2017-1 Software Architecture And Engineering

51704 characters in 7972 words on 1416 lines

Florian Moser

February 20, 2018

1 Software Engineering

1.1 A collection of techniques, methodologies, and tools that help with the production of

a high quality software system
with a given budget
before a given deadline
while change occurs

constraints of good software
Scalability
Repairability
Portability
Reusability
Understandability
Maintainability
Security
Usability
Reliability
Robustness
Performance
Correctness
Interoperability
Evolvability
Verifiability
Backwards Compatability

1.2 Software Design

informal Modeling
abstract models to simplify understanding (UML)

formal Modeling
formally write down the model; has tool support (alloy)

Design principles
how to fit reused class into class hiearchy?

Architectural & design patterns
general, reusable solutions to common design problems

1.3 Testing

function testing
focuses on input / output behaviour (given functionality; how to structure input to find all variants?; needs only method signature)

structural testing
uses design knowlegde about algorythms to figure out corner cases

atomatic test case generation
generate test cases that execute a given path throught the program

dynamic program analysis
focuses on subset of program behaviours; and proves their correctness (under approximation)

static program analysis
capture all possible program behaviours in a mathematical model; and prove properties (over approximation)

1.4 Static Analysis

Mathematical foundations
Abstract interpretations
Practical applications

1.5 Software development

requirements elicitation
what the customer really wants
requirements engeneering (find out & write down what the customer wants)
requirements validation (crossreading)
requirements elicitation (create scenarios, use cases and write formal specifications)

design
how to display it
system design (use linked list or array list)
detailed design (choose behavour in corner cases, like if key not found in dictionary: exception or return null?)

implementation
implement it

validation
check if it fits the requirements

1.6 why projects fail

lack of user input (13%)
incomplete requirements (12%)
Changing requirements (11%)
Unrealistic expectations (10%)

2 REQUIREMENTS ELICITATION

2.1 requirements engeneering

describe user's view
identify what not how

part of requirements
functionality
user interaction
error handling
environemental conditions

NOT part of
system structure
implementation technology
system design
development methodology

2.2 functional requirements

functionality
what is the software supposed to do
relationship input to output
response to abnormal situations
exact sequence of operations
validity checks on the inputs
effect of parameters

external interfaces
interaction with people, hardware, other software
detailed descriptions of all inputs & outputs (description of purpose, source of input / destination of output, valid range, accuracy, tolerance, units of measure, relationships to other inputs/outputs, screen & window formats, data & command formats)

2.3 non-functional requirements

2.3.1 performance

speed, availability, response time, recovery time

static nummerical requirements
number of installations, simultanious users, amount of information handled

dynamic numerical requirements
number of transactions processed in timeframe (ex: 95% in under 1s)

2.3.2 attributes (quality)

potability, correctness, maintainablity, security

2.3.3 design constraints

operating environement

standart compliance
report format, audit tracking

implementation requirements
tools, programming languages, technology & methodology → fight for it!

operations requirements
administration & management of the system

legal requirements
licensing, regulation, certification

2.4 quality criteria of requirements

correctness
requirements represents the clients view

completeness
all possible scenarios are described, including exceptional behaviour

consistency
requirements do not contradict each other

clarity
reuqirements can only be interpreted in one way

realism
reuqirements can be implemented & delivered

verifiability
requirements can be verified (tests can be written to prove this)

traceablity
each feature can be traced to a set of functional requirements

2.5 general example

use time units / specific units to prove your point (not "fast", "in 2 seconds")

2.6 requirements validation

the sooner an error is found, the cheaper
occurres after requirements engeneering

reviews
by developers & clients

prototyping
throw-away prototypes (user interface or fully functional) to show functionality, study feasability, give clients an impression

2.7 requirements elicitation

indentify the following & write formal specification
is understood by customers & users

2.7.1 information sources

enduser, client, documentation, observation of tasks

2.7.2 actors

represent roles (kind of user, external system, physical environement)

to ask
who supported, who executes what, which environement

2.7.3 scenarios

document behaviour from the user's view
describes common case
focus on understandability
instance of an use case

how to indentify
what are the tasks needed, what information is accessed by the user, what input needs the system, which events needs to be reported

2.7.4 use cases

describes all edge cases
focus on completeness
list of steps describing interaction between actor & system to archieve a goal

use case contains
unique name (edit entry)
initiating & participating actors (admin)
flow of events (steps)
entry conditions (at least one entry must exists)
exit conditions (the entry has been updated)
exceptions (system faillure)
special requirements (admin needs keyboard)

2.7.5 nonfunctional requirements

definied together with functional because they have dependencies (help function for better usability)
typically contains conflicts (speed → C, maintainability → C#)

3 DESIGN

3.1 mastering complexity

decomposition system
partition overall development effort
support independet testing & analysis
decouple parts of a system so changes to one part do not affect other parts
permit system to be understood as a composition of mind-sized chunks to be understood one at a time
enable reuse of components

3.2 System design

determine software architecture as composition of subsystems

components
computation units with specified interface (databases, layers)

connectors
ineractions between components (method calls, events, pipes)

3.3 Detailed design

choose among different options
data structures, algorythms, subclass hierarchies

things to choose
NULL permitted as value? thread safety? available methods?

concepts
copy-on-write, destructive updates, reference counting, lazy initialization, valid entries, optimizations change behaviours?, shared elements

4 MODELING

4.1 Design documentation

document the design decisions made (with NULL values, lazy-initialization, etc)

design decisions
determine how code should be written
made in initial development, inheritance, writing client code, during maintenance

must be communicated to different developers

source code not suffient, as it only contains the obvious information → developers require difficult infos to extract from code to be documented

document
result values of method (and when they occurr)
side effects of methods
consistency conditions for data strucutures (null values etc)
how data structures evolve over time (arraylist → when is array resized?)
whether objects are shared
which details are essential, which are incidential (functionality vs performance optimization)

4.2 document for clients

how to use the code
document the interface

4.2.1 how to call correctly constructors & methods correctly

any precondition to the state of the object?
allowed values?

4.2.2 what is returned by methods

how are failures dealt with

what are valid responses

4.2.3 how method calls affect state

heap effects (effects on general state, state of a passed objects)
other effect as thrown exceptions
runtime of method (linear or quadratic)

4.2.4 also document

public fields, supertypes

4.2.5 interface documentation

global properties which are preserved
global requirements by all methods

consistency
client visible invariants (list item order)

evolution
property of sequences of states (immutable structure has always same content)

abbreviations
requirements & guaranteed for all methods (thread safety)

4.3 document for implementors

how does the code work
document the implementation
focus on WHAT properties are not HOW they are archieved

similar to interface
more details (include effects on fields), includes hidden methods

data structure more prominent
properties of fields; internal sharing (if \textdollar shared is true then it is shared); invariants (list is not changed)

documentation of algorythms
justification of assumption (\textdollar var not null)

4.4 documentation key properties

methods & constructors
arguments & input state, results & output state, effects

data structures
value & structural invariants, one-state & temporal invariants

algorythms
behaviour of snipptes, explanation of control flow, justification of assumptions

4.5 how to document

comments
simply write text; has limited tool support

types & modifiers
final, private etc; tool support: has static, runtime checking & auto-completion

effect systems
produces overhead; read-write effects, allocation/de-allocation, locking, exceptions (try, catch or "throws IOException")

metadata
annotations / attributes for syntactic & semantic information, tool support: typechecking, static, dynamic processing

assertions
specify semantic properties in code; tool support: runtime, static checking, testcase generation

contracts
assertions for interfaces & implementations, method pre- & postconditions, invariants; tool support: runtime, static cehcking, testcase generation

techniques
tradeoff between overhead, expressiveness, benefit, precision
more formal → more tool support
mix different techniques

4.6 informal modeling

design iteratively; underspecification and then add details, and design decisions (algorythms, data structures, control flow)

specific different views on design
architecture (crash possible?), test generation (all states reached?), security review (authorization valid)

design specification
source code must decide, but design descisions difficult to extract

with UML

strengths
describe particular views, omit information or specify it informally, graphical notation makes communication easier

weaknesses
precise meansing unclear, incomplete / informal model lack tools support, many details are hard to despict

4.7 modeling

4.7.1 abstraction from reality

objects & relations

4.7.2 simplifications

ignore details depending on the purpose of the model

4.7.3 draw conclusions for difficult szenarios by using the simple steps of the model

4.7.4 dealing with complexity

4.7.5 static modeling

describe structure

4.7.6 dynamic modeling

describe behaviour

sequence diagrams
describe collaboration

state diagrams
discribe lifetime of single object

5 UML

5.1 Unified Modeling Language

text, graphical notation

for
documentation, analysis, design, implementation

OMG (object management group) standard recommended

5.2 notations

case diagrams
requirements

class diagrams
structure

interaction diagrams
message passing

state & activity diagrams
actions

implementation diagrams
component model (depdencencies) and deployment model (structure of runtime system)

OCL (object constraint language)

5.3 classes

name (required)
attributes with Type (name : String)
methods with Signature & Type (getName(force: Boolean) : String)

5.4 instances

name:type (underlines, name is not required)
attributes represented with their values

5.5 associations

5.5.1 can be

sends a message, creating, attribute of value, receives a message

5.5.2 line with optional roles (employer, employee) and optional label (works for)

5.5.3 can contain multiplicity (city 1 –is capital of– 0..1 country) (or 3..* for many)

5.5.4 can be directed (person $\longrightarrow$ company); one or unidirectional

5.5.5 aggregation

arrow (with scewed rectagle as arrow head, not filled out)

example
Professor —–<WHITE> Group

"belongs to"

no sharing

5.5.6 composition

arrow (with scewed rectagle as arrow head, filled out)

example
Room —–<BLACK> Building

"is part of"

no sharing

5.5.7 generalization

arrows, with traingle as head

example
Professor —-|> Person

"is a"

inherits attributes & methods

5.6 dynamic modelling

make only for classes with significant dynamic behaviour
use only relevant attributes

5.7 sequence diagrams

instances of actors & objects as columns
rows as time units
method calls as arrows which connect the different columns
grauer balken in einer column zeigt wie lange die aktion geht, startet bei method call (arrow to the column), stopt bei return (arrow from column away)

creation / descruction
arrow to object (so column starts more to the bottom), cross means deconstruction (by garbage collector)

views
can draw rectangle (with left top corner has description)
write "par" to make method calls parallel
write "alt" for if/else branches, dividing alternative action with a dashed line, writing the condition or [else] at the left

5.8 state diagramms

black point as start, arrow to states (rounded rectagle), allow to ned states

arrows
contains event [condition] (not required) and specified action. example descriptions: "open()", "[low memory]", "after 10s"

endmarker
back point with cycle around

state
contains do activity, entry, exit action (activity which get executes in state (do), action on reaching the state (entry), action on leaving (exit))

event
something that happens at specific point in time (time event, message receive)

action
operation in response to event (computation)

activity
operation performed as long as object in specific state (continuous computation)

5.9 contracts

OCL (object constraints language)

5.9.1 used to specifiy

invariants of objects, pre/post conditions of operations, conditions

5.9.2 special support for

navigation thorugh UML, assiciations

5.9.3 can use

self (as own context), attributes, role names, side-effect free methods, logical connectives, operations in integers / sequences

5.9.4 example

context Person inv
self.age ≥ 0
context Person inv
self.Dog.age ≥ 0
context Person:Work(time:int)

pre
time ≥ 0

post
HasWorked() = HasWorked@pre() - time

5.10 mapping models to code

5.10.1 MDD

model driven development

5.10.2 generate code from models

5.10.3 advantages

support many platforms, avoid boilerplate code, leads to uniform code, enforce coding conventions, models are not mere documentation

5.10.4 problem

abstraction mismatch (not always possible to map to code, modle should not depend on specific language)
specification incomplete ("open()") / informal ("all conditions met")

switching between model & code
modifications of code (due to the the stuff mentioned above) has to be synced with models

5.10.5 reality

works in specific domains (business process modelling)
works for basic properties

5.11 formal models

notation & tools are based on mathematics (and therefore precise)
describe some aspects of a system
enable automatic analysis (find ill-formed example, proving properties)

6 ALLOY

6.1 what

formal modelling langauge based on set theory
specify collection of constraints
alloy analyzer generates example based on constraints

6.2 signatures

like a class
set of atoms (instances)
different sig $\rightarrow$ different sets

example
sig Person {}
sig Professor extends Person {} //prof is in the set of person
abstract sig Human {}
lone sig God {} //one or none
one sig Truth {} //exactly one
some sig Person {} //some is default, 1 or many

6.3 fields

fields declares relation of atoms

6.3.1 example

6.3.1.1 sig Person {

leader
one God //leader or type God, exactly one, is the default

hero
lone Person //may has a hero, may has not

children
set Person // 0 or many children

parents
some Person //1 or many parents

6.3.1.2 }

6.4 set operators

union
+

intersection
&

difference
-

subset
in

equality
=

cadinality
#

empty set
none

universal set
univ

6.5 relation operations

6.5.1 cross product

→
creates a tuple

6.5.1.1 sig State {

aircraftLocation
Aircraft → AircraftLocation

6.5.1.2 } {

all a
Aircraft | some ap : Airport | (a → ap) in s.aircraftLocation

6.5.1.3 }

6.5.2 relational join

.
connects properties

6.5.2.1 sig Person {

friend
Person

6.5.2.2 } {

all f
friend.friend | all

6.5.2.3 }

6.5.3 transposition

~
reverses relation

6.5.3.1 sig Person {

friends
set Person

6.5.3.2 } {

friends = ~friends

6.5.3.3 }

6.5.4 transitive (reflexive) closure

^, *
FSObject in Root.*contents
(File+Dir-Root) in Root.^contents

6.6 constraints

negation
! or not

conjunction
&& or and

dijunction
|| or or

implication
⇒ or implies

alternative
else

equivalence
⇔ or iff

quantifications
no, some, lone, one, all

6.7 some rules

#{ f: FSObject | f inFile + Dir } > 0
#(File + Dir) > 0

all p
Person | p.hasFriend

all p1, p2
Person | p1 in p2.*friend

all p1
Person, p2 : Professor | some p3 : Person | p1 = p2 or p3 = p2

all disj p1, p2
Person | p1 != p2

all b
bookings | this in b.consistsOf

all disj s, t
Student | s.id != t.id

all i
ID | one s: Student | s.id = i

all s
Student | (s.university != none) ⇔ (s.isLegal = True)

6.8 predicates & functions

6.8.1 predicates are named formulas

pred isLonely[p
Person] { all p2 : Person | no p in p2.friend }

6.8.2 functions are named expressions

fun loneyFriends[p
Person] : set Person { all p2 : Person | p2 in p.friend | isLonely[p2] }

6.8.3 can run predicate or function to find examples

run loneyFriends
run loneyFriends for 5
run loneyFriends for 5 Friends, 6 Professor
run loneyFriends for exactly 5 Friends
run loneyFriends for 5 but (exactly) 3 Friends

6.9 facts

add constraints that always hold
fact { all p : Person | #(lonelyFriends[p]) = 0 }

6.10 assertions

assert my_assert { all p : Person | #(lonelyFriends[p]) = 0 }
check my_assert for 5

6.11 under/overconstrain

underconstraining
permit impossible structures

overcontraining
disallows valid structures

inconsistencies
if fact (1 != 0) → all will pass!

avoid overconstraining
just use assertions wherever possible

6.12 alloy for dynamic

6.12.1 pred init[u

User]{
#u.forSale = 0

6.12.2 }

6.12.3 pred offer[u, u'

User, i: Item] {

(#u.forSale < 3 or u in PremiumUser) ⇒
(u'.forSale = u.forSale + i)

else
(u'.forSale = u.forSale)

6.12.4 }

6.12.5 pred inv[u

User] {
#u.forSale > 3 implies u in PremiumUser

6.12.6 }

6.12.7 assert invHolds {

6.12.7.1 all u

User | init[u] ⇒ inv[u]

6.12.7.2 all disj u_before, u_after

User |

all i
Item | (inv[u_before] && offer[u_before, u_after, i] ⇒ inv[u_after]

6.12.8 }

6.13 alloy for dynamic with states

pred update[a, a':Person] {
}

6.13.1 pred removeAll[a, a':Person {

a'.friends = none

6.13.2 }

6.13.3 pre inv[] {

6.13.4 }

6.13.5 assert initEstablishes { all s': State, ... | init[s', ...] ⇒ inv[s'] }

6.13.6 check initEstablishes

6.13.7 assert opiPreserves { all s, s': State, ... | inv[s] && opi[s, s', ...] ⇒ inv[s'] }

6.13.8 check opiPreserves

6.13.9 open util/ordering[State]

6.13.10 fact traces {

init[first] &&

all s
State - last |
(some... | op1[s, s.next, ...]) or ... (some... | opn[s, s.next, ...])

6.14 alloy simple automata example

sig Counter { n: Int }

pred inc[c, c'
Counter] { c'.n = c.n.add[Int[1]] }

pred init[c
Counter] {c.n = Int[0] }

fact traces { init[first] && all c: Counter - last | inc[c,c.next] }

6.15 analyzing models

consistency
F is consistent if it can be fullfilled there_is s ∗ C(s) ˆ F(s)

validity
if it evaluates to true always when all constraints are satisfies for_all s ∗ C(s) ⇒ F(s)

check for valid
sig Node { next : Node}
check for 3
→ generate (1,1), (1,2), (1,3), (2,1), ...
→ generate constrains from formulas
→ filter out generated model which do not fullfil constraints

consistency checking (done with RUN command)
so alloy translates constrains & formula and tries to find assignement → if yes, display model

validity checking (done with CHECK command)
alloy checks for invalids because its faster (inverse validity definition)
so alloy translates constrains & negated formula and tries to find assignement → if no, all valid

7 COUPLING

7.1 Representation exposure

if modules expose internal data to clients they get tighly coupled
data representation is difficult to change
modules cannot maintain invariants
concurrency very complex
unexpected side effects if exposing sub-objects / structures

7.1.1 shared data structures

modules get coupled, problems with changing, concurrency, side effects

7.1.2 approach 1 (restricting access to data)

can only access to simple restrictive interface
information hiding

non-leaking
do not return references to internal objects (clone if necessary)

non-capturing
do not store arguments

facade pattern
single, simplified interface without hiding the details completely

7.1.3 approach 2 (making shared data immutable)

copies (to change data eventually) remain run-time performance problem

flyweight pattern
pool of Flyweight; client requests one with a key; if not found it is created and added to a pool, then return to client

7.1.4 approach 3 (avoid shared data)

just copy changed data

7.1.4.1 pipe & filter

data flow for communication; no common state

7.1.4.1.1 filter

read data from input; compute; write data to output

7.1.4.1.2 pipe

streams; join / split connectors (the lines between the filters)

7.1.4.1.3 properties

data is processes incrementally, filter independent, output beginns before input finished, filters dont know the others

7.1.4.1.4 filters

input/output stream; may lookahead, may have local state, repeat till no more input

example
split duplicate, split RR

7.1.4.1.5 fusion

combine filter; reduce communication cost, less paralellization

7.1.4.1.6 fission

split filters; introduce parallelism, more communication needed

7.1.4.1.7 strenghts

reuse (if filters have same format), ease of maintenance (single filters can be replaced easely), parallelism

7.1.4.1.8 weakness

sharing global data is expensive, difficult to design, not interactive, error handling very difficult, no complex data can be passed (ASCII on linux)

7.2 procedural coupling

7.2.1 problems

reuse (multiple objects coupled, no seperation of concerns), adaptation (changes in callee may needs change in caller)

7.2.2 approach 1 (move code)

move code to seperate concerns;
common to duplicate code to not be dependent on other companies

7.2.3 approach 2 (event based style)

components generate events, and register for events
generators do not know subscribers

7.2.3.1 observer pattern

subject with Attach() Detach() Notify() { call Update() for all attached subjects }

7.2.3.2 model-view-controller

controller
handle input

model
contains core functionality

view
displays info

implications
user-interface & models must stay consistent

aufbau
view —sends events-→ controller <–receives update notifications– model

7.2.3.3 strenghts

stong support for reuse, adaptation

7.2.3.4 weakness

loss of control, ensuring correctness difficult

7.2.4 approach 3 (restricting access)

enfore policy what can be called by what module → ”layering”

example
presentation, logic, data

strengths
patition complex problems, maintenance easy, reuse (can exchange layers)

weakness
performance

7.3 class coupling

7.3.1 inheritance couples sub to superclass

changes in super may break subclass, limited options for other inheritance

7.3.2 approach 1 (replace inheritance with aggregation)

replace with aggregation, subtyping, delegation

7.3.2.1 aggregation

take methods needed from another class and present it as own

example
have object of class cat inside dog; and expose properties needed for dog; but let cat execute it (dog.walk = cat.walk)

7.3.3 approach 2 (use interface)

replace occurrence of class name with supertypes
use the most general type needed (or interfaces)

let clients construct the superclass (__construct(IInterface \textdollar implementation));

but difficult to test

7.3.4 approach 3 (delegating allocation)

dependency injection
allocations are defined in config file, framework does the initialization

factories
delegate allocation to special class (abstract factory) which does this
concret factory (which implementes the abstract factory) is chosen by the client

7.3.5 low coupling is design goal

7.3.6 trade offs

perfornace & convenience, adaptability, code cuplication

7.3.7 coupling to stable (framework) classes less critical

8 ADAPATION

8.1 changes

software changes frequently
new features, interfaces, performance tuning

8.2 parameterization

prepare modules for change

8.2.1 parametric in

values they manipulate
not two explicit; use list

data structures they use
interfaces & factories

types they use
use generic types / base types

algorythms they apply
use delegates

8.2.2 strategy pattern

interface Selector<D> (”Strategy”)
class MySelector<D> implements Selector<D> (”Strategy1”)
client deals with Strategy<D>
strategy is selected /passed by method call
encapsulate different algorythms from client

8.3 specialization

8.3.1 dynamic method binding

methods can be specialized by overriding & dynamic method binding (inheritance)

8.3.2 can be understood as a case distinction

8.3.3 drawbacks dynamic method binding

reasoning
invariants maintaining?

testing
more potential behaviours

versioning
harder to evolve without breaking subclasses

performance
overhead of method lookup at runtime

→ choose binding of method carefull; apply final or virtual keywords

8.3.4 state pattern

Context → state (which is implemented by ConcreteState1, ...)
context has state as variable; and can choose at runtime which state to apply (can change in between executions)
state changes behaviour

8.3.5 visitor pattern

traverse structure of objects
IVisitor which contains a method overload for each element needed to be traversed
IElement contains Accept(IVisitor v) {v.Visit(this);} method
→ visitor is now central point to print / save all elements

8.4 summary

parameterization
supply different arguments to modify behaviour

specialization
adding subclasses / override methods to modify behaviour

9 TESTING

9.1 why bugs

predicting the behaviour of source code is difficult

mistakes
unclear requirements, wrong assumptions, design & coding errors

9.2 increase reliability

fault avoidance
detect faults statically, development methododologies, review, program verification

fault detection
detect faults while executing the program; testing

fault tolerance
recover from faults at runtime, adding redundancy (n-version programming)

9.3 testing general

successful test find error
error is deviation from desired outcome (by function, non-function requirements)
execute program to find error

9.3.1 impossible to test fully

theoretical
termination

pratical
pohibitive in time & cost

9.3.2 stages

requirements elicitation
system tests

system design
integration tests

detailed design
unit test

9.4 test harness

test framework

testdriver
applies testcases to Unit Under Testing (UUT)

UUT uses Test Stub's, implementations of components used by UUT (provides fake data, simulates environement)

9.5 Unit Testing

testing individual subsystems;
confirm each subsystem works correctly
need unit test for each input values → to get reasonable coverage need to test multiple

parameterized unit tests
unit tests with arguments which can be set by the test framework, avoid boilerplate, allows generation of test data

9.6 test execution

execute test cases, re-execute after every iteration

regression testing
ensuring everything still works after applying changes

9.7 rules

fully automatic
test must be excutes fully automatic and check their own results

test suite
reduce time needed

run frequently
at least once a day

unit test to expose bug
if a bug report received; write unit test that exposes it

incomplete testing > no testing

boundary conditions
concentrate on these cases ”edge cases”

exception testing
test exceptions when things go wrong

write tests that catch most bugs, instead of writing none

9.8 integration testing

testing groups of subsystems; and eventually the whole system
confirm interfaces between subsystems
bottom-up (top not implemeneted yet), top-down (bottom submodules not implemenetd yet), big-bang approach (test all in once)

9.9 system testing

test entire system
determine if system fulfills functional & non-functional requirements

9.9.1 strategies

9.9.1.1 functional requirements

functional tests

goal
test functionality

test system as black box

testcases based on use cases

desribe
input data, flow of events, resuts (which are checked)

9.9.1.2 non-functional requirements

performance tests

goal
test performance

9.9.1.3 clients understanding of requiements

acceptance test

goal
demonstrate that the system meets the requirements

performed by the client!

alpha test
customer @ developer; which is ready to fix bugs

beta test
@ clients site, developer not present, realistic workout in target environement

9.9.1.4 user environement

installation tests

9.10 independent testing

programmers test happy paths because they have vested interest not to find mistakes
testers must seek to break the software → should be independent
all but unit tests should be performed by testers

facts
the developer should test himself
testers are involved from the start
testers work together with developers at test suite
testers are not solely responsive for quality of software

9.11 testing steps

select what will be tested
select test strategy
define test cases
create test oracle (expected results)

9.12 testing strategies

9.12.1 exhaustive testing

check UUT for all possible inputs

9.12.2 random testing

select data uniformly

goal
cover corner cases

advantage
avoids designer bias, tests roboustness (reation to invalid input)

disadvantage

treats all inputs the same

for all test stages

9.12.3 functional testing

requirements knowledge determines test cases

goal
cover all requierements

find incorrect functions, interfaces errors, performance leaks

limitations
does not detect design / coding errors, does not reveal errors in specification

for all test stages

9.12.4 structural testing

design knowledge determines test cases

goal
cover all code

limitations
focus on code and not requierements, requires design logic (only programmers know), highly-redundant tests

for unit testing

10 FUNCTIONAL TESTING

10.1 partition testing

divide input into equivalence classes
choose test cases for each equivalence class

10.2 selecting representative values

after partitioning; select concrete values from each of the partitions to test
large number of errors occurr at boundary of the input domain → so select elements of edge of equivalence class & some from the middle

10.3 cominatorial testing

combine boundary testing & equvalence classes
too much example to test if combined

select specific combinations
semantic constraints, cominatorial selection, random

do not select unnecessary combinations (which have no influence to each other)

semantic constaints
at least one test case for each constraint

pairwise combinatorial testing
two or three values interactions reveal most errors
focus on all possible inputs for each pair of inputs
reduces the number of inputs drastically
important if a lot of system configuration needs to be tested
combine with other approaches

11 STRUCTURAL TESTING

11.1 why

detailed design & coding introduces behaviours which are not specified
White-box test a unit to cover a large portion of its code

11.2 control flow testing

11.2.1 basic block

block of code with one input & one output point; upon entering the rest of the code is executes once, in order

11.2.2 intraprocedural control flow graph (CFG)

top to bottom
entry block
arrows to each basic block
label @arrows have condition written on it (example b2 = (i < a), produces two arrows b2, -b2)
point to exit block when finished

11.2.3 coverages

statement coverage
how many portions of the CFG are executed (nodes & edges)
#excuted / #total
→ but still possible to miss bug

branch coverage
test all possible branches in control flow (edges)
complete branch coverage implies complete statement coverage
→ still possible to miss bugs

path coverage
test all possible paths (sequence of branches)
complete path coverage implies complete branch coverage
→ not feasible with loops (arbitrary # of paths)

loop coverage
for each loop, test 0, 1, and 1+ iterations
coverage = #loops with 0,1,1+ iterations / #loops ∗ 3s

data flow coverage
evaluated with DU pairs
coverage = #DU-pairs / used DU-pairs

11.2.4 method calls

CFG treat method calls as simple statments; but they may invoke different code depending on state
testing dynamically bound by viewing it as a case distinction for all possible implementations → then do branch testing
but this leads to combinatorial explosion → use semantic constraints & pairwisecombinations testing

11.2.5 exceptions

documented exception (checked) (as CollectionEmptyException)
can be treated like branches

undocumented (unchecked) exception (MemoryOverflowException)
impractical to represent all in CFG

checked exception
invalid conditions outside the immediate control of the program (invalid user input, network outage)
are declared in method signatures in java
test like normal control flow

unchecked exception
defects in the program or execution environement (illegal arguments, division by null)
ignore exceptions thrown by other methods, but consider throw staments in own code
never use unchecked for control flow! (like NullPointerException)

11.3 data flow coverage

Test those paths where a computation in one part of the path affects the computation of another

11.3.1 variable definition

basic block that assigs a variable to v

11.3.2 variable use

basic block that uses the assigned variable

11.3.3 definition clear path

n1, ..., nk where n1 defines the variable, and nk uses it → do not necessarily go from entry to exit

11.3.4 DU-pairs

defintion-use pair (DU pair)
defintion clear path in the CFG

DU-pair coverage
test all paths that provide a value for variable use

(1,3)
1 is LineNr where the variable was defined, 3 is LineNr where the variable is used

11.3.5 determining DU pairs

Reach(n)
contains all the defintions made from before (UNION from all paths)

ReachOut(n)
contains all definitions which survive this line (most of the time Reach(n)

== ReachOut(n))

evaluate Reach(n) & ReachOut(n)
1. make a table with columns lineNr(n), Reach(n), ReachOut(n)
2. start from top to bottom, with Reach(1) is empty (leere Menge)
3. for each line, if variable is assigned put variable_name_line_number into ReachOut(n), else put Reach(n)
4. join in loops and gotos

evaluate DU pairs
1. build tbale as described above
2. get all reading locations of variable in question
3. for each reading location (say line 6), look at Reach for the corresponding line (say var_1, var_3) and build any possible combination ((1,6),(3,6))

11.3.6 complete DU coverage needs more than one loop iteration

11.3.7 choose testing that maximizes branch & DU-paris coverage

11.3.8 measure DU-pair coverage with maps

11.3.9 not all DU-pairs are feasible (has to over-approximate)

11.3.10 DU-pair anomalies may detect errors

double-definition, use of unassigned, no usage

11.4 interpreting coverage

high coverage does not mean code is well tested, but contrary applies
coverage tools help to find parts of software which are not well tested
test suite grows exponential with coverage
criterias lead to better testing than random testing
more demanding coverage cirteria leads to bigger test suites but not to detecting more bugs
cost-efficieny of all test aproaches about the same

experimental evaluation
seed defects in code; test with test suite and check if it is catched

12 SOFTWARE ENGENEERING

12.1 pure methods

fullfill both properties
i) does not modify any objects which existed before calling (but may modify objects it has created)
ii) will return the same result if the state is same (same object, same arguments)

example for pure methods
hash()

example for non-pure
returnRandomValue()

12.2 c# contracts

12.2.1 public class MyCheckedClass {

private int[] elems = new int[10];
[ContractInvariantMethod]

private void ObjectInvariant() {
Contract.Invariant(elems != null);

}

private void Set(int[] myElements) {
Contract.Requires(myElements != null);
Contract.Ensures(Contract.OldValue(elems) != elems || Contract.OldValue(elems) == myElements);
Contract.ForAll(0, myElements.Count() - 1, i ⇒ elems[i] == myElements[i])

}

12.2.2 }

12.3 patterns

12.3.1 creational pattern

create objects in a manner suitable for the situation

abstract factory
creation method which returns the Factory itself, which in turn then creates new objects
parses xml configuration files to look for the implemenetations to use; then returns a factory with that injected information
→ newInstance() method

builder
creation method which returns reference to itself
php property setter pattern, allows to set multiple props on same line
→ string.append

static factory
creational method which returns an implementation of an abstract type / interface
used with compile-time / configuration data so factory knows what implementations to use
→ NumberFormat.getInstance()

prototype
creation method return different instance of itself
create() method in entities
→ Object.clone()

singleton
creation method returning same instance everytime
static class in hiding
→ getInstance() method, Dektop.getDesktop()

12.3.2 structural pattern

relationships between objects

adapter
taking an instance of a different abstract type & return a new instance which decorates/overrides the given instance
takes an instance and returns another instance which overrides the given instance
→ java.io.InputStreamReader(InputStream)

bridge
taking an instance of a different abstract type & return a new instance which delegates/uses the given instance
takes an instance and returns another instance which is uses the given instance
→ java.NewSetFromMap(map)

composite
behavioral methods taking an instance of same abstract/interface type into a tree structure
in tree, like AddNode()
→ java.awt.Container#add(Component)

decorator
creational methods taking an instance of same abstract/interface type which adds additional behaviour
a IReader takes an IReader as constructor argument; internally may calls the passed IReader
→ InputStream has constructur taking instance of same type

facade
behavioral methods which internally uses instances of different independent abstract/interface types
similar SyncApiService, redirect calls to the correct types
→ java.ExternalContext which uses HttpServletResponse, HttpServletRequest etc internally

flyweight
creational methods returning cached instance
pool of availble objects; flyweigth is asked to return specific one; takes it from the pool or creates new one
→ Integer#valueOf(int), can be made with Boolean, strings, etc

proxy
creational methods which returns an implementation of given abstract/interface type which in turn delegates/uses a different implementation of given abstract/interface type
ProxyUserService which uses the GeneralUserService. GetUserService() would the method be named
→ the services of a DAL in java

12.3.3 behavioural

communication patterns between objects

chain of responsibility
methods which (indirectly) invokes the same method in another implementation of same abstract/interface type in a queue
passing on certain input arguments based on the value of those, logger (by LOG_LEVEL) or middleware in slimPHP
→ java.util.logging.Logger#log()

command
methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been encapsulated by the command implementation during its creation
RelayCommand()
→ javax.swing.Action

interpreter
behavioral methods returning a structurally different instance/type of the given instance/type
parsing/formatting is not part of the pattern, determining the pattern and how to apply it is
→ java.text.Normalizer

iterator
methods sequentially returning instances of a different type from a queue
IEnumerate etc
→ java.util.Enumeration

mediator
behavioral methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance
Timer.schedule(TimeSpan span, Action action) → the timer executes the action after the given time
→ java.util.Timer

memento
behavioral methods which internally changes the state of the whole instance
Date→setDate("20.08.1995")
→ java.util.Date

observer
methods which invokes a method on an instance of another abstract/interface type, depending on own state
register for events at observer, when event happen the observer will call you
→ javax.faces.event.PhaseListener

state
behavioral methods which changes its behaviour depending on the instance's state which can be controlled externally
scheduler.ExecuteTask() → waits longer or less long depending on CPU
→ javax.faces.lifecycle.LifeCycle#execute()

strategy
methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been passed-in as method argument into the strategy implementation
list.sort() uses a comparator.compare() method to sort the elements
→ java.util.Comparator

template method
methods which already have a "default" behaviour definied by an abstract type
non-abstract methods of else AbstractClass
→ java.io.InputStream

visitor
two different abstract/interface types which has methods definied which takes each the other abstract/interface type; the one actually calls the method of the other and the other executes the desired strategy on it
element1 E with method (IVisitor v) { v.visit(this); }, visitor V with methods (IElement1 elem) { print(elem); } (IElement2 elem) { print(elem); }
→ java.nio.file.FileVisitor implemented by SimpleFileVisitor, method which accept DIR and FILE

12.4 SPL language

very simple language; if-else constructs & derivation rules

12.4.1 basic properties

variables are not declared
expressions have no sideeffects
only basic statements: no functions, heap, exceptions,. . .
semantics usually specified at abstract syntax level

12.4.2 basic building blocks

Z natural numbers
-1 | 0 | 1, denoted x

Var variables
y | x | z, denoted v

A Definition Statement
A∗A | A+A | Z | Var, denoted a → you can evaluate a with: <a, state> (⇒a, turned 90 degrees) v

B Boolean Statement
true | false | BˆB, denoted b

S Statement
skip | Var := A | if B then S else S, denoted s

12.4.3 derive a program

start with non-terminals & derive till no non-terminal can be replaced anymore (end: if x < 5 then x = 5 else skip end

12.4.4 rules of inference

architecture
big line; on top the hypothesis, under the line the conclusions, may has condition to the right

possible structures for hypthesis
arrow to <statement, state>, arrow to state, down arrow (like ⇒ but turned 90 degrees) and then A type to the right (like false, 1, 14, 12 + 14)

axioms
no hypthesis needed

12.4.5 operational semantics

big step
one pyramid; all done in one step

small step
multiple pyramids, c1 → c2 → ... → cn till programm fully evaluated

12.4.6 examples

<stmt1, state1> → state2
———————————
<stmt1; stmt2, state1> → <stmt2, state2>

13 STATIC PROGRAM ANALYSIS

13.1 challenge

build a static analyzer that is able to prove as many programs as possible

13.2 approaches

over-approximation
static analysis

under-approximation
dynamic analysis

over & under approximation
symbolic execution

13.3 cool facts

can prove interesting properties
can find bugs in large scale programs, and detect wrong API usage
combination of math & system building
run the program without giving concrete input
no need for manual annotations (as loop invariants)

13.4 static analysis via abstract interpretation

select/define an abstract domain
select based on the properties to prove

define abstract semantics for the language
prove sound with respect to concrete semantics, define abstract transformers

iterate abstract transformers over the abstract domain till fixpoint is found (fix-point is the over-approximation)

13.5 abstractions

sign
Top; +,-; 0; Bottom
if y is -; y = y+1 → y is Top
if y is 0; y = y+1 → y is Top (imprecise!) or y is + (precise)
+,- include 0!

interval
good for range of variables
fächer; [-infinity, infinity]; [-infnity, -1], [-infnity, 0], ..; [0,0], [1,1] ...; Bottom
a ({ [1, {x→1, y→1}], [1, {x→1, y→5}]}) = 1 → (x → [1,1], y → [1,5])

y (1 → (x → [1,1], y → [1,5])) = { [1,{x→1, y→1}], [1,{x→1, y→2}], ...}
definition of transformer example Fi (m)3 = [y:= 7]i (m(2)) U [goto 3]i(m(6))
start with Top ([-infinity, infinity])

parity
Top; Even, Odd; Bottom

comparable
sign & interval

precise
interval more precise than sign because it has all states of sign + more

13.6 solve abstraction exercises

13.6.1 do flow

1. create table with columns (ptr, variable1, variable2) and rows 1,2,3,... (program labels)

2. go to first label (at 1), fill row 1 with T for all already initialized variables (arguments of function), the rest is Bottom

3. go to second label, evaluate label 1 result in row 2 (if label 1: x = 2, then row 2, variable x = T (in sign))

13.6.2 do transformers

concret
as example take m1 = x → [1,3], y → [4,5], m2 = x → [1,2], y → Bottom
[x < y](m1) = x → [1,3], y → [4,5]
[y > x](m1) = x → Bottom, y → Bottom
[y > x](m2) = x → [1,2], y → Bottom

syntactic
[x := a](m1) = m[x → [p, q]] where <a,m> ⇒i [p,q]

13.6.3 do concrete trace

1. choose start values for all function arguments
2. each line of code gets one entry in trace, {<1, {x → x_0, y → y_0}>, <2, {x→ x_1, y→ y_0}>,...}

13.6.4 do abstract trace

1. do concrete traces
2. combine traces with an abstraction, for example interval. use widening if applicable
example is {1 → {x → [1,2]}} → {2 → {x → [1,3]}} //take all concrete traces & combine them
if you need to create invalid concrete trace you need to simply choose values which are valid in abstract but do not make sense in concrete (so disobey instructions!)

13.6.5 tricks

where sign does more work than interval
x = 1; if (x!=1) {mystatementtodelay}

where interval does more work than sign
x = 1; while (i > 0) {x = x + 1; i = i-1;}

two equal programs (=same final state) but not same interval evaluation
x=2; y=x∗x; i=0
x=2; y=0; i=x; while(i > 0) {y+=x; i–}

interval but not parity
int x = 1; assert x > 0;

parity but not interval
int x = 2∗i; assert x mod 2 == 0

13.7 abstract transformers

how to handle statements of the language on the abstract domain
must be defined once for each programming language
always produces superset of what a concrete transformer would produce
sound if produces superset
precise if superset produced clevery in the respective domain

joins (U)
if need to merge two abstract elements at certain point (due to goto or loops) we perform a join → produce least upper bound

widening (meet, N)
if joins have been executed multiple time; we probably need to widen: [2,2] + [3,3] = [2,inifinity]

14 MATHEMATICAL CONCEPTS

14.1 structures

14.1.1 poset

partially ordered sets
set equiped with a partial order (transitive, reflexive, anti-symmetric)
captures implications between facts

shown as Hasse diagrams
build pairs (lower, higher) and enumerate all possible ones (a,a), (Bottom, Top), ...

least / greatest element
if one element is the least or the greatest (must be only one)

lower / upper bound
all smaller / bigger elements

least upper (U, join) / greatest lower (N, meet)
the single element directly following the elements in question in lower / upper bound

14.1.2 lattice

more constraints than poset
where least upper / greates lower exist for every element of the poset

complete
all subsets are lattices

14.2 functions

replace "≤" with [_ in the following paragraph, "obermenge"

monotone
if a [= b in poset → f(a) [= f(b)
so if a below b then f(a) must be below f(b) too
example b [= c but f(b) ![= f(c) → therefore not monotone (subgroup stuff!)
intuition if g(x) changes branch of poset then function is not monotone

fixed point
iff f(x) == x
set is called Fix(f)
arrow of function points to itself

post-fixed point
iff f(x) [= x
set is called Ref(f)

least fixed point
single smallest fixed point

approximate
g approximates f iff each value in g is same or less precise
f(b) = d ≤ g(b) = c
the function f=infinity approximates all other functions!

14.3 Tarski's theorem

confirms there is a fixed point where dealting with montone functions & complete lattice
post-fixedpoint is above the least fixed point.

14.4 static program analysis

let P be set of reachable states, F function of all input states & transitions possible
F(P) = P is fixpoint

define F# such that it approximates F → is done once for a programming language

use theorems which state that F# approximates the least fixed point of F

automatically compute a fixed point V such that F#(V) = V

14.5 more to F#

F#(x) must be superset of F(x)
to do this; we define an abstraction function alpha a (which puts the value into the abstract domain) and a concretization function gamma y (which reverses)
to prove out F# is correct; we must prove that for all abstract element → concretize it → apply F → abstract it now the result must be less/equal that applying F# directly
we can therefore simply assume F# = T → would be sound, but imprecise

most precise approximation would be $a(F(y(x)) = F\#(x) \rightarrow$ often not possible
is definined for the particluar abstract domain we are working on
f# evaluates to Top if initial label, and [[action]](m(l')) otherwise; action is the abstract transformer

14.6 least fixed point approximation

1
monotone function F: C→C and F#: A→A

2
a & y forming a galoise connection (must be montone; a^-1 = y)

3
F# approximates F (by defintion above) $a(F(y(x))) \leq F\#(x)$

then
a(least fixed point(F)) ≤ least fixed point(F#)

14.7 least fixed point approximation

if a&y do not form a galoise connection

1
monotone function F: C→C and F#: A→A

2
y monotone

3
F# approximates F (by defintion above) $F(y(x)) \leq y(F\#(x))$

then
least fixed point(F) = y(least fixed point(F#))

14.8 relational abstraction

non-relational domain
does not keep the relationshop between variables (for example interval)

relational domains
keeps the relationship (for example octagon & polyhedra)

14.9 octagon domain

constraints of the form
$+- x +- y \leq c$

example
$x+y \leq 4$; $y \leq 10$

14.10 polyhedra domain

constraints of the form
$c1x1 + c2x2 + ... \leq c$

example
$x-3y \leq 10$;

14.11 connecting math & analysis

Complete Lattice
Defines Abstract Domain and ensure joins exist.

Joins
Combines facts arriving at a program point

Bottom
Used for initialization of all but initial elements

Top
Used for initialization of initial elements, widening used to guarantee analysis termination

Function Approximation
Critical to make sure abstract semantics approximate the concrete semantics

Fixed Points
This is what is computed by the analysis

Tarski's Theorem
Ensures fixed points exist.

14.12 pointer analysis

14.12.1 aliases

two pointers are aliases if they point to the same object

14.12.2 points to pair

(p, A) means p points to A

14.12.3 all objects allocated at same label are represented as single object (called A_line_number)

14.12.4 domain

two maps; one maps pointers to abstract objects; the other one maps fields of abstract objects to abstract objects
no widening needed as it is finite
1 → (p→{a1, a2}, a2.f → {a1})

14.12.5 flow sensitive vs insensitive

respects programs control flow vs assume all execution paths are possible (no order between statements)

14.12.6 insensitive algo

1. Write down all variables which occurr (x, y, z)
2. For all objects, note properties (A0.next, A1.next, A0.p, A1.p)
3. start with evaluating variable assignments, afterwards do properties

14.12.7 sensitive algo

1. create table; rows are step 1 & 2 from insensitive algo
2. create columns at critical points; at start, when entering loop, when joining loops, ...
3. evaluate from top to bottom; dont forget to join loops and the special rule about x.p

14.12.8 to remember

can not prove var1 != var2 → because both could be null
after returning from loop (while, for, ..) meet both branches (in & out loop)!

p.f = q where p → {A} and A.f → {B} and q → {C} this gives
A.f → {B,C}

14.13 symbolic execution

between testing & static analysis
completely automatic, but may miss programm executions!
assiciate each value with a symbolic value which acts as a constraint to what is possible in this specific part of the program

keeps two fomulas
symbolic store & path constraint → symbolic state is the conjunction, SMT solver provides possible values

evaluation of conditional affect the path constraint; at start it is simply set to true; each conditional then produces new entries

handling loops
limitation! → we simply replace (\textdollar i = 0; \textdollar i < k; \textdollar i++) with (\textdollar i = 0; \textdollar i < 3; \textdollar i++) → under-approximation!

example
{x → x0} y = x + x {x → x0, y → x0 + x0}

exercises
"given is a label to be reached; enumerate all PCT which reach this statement"

14.14 concolic execution

combine symbolic execution & concrete execution (normal)
concrete execution should drive the symbolic execution
we differentiate between arguments to the function (symbolic) and values (like loop variables) created inside function (concrete)

14.14.1 steps

addtionally to the symbolic store we keep a concrete store; where variables have explicit values
we choose starting values for the function arguments and let this determinte the path we take
we track assignments with concrete values & symbolic values
we track condition statements by adding it to the path constraint
after finishing, we negate parts of the path constraints and build new starting values (with SMT)

14.14.2 solve exercises

choose start values for symbolic values
add loop variables & similar to concrete store (use symbolic variables in their definition if possible)
go through code; when there are if statments (don't forget loops) add it to the path constraint

negate parts of the path contraint to produce new input which reaches different places

example suppose e_0 and b_0 are out start value, PC_0 could look like this
(0>e_0) && (b_0==2)

14.14.3 better than symbolic

we can now use the concrete store to evaluate methods from outside out scope (by storing the concrete return value)
this allows to evaluate non-linear stuff; but may prevent us from reaching all reachable statments (under-approximation)

14.15 SMT solver

converts boolean path constraints to concrete possible assignments to fullfil it

14.15.1 constraint solving critical for performance

SMT solver should support as many logical fragments as possible
SMT solver should be able to solve them quickly
the engines should try to exploit domain structure to make the SMT formulas easier

optimizations
caching → just try if last result from SMT still works; if no only then call SMT again

some SMT fomulas may be unable to be processed by the solver!

14.15.2 non-linear constraints

difficult for SMT solver; hence they will under-aproximate