming style and comments

Final recommendation:

- Appropriate names (well chosen, significant and structured)
- Good coding practices
- Comments to indicate external references and programmer intent. Sometimes a summary may be necessary for complex codes that can not be abstractized.

Formative evaluation

1. Enumerate the categories of comments in the order of their importance, starting with the most important ones.

https://forms.gle/6jviTxEPTNJ88MSW7

Topics covered

- Programming for mentenability
- **Programming for reliability**
- Performance optimization
- Testing vs. debugging

Programming for reliability

- Simple techniques for reliability improvement that can be applied in any software company.
  - *Fault avoidance* -- Program in such a way that introducing faults into the program is avoided.
  - *Input validation* -- Define the expected format for user inputs and validate that all inputs conform to that format.
  - *Failure management* -- Implement the software so that program failures have minimal impact on product users.

•Fault avoidance
•Input validation
•Failure management

Fault avoidance

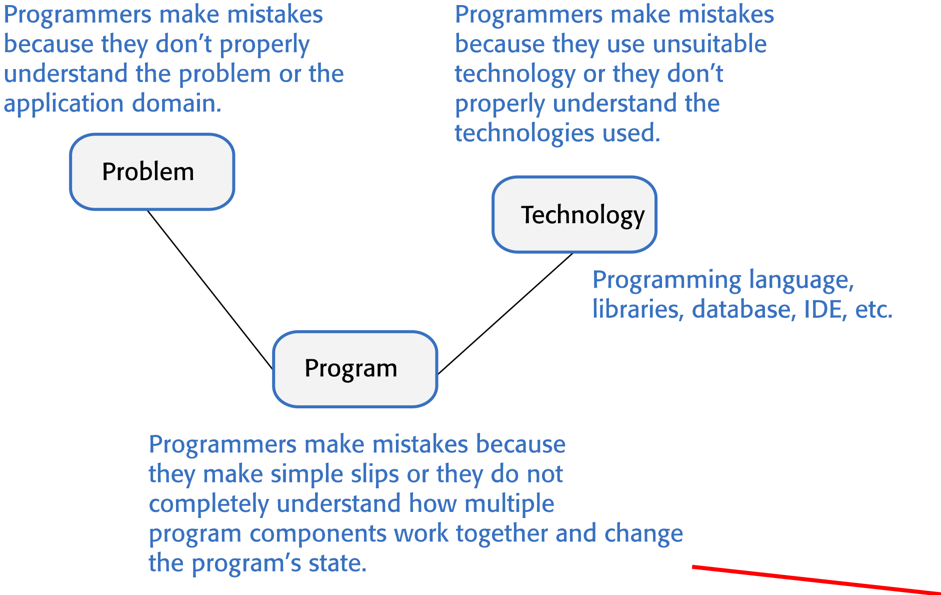
Fault avoidance -- Program in such a way that introducing faults into the program is avoided.

- Reducing program complexity
- Using design patterns
- Refactoring

- Fault avoidance
- Input validation
- Failure management

Causes of program errors

Figure 8.2 Underlying causes of program errors
Ian Sommerville – Engineering Software Products



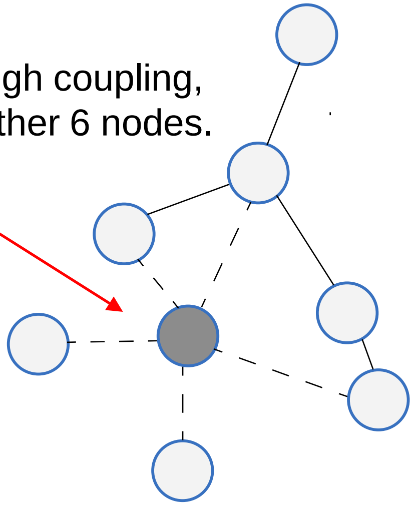
Programmers make mistakes because they don't properly understand the problem or the application domain.

Programmers make mistakes because they use unsuitable technology or they don't properly understand the technologies used.

Programming language, libraries, database, IDE, etc.

Programmers make mistakes because they make simple slips or they do not completely understand how multiple program components work together and change the program's state.

Node with relatively high coupling, in relationships with other 6 nodes.



The shaded node interacts, in some ways, with the linked nodes shown by the dotted line

Figure 8.3 Software complexity
Ian Sommerville – Engineering Software Products

Program complexity

- Complexity is related to the *number of relationships* between elements in a program and the *type and nature of these relationships*
- The number of relationships between entities is called the *coupling*. The higher the coupling, the more complex the system.
- A *static* relationship is one that is stable and does not depend on program execution.
  - Whether or not one component is part of another component is a static relationship.
- *Dynamic* relationships, which change over time, are *more complex* than static relationships.
  - An example of a dynamic relationship is the ‘calls’ relationship between functions.

Types of complexity

- ***Reading complexity***
  This reflects how hard it is to read and understand the program.
- ***Structural complexity***
  This reflects the number and types of relationship between the structures (classes, objects, methods or functions) in the program.
- ***Data complexity***
  This reflects the representations of data used and relationships between the data elements in the program.
- ***Decision complexity***
  This reflects the complexity of the decisions in the program

Complexity reduction guidelines

- ***Structural complexity***
  - Functions should do one thing and one thing only
  - Functions should never have side-effects
  - Every class should have a single responsibility
  - Minimize the depth of inheritance hierarchies
  - Avoid multiple inheritance
  - Avoid threads (parallelism) unless absolutely necessary
- ***Data complexity***
  - Define interfaces for all abstractions
  - Define abstract data types
  - Avoid using floating-point numbers
  - Never use data aliases
- ***Decision complexity***
  - Avoid deeply nested conditional statements
  - Avoid complex conditional expressions

Complexity reduction guidelines

Every class should have a single responsibility

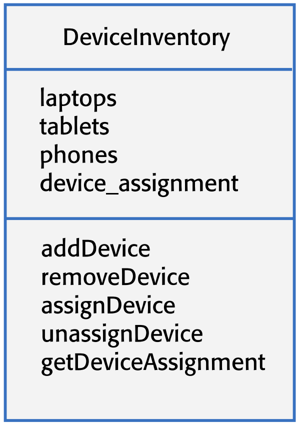
- Fault avoidance
  - Program complexity
  - Design patterns
  - Refactoring
- Input validation
- Failure management

A single reason to change a class.

- smaller and more cohesive classes
- less complex and easier to understand and change.

Method

- Gather together the things that change for the same reasons.
- Separate those things that change for different reasons.

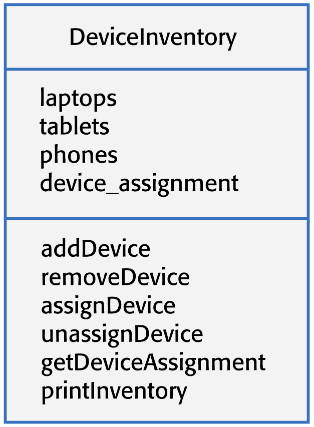


DeviceInventory

laptops
tablets
phones
device_assignment

addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment

(a)



DeviceInventory

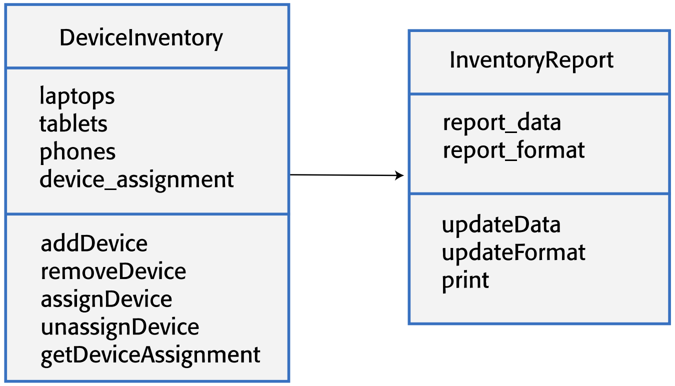
laptops
tablets
phones
device_assignment

addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment
printInventory

(b)

(b) Another data type (a report) is associated with the class ⇒ Additional ‘reason to change’ = change the format of the printed report.

(a) Reason to change = fundamental change in the inventory (ex. recording who is using their personal phone for business purposes.)



DeviceInventory

laptops
tablets
phones
device_assignment

addDevice
removeDevice
assignDevice
unassignDevice
getDeviceAssignment

InventoryReport

report_data
report_format

updateData
updateFormat
print

Solution : add a new class to represent the printed report.

Complexity reduction guidelines

Avoid deeply nested conditional statements

- Deeply nested conditional (if) statements = selection from a possible set of choices.

Example: Adjusted premiums based on the age and experience of drivers.

- function ‘agecheck’ used to calculate an age multiplier for insurance premiums.

Good practice : name constants rather than using absolute numbers.

```
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def agecheck (age, experience):        # Assigns a premium multiplier depending on the age and experience of the driver
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER *YOUNG_DRIVER_EXPERIENCE_MULTIPLIER
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

Complexity reduction guidelines

Avoid deeply nested conditional statements

- Better approach : use guards with multiple returns
- ***Guard*** = conditional expression placed in front of the code to be executed

Switch statement in some languages (ex. Java, C++,etc.)

In Python need to be simulated

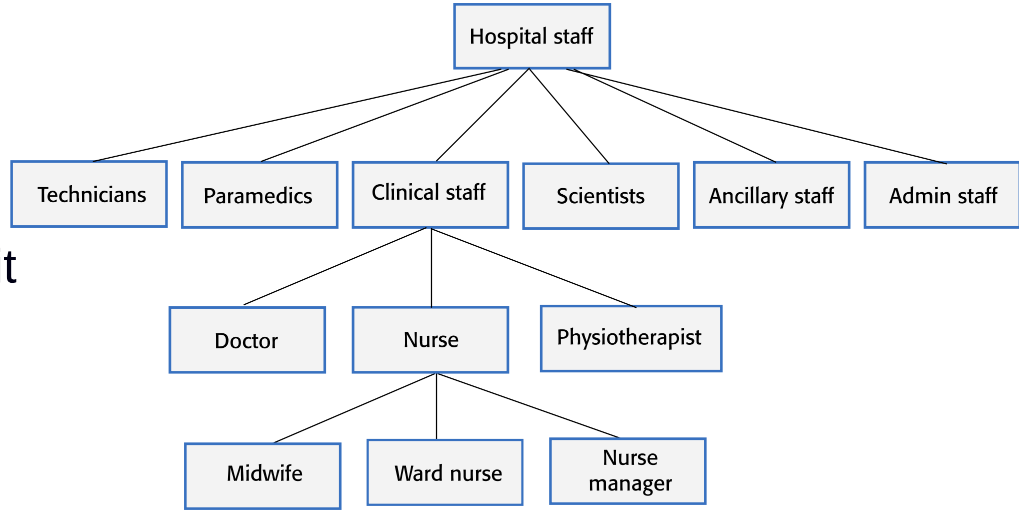
```
def agecheck_with_guards (age, experience):

        if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER * YOUNG_DRIVER_EXPERIENCE_MULTIPLIER
        if age <= YOUNG_DRIVER_AGE_LIMIT:
                return YOUNG_DRIVER_PREMIUM_MULTIPLIER
        if (age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE) and experience <= OLDER_DRIVER_EXPERIENCE:
                return OLDER_DRIVER_PREMIUM_MULTIPLIER
        if age > ELDERLY_DRIVER_AGE:
                return ELDERLY_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER
```

Complexity reduction guidelines

Avoid deep inheritance hierarchies

- Inheritance – attributes and methods of a class are inherited by sub-classes.
- Effective and efficient way of reusing code and of making changes that affect all subclasses.
- Increases the *structural complexity* of code as it increases the coupling of subclasses.



- Problem : to make changes to a class
  - analize ***all*** its super-classes to find the proper place to make the change.
  - analyze ***all*** related subclasses against unwanted consequences ⇒ high complexity ⇒ easy to make mistakes in the analysis and introduce faults into the program.

A solution : Remove lower level but introduce *guards* when programming ⇒ decision complexity.

⇓

Tradeoff *structural complexity* for *decision complexity.*

Design pattern

Def. ***Design pattern*** = general reusable solution to a commonly-occurring problem within a given context in software design.

Design patterns

- are object-oriented and describe the *essence* of the solutions.
- describe the *structure* of a problem solution but have *to be adapted* to suit the application and the used programming language.

Underlying programming principles :

Separation of concerns

- Each abstraction in the program (class, method, etc.) should address a separate concern and all aspects of that concern should be covered there.
- Identify the aspects of the application that vary and separate them from what stays the same.

Separate the ‘what’ from the ‘how’

- the ‘what’ = information that is required to use a service, available to service users.
- the ‘how’ = implementation of the service, hidden from the service users.
- Program to an interface (more generally, to a supertype), not to an implementation.

Common types of design patterns

- Creational patterns
  - concerned with class and object creation.
  - define abstract ways of instantiating and initializing objects and classes.

Examples : Factory (create objects with slightly different variants), Prototype (create clones)

- Structural patterns
  - Concerned with class and object composition.
  - Description of how classes and objects may be combined to create larger structures.

Examples : Adapter (match semantically-compatible interfaces), Façade (single interface to a group of classes each implementing part of the functionality accessed through interface)

- Behavioural patterns
  - Concerned with class and object communication.
  - How objects interact by exchanging messages, the activities in a process and how these are distributed amongst the participating objects.

Examples : Mediator (mediates communications), State (implements a state machine), Observer (publisher/subscriber metaphor)

Refactoring

The reality of programming : as changes and additions are made to existing code, its complexity inevitably increases.

Initial abstractions and operations become more and more complex because they are modified in unanticipated ways ⇒ The code becomes harder to understand and change.

Def. ***Refactoring*** = the activity of *code improving* in order to become easy to understand and modify, *without altering its externaly observable behaviour*.

- Reduces 'reading complexity' ⇒ the program becomes more readable and more understandable.
- Improves maintainability ⇒ easier to change, reduced chances of making mistakes when new features are introduced.
- Powerful technique to produce quality code.
- Most of the software tools (editors, IDEs) offer automated support for refactoring.

Ex. Eclipse, NetBeans, JDeveloper, Visual Studio (Visual Assist).

Code smells

Def. “code smell” – *simptom* in source code which indicates a *possible* problem.

Examples with refactoring solutions:

- *Large classes*
  The single responsibility principle may be violated. -- Break down large classes into easier-to-understand, smaller classes.
- *Long methods/functions*
  The function may do more than one thing. -- Split into smaller, more specific functions or methods.
- *Duplicated code*
  Changes have to be made everywhere the code is duplicated. -- Rewrite to create a single instance of the duplicated code that is used as required
- *Meaningless names*
  Sign of programmer haste; make the code harder to understand. -- Replace with meaningful names and check for other shortcuts that the programmer may have taken.
- *Unused code*
  Increases the reading complexity of the code. -- Delete it even if it has been commented out. If you find you need it later, you should be able to retrieve it from the code management system.

Refactoring

Refactoring examples:

- *Extract method* – transform a fragment of code in a method with an appropriate name and replace with a call to that method.
- *Extract class*
- *Substitute algorithm* – replace the body of a method with a more clear algorithm that returns the same result.
- *Move method* – move an algorithm from one class to another.
- *Rename method*
- *Rename field*
- *Rename variable*
- *Replace magic literal*
- *Replace nested conditional with guard clauses*
- *Remove dead code*

Martin Fowler – Refactoring : Improving the Design of Existing Code - catalog with useful refactorings.
www.refactoring.com
wiki.java.net/bin/view/People/SmellsToRefactoring

Steve McConnell – Code Complete, Second Edition, cap 24.

Complexity reduction by refactoring

Examples

- *Reading complexity*
  Rename variables, methods, classes, to make their purpose more obvious.

- *Structural complexity*
  Break long classes or methods into shorter units that are likely to be more cohesive than the original large one.

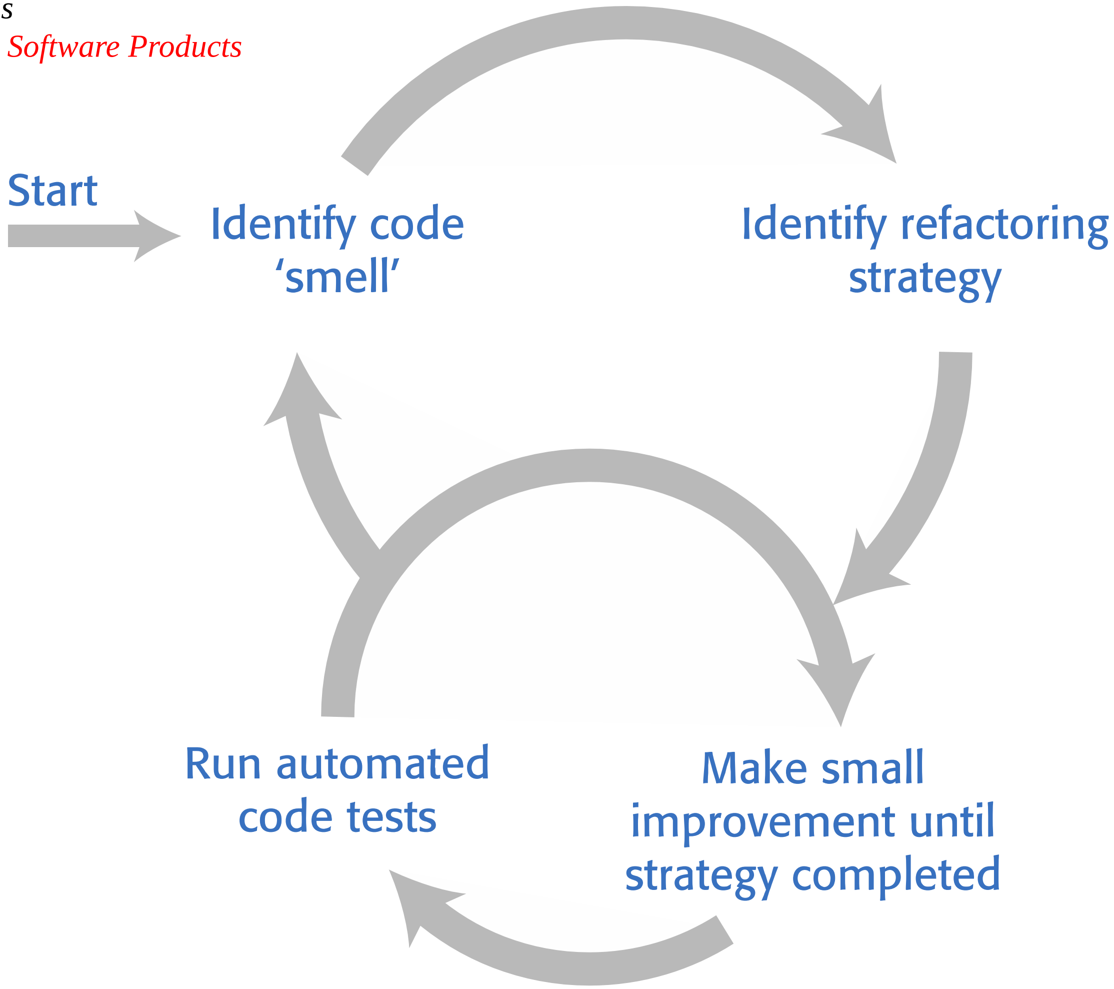
- *Data complexity*
  Change database schema or reduce its complexity.

- *Decision complexity*
  Replace a series of deeply nested `if-then-else` statements with guard clauses.

- **Fault avoidance**
  - Program complexity
  - Design patterns
  - **Refactoring**
- Input validation
- Failure management

Refactoring process

Figure 8.8 A refactoring process
Ian Sommerville – Engineering Software Products


Start
Identify code 'smell'
Identify refactoring strategy
Make small improvement until strategy completed
Run automated code tests

Input validation

- Input validation for:
  - correct format;
  - value within the range defined by input rules.
- Critical for security and reliability.
  - catches accidentally invalid inputs that could crash the program or pollute the database.

Good practice

- Define *rules for every type* of input field
- Include *code that applies* these rules to check the field’s validity.
  - If it does not conform to the rules, the input should be rejected.

Rules for name checking

Rules :

- Name length : between 2 and 40 characters.
- Characters : alphabetic or alphabetic with an accent, plus hyphen and apostrophe as separator characters.
- Start with a letter.

Consequences of applying these rules :

- eliminate very long strings that might lead to buffer overflow,
- eliminate embed SQL commands in a name field.

Methods of implementing input validation

- ***Built-in validation functions***
  Use input validator functions provided by the web development framework.
  (Example: Most frameworks include a validator function that will check that an email address is of the correct format.)

- ***Type coercion functions***
  Use type coercion functions that convert the input string into the desired type.
  (Example: int() in Python ⇒ If the input is not a sequence of digits, the conversion will fail).

- ***Explicit comparisons***
  Define a list of allowed values and possible abbreviations and check inputs against this list. (Example : if a month is expected, check this against a list of all months and recognized abbreviations.)

- ***Regular expressions***
  Use regular expressions to define a pattern that the input should match and reject inputs that do not match that pattern.

Regular expressions

- Regular expressions (REs) = a way of defining patterns.
- Used to define search with a pattern so that all items matching that pattern are returned.

Obs. Most programming languages have a regular expression library.

Example: Unix command to list all the JPEG files in a directory, using Python library:

```
ls | grep ..*\.jpg$
```

(`grep =` "Global search for Regular Expression and Print matching lines“)

Single dot means ‘match any character’

* means zero or more repetitions of the previous character

..* means ‘one or more characters’

File suffix is .jpg

$ character means that it must occur at the end of a line.

Regular expressions

Example : A name checking function

Procedure :

- Define the pattern that matches all valid strings
- Check the input against this pattern
- Reject inputs that do not match

```
def namecheck (s):
# checks that a name only includes alphabetic characters, -, or single quote
# names must be between 2 and 40 characters long
# quoted strings and -- are disallowed

        namex = r"^[a-zA-Z][a-zA-Z-']{1,39}$"
        if re.match (namex, s):
                if re.search ("'.*'", s) or re.search ("--", s):
                        return False
                else:
                        return True
        else:
                return False
```

Number checking

- Check numeric inputs to be:
  - not too large,
  - not to small,
  - sensible values for the type of input.

Example: expected input for person height in meters ⇒ value between 0.6m (a very small adult) and 2.6m (a very tall adult).

- Reasons for number checking :
  - Too large or too small to be represented ⇒ unpredictable results and numeric overflow or underflow exceptions.
  - Number checking ⇒ properly handle of the exceptions ⇒ program does not crash.
  - Database used by several programs that make assumptions about the numeric values stored ⇒ the numbers must be as expected, else unpredictable results.

Input range checks

- Checking that inputs represent sensible values.

- will protect the system from accidental input errors
- may stop intruders, who have gained access using a legitimate user’s credentials, from seriously damaging their account.

Example: Input of the reading from an electricity meter is checked to be:

- (a) equal to or larger than the previous meter reading

AND

- (b) consistent with the user’s normal consumption.

Failure management

Program failures causes :

- Faults *into a program* caused by *software complexity* (irrespective of how much effort is put into fault avoidance).
- Failure of an *external service or component* that the software depends on.

Solution to increase reliability:

1. Plan for failure
2. Make provisions in the software for that failure to be as graceful as possible.

Failure categories

- Data failures
  - Incorrect outputs of computations (ex. compute age by adding a date with a number)

  May be used in other computations that generate more incorrect information, that may be stored in a database ⇒ polluted database

  - Reported by users which notice data anomalies
- Program exceptions
  - Exceptions not handled ⇒ control transferred to the run-time system which halts execution and the program crashes. (Ex. request to open a file that does not exist raises an IOException.)
- Timing failures
  - Interacting components fail to respond on time
  - The responses of concurrently-executing components are not properly synchronized.

Failure effect minimization

Requirements to fulfil in the event of a failure, in order to minimize its effect:

- Persistent data (i.e. data in a database or files) should not be lost or corrupted;
- The user should be able to recover the work that they've done before the failure occurred;
- Software should not hang or crash;
- Provide 'fail secure' so that confidential data is not left in a state where an attacker can gain access to it.

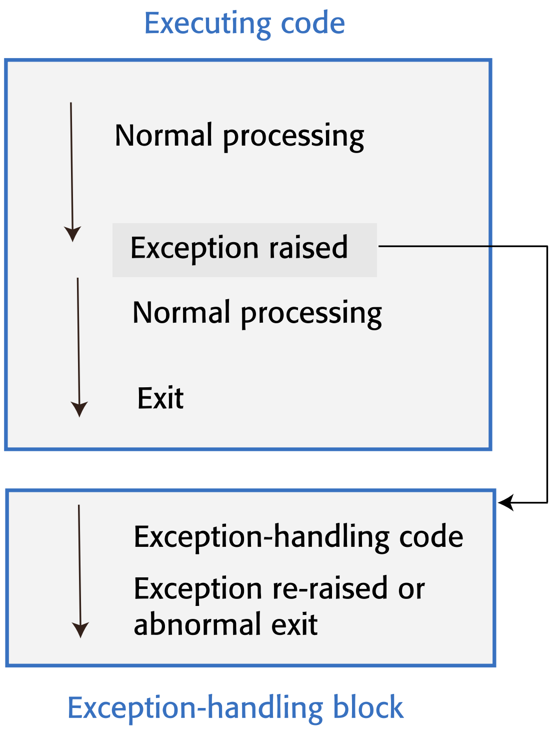
Solutions for reducing failure effect:

- Secure failure
- Activity logging
- Periodically auto-saving data
- Verification of external services responses

Reducing failure effect
Secure failure

- Fault avoidance
  - Program complexity
  - Design patterns
  - Refactoring
- Input validation
- **Failure management**

- Exception = event that disrupts the normal flow of processing in a program.
- When an exception occurs, control is automatically transferred to exception management code.
- Most modern programming languages include a mechanism for exception handling.
  - Python : **try-except**
  - Java: **try-catch.**


Executing code

Normal processing

Exception raised

Normal processing

Exit

Exception-handling code

Exception re-raised or abnormal exit

Exception-handling block

- Exception handler
  - Tidy up before the system shuts down: closes the files, releases the resources, protects the sensitive data, etc.
  - Sometimes is possible to define an exception handler that recovers from a problem, but this involves rolling back execution to a known correct state.

Secure failure

Example

- •Fault avoidance
  - Program complexity
  - Design patterns
  - Refactoring
- •Input validation
- •**Failure management**

```
def do_normal_processing (wf, ef):

# Normal processing here.
# Code below simulates exceptions
# rather than normal processing

  try:
    wf.write ('line 1\n')
    ef.write ('encrypted line 1')
    wf.write ('line 2\n')
    wf.close()
    print (‘Force exception')
    tst = open (test_root+'nofile')

  except IOError as e:
    print ('I/O exception ')
    raise e
```

Ensures that no confidential information is exposed in the event of a system failure.

```
def main ():
  wf = open (test_root+’work.txt', 'w')
  ef = open(test_root+'encrypted.txt', 'w')
  try:
    do_normal_processing (wf, ef)

  except Exception
# If the modification time of the unencrypted work file (wf)
# is later than the modification time of the encrypted file (ef)
# then encrypt and write the workfile
    print ('Secure shutdown')
    wf_modtime = os.path.getmtime(test_root+'work.txt')
    ef_modtime = os.path.getmtime(test_root+'encrypted.txt')
    if wf_modtime > ef_modtime:
      encrypt_workfile (wf, ef)
    else:
      print ('Workfile modified before encrypted')
    wf.close()
    ef.close()
    os.remove (test_root+'workfile.txt')
    print ('Secure shutdown complete')
```

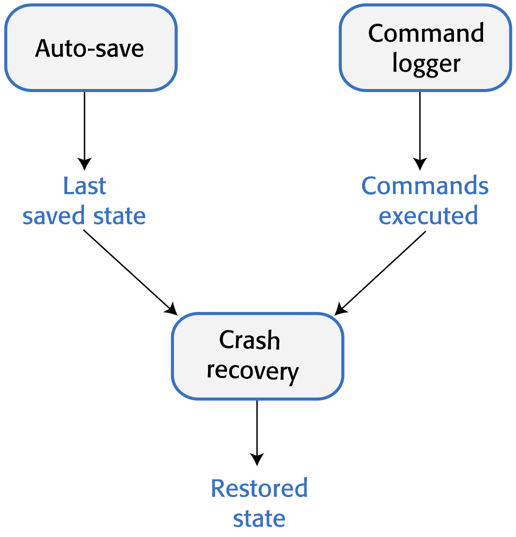
Reducing failure effect
Activity logging
Auto-saving

- Fault avoidance
  - Program complexity
  - Design patterns
  - Refactoring
- Input validation
- **Failure management**

- Activity logging
  - A list of user actions since the last time the data was saved to persistent store.
  - A way to replay the actions in the list against saved data.
- Auto-save
  - Automatically save the user’s data at set intervals (ex. every 5 minutes).

⇓

  - In the event of a failure, the saved data can be restored with the loss of only a small amount of work.
  - Usually, are saved only the changes that have been made since the last explicit save.



Reducing failure effect

Verification of external services responses

- No control exists over external services ⇒ the only information on service failure is whatever is provided in the service’s API.

- Services may be written in different programming languages ⇒ errors are usually returned as a numeric code, not as exception types.

- When calling an external service :
    - Initiate a mechanism (based on timeout) to detect if the service does not respond.

  If the service responds :
    - Check the return code to see if it indicates a successful operation or an error.
    - Check the validity of the result of the service, to make sure that the external service has carried out its computation correctly.
    - A possible error message will be translated in an understandable form.

- Fault avoidance
    - Program complexity
    - Design patterns
    - Refactoring
- Input validation
- **Failure management**

Example : Using assertions to check results from an external service

Assume that the function `check_credit_rating` calls an external service to get a person's credit rating. It takes a *name, postcode (zip code*) and *date of birth* as parameters and returns a sequence with the database information (*name, postcode, date of birth*) plus a credit score between 0 and 600. The final element in the sequence is an error_code which may be 0 (successful completion), 1 or 2.

```
def credit_checker (name, postcode, dob):
   NAME = 0
   POSTCODE = 1
   DOB = 2
   RATING = 3
   RETURNCODE = 4
   REQUEST_FAILURE = True
   ASSERTION_ERROR = False
   cr = ['', '', '', -1, 2]

   #  Check credit rating simulates call to external service
   cr = check_credit_rating (name, postcode, dob)
   # Code to verify the result obtained from the external service
   try:
      assert cr [NAME] == name and cr [POSTCODE] == postcode and cr [DOB] == dob \
         and (cr [RATING] >= 0 and cr [RATING] <= 600) and \
         (cr[RETURNCODE] >= 0 and cr[RETURNCODE] <= 2)
      if cr[RETURNCODE] == 0:
         do_normal_processing (cr)
      else:
         do_exception_processing (cr, name, postcode, dob, REQUEST_FAILURE) #Response with error
                                                                             indicated by the service
   except AssertionError: #Incorrect computation
         do_exception_processing (cr, name, postcode, dob, ASSERTION_ERROR)
```

Formative evaluation

1. Select the methods to avoid introducing programming faults.

2. What is the meaning of secure failure and how is it realized ?

3. What must to be done to reduce the effects of failures generated by an external service used by the application ?

https://forms.gle/tidVEu7UQTNgmvYi8

Topics covered

- Programming for mentenability
- Programming for reliability
- **Performance optimization**
- Testing vs. debugging

Performance optimization

- *Corectness* has maximum priority
- *Performance*
  - Critical only in real time systems
  - Implies (generally) a tradeoff with the maintenability and the clarity of the program.

Recommendation:

- Realize a *correct* and *maintenable* program
- Optimize the performance if necessary
- Analize the execution profile (using a profiler tool) to identify critical code areas for the performance
- Optimize those modules which have considerable impact on the performance.

Performance optimization

Other solutions:

- Clarity, simplity, modularity of the program help improving the performance
- Use of optimizing compilers
- Reuse of high quality code (standard implementations for data structures and algorithms)

Topics covered

- Programming for mentenability
- Programming for reliability
- Performance optimization
- **Testing vs. debugging**

Testing vs. Debugging

Verification testing and debugging are *distinct processes*.

- Verification testing is concerned with ***establishing the existence*** of defects in the program.
- Debugging is concerned with ***localizing and repairing*** these errors.
- Debugging implies ***formulating hypothesis*** about the behaviour of the program, followed by testing these hypothesis in order to find the error in the system.

Debugging

Def. Debugging – the activity of localizing and eliminating the errors in code discovered during testing activity.

Debugging – iterative process :

- Create a hypothesis about what causes the error
- Write test cases to prove or disprove the hypothesis
- Change code to try to fix the problem

Debugging

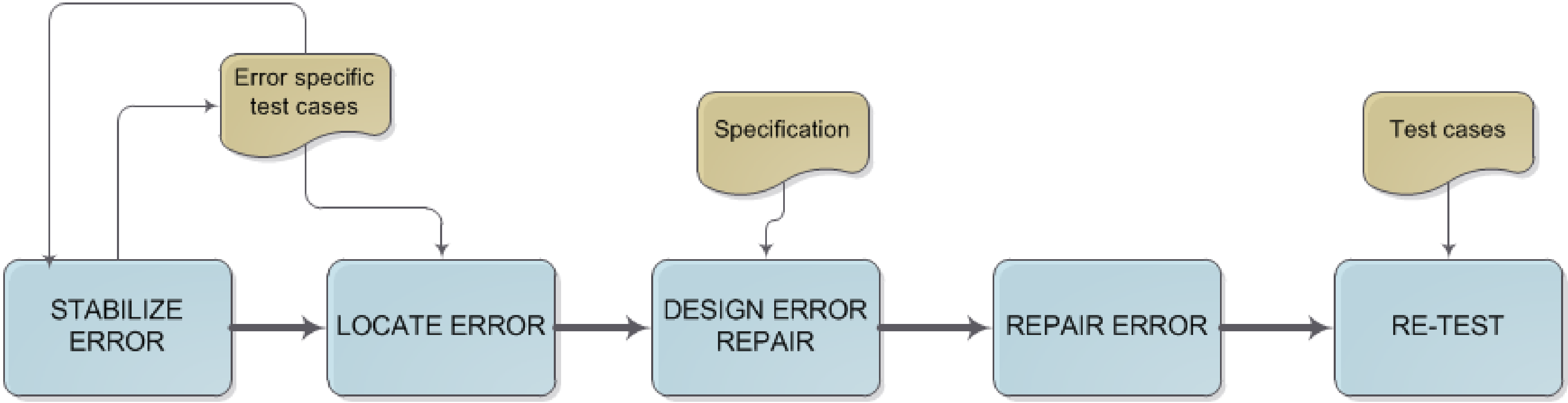
Phases:

- ***Stabilization*** – *reproducing the error* on a particular configuration (in many cases the developer's machine) and finding out the *conditions* that led to the error by constructing a *minimal* test case.

Obs. It is not necessary to inspect the code but to identify which *input conditions* combined with which *program states* produce the error.

- ***Localization*** – finding the sections of the code that led to the error
- ***Correction*** – changing the code to fix the errors
- ***Verification*** – making sure the error is fixed and no other errors were introduced with the changes in the code.

Debugging process



Debugging

Stabilization

Outputs: series of test cases that produce the error and possibly some cases that perform correct.

Stabilization implies minimization of the conditions that produce the error ⇒ after writing a test case that reproduce the error, try to write a simpler one that also fails.

Errors hard to stabilize – produced by:

- Uninitialized variabiles
- Dangling pointers
- Interaction of several threads

Occur at random on the same data inputs, depending on the state of the program.

Debugging

Localization

Rules of thumb to find errors:

- Inspect the design
- Inspect the code
- Routines with more than one error tend to have even more errors
- Newly created code
- Heuristics specific to the program

Debugging – useful tools

- Source code comparators
- Lint-like tools – detect error-prone code, based on static analysis
- Interactive debuggers : intrerrupt the program in predefined points (breakpoints) and examine the values of the variables; let you control the control sequence.
- Special constructed libraries that reimplement standard libraries but with extra safeguard, to detect and prevent errors.
- DBC (design by contract) facilities included in languages (ex. Languages on .NET platform) or offered by separate libraries (ex. Java-on-contracts for Java) : ADT, preconditions, postconditions, invariants, etc.
- Defensive programming facilities : assertions = predicates placed in the code.

Formative evaluation

1. Select the activities included in code debugging.

- establishing the existence of errors
- error correction
- localization of the error
- creation of acceptance tests
- verification
- code refactoring

https://forms.gle/Ayz16UAEbgRWRx1n8

Key points 1

- The most important quality attributes for most software products are reliability, security, availability, usability, responsiveness and maintainability.
- Introducing faults into program is avoided using programming practices that reduce the probability of making mistakes.
- Minimizing complexity in programs decreases the chances of programmer errors and makes the program more simple to change.
- Design patterns are tried and tested solutions to commonly occurring problems. Using patterns is an effective way of reducing program complexity.
- Refactoring is the process of reducing the complexity of an existing program without changing its externaly visible behaviour.
- Input validation involves checking all user inputs to ensure that they are in the format that is expected by the program. Input validation helps avoid the introduction of malicious code into the system and traps user errors that can pollute the database.

Key points 2

- Regular expressions are a way of defining patterns that can match a range of possible input strings. Regular expression matching is a compact and fast way of checking that an input string conforms to the rules you have defined.

- Numbers will be checked to have sensible values depending on the type of input expected. Number sequences should also be checked for feasibility.

- Assume that the program may fail and manage these failures so that they have minimal impact on the user.

- Exception management is supported in most modern programming languages. Control is transferred to a programmer defined exception handler to deal with the failure when a program exception is detected.

- As the program executes, log user updates and maintain user data snapshots These are used to recover, in the event of a failure, the work that the user has done. Also include ways of recognizing and recovering from external service failures.

Key points 3

- First, a *correct* and *maintenable* program will be realized, and, if necessary, its performance will be optimized..

- Verification testing and debugging are *distinct processes*. Verification testing is concerned with *establishing the existence* of defects in the program while debugging is concerned with *localizing and repairing* these errors.