Model Driven Development: software development method that focuses on creating + exploiting models in software development activities

Forward engineering: translating models to code / reverse engineer

MBE: model-based engineering (analyze systems)
- models not main focus, code may still be written manually

MDE: model-driven engineering (automate engineer. activ)
- models kept updated, minor adjustments

MDD: model-driven development (automate software dev)
- code generated by model, changes made to model not code

MDA: model-driven architecture (practice MDD w/ OMG stand)
- specific standards (most strict, least flex)

Model is abstraction of a software that can be analyzed before the software is built and used to automate software dev
- → abstracted w/ generic modeling language (UML)
- → domain-specific language (DSL) (SQL, HTML)

abstract model —model transformation→ concrete model —code generation→ code

Benefits: write code generator once and use many times, specifying model to generator and invoking faster than writing code manually
- simple: source of truth is model not code, easier to analyze
- portable: same model can generate code for diff platforms (lang, framework, OSs) / artifacts (code, db schema, spec)
- consistency: generated have desired princ, design, naming

Issues: maintenance: must have competency, bugs, new dependencies. code becomes dep on code gen tool
- complexity: less optimized than manual, code templates may support more complex use cases than need

Develop a Code Generator
1. dev modeling language:
   - lang syntax: write .xtext that def. UML lang
   - generate lang API: run xtext gen to create lang infra
   - use API to add validation rules
   - semantics / dev write / see / mapping
   - abstract syntax: defined w/ class diag (metamodel)
   - concrete syntax: textual / graphical (grammar)

Xtext allows the def of a modeling language's textual grammar and metamodel together using a form of BNF. other grammars can be imported or re-used
- grammar and metamodel has unique name
- metamodel has a name and a unique URI

Classes Rules: <name>: <expression>;

Enum Rules: enum <name>: <literal1> = '<keyword1'>;

Keywords: can contain terminal / non-term. symb

Terminal enclosed w/ quotes + repr. gram' keywords

Attributes: non-terminal repr class features

<name> = <type> ← feature syntax
- <type> is data type (ID, STRING, INTEGER, DOUBLE), the feature is an attribute of the rule's class
- STRING is quoted string ("hello") but ID is not (no quotes)
- <type> is bool: <name> ?= '<keyword>' static ?= 'static'

Compositions

<name> = <rule>: feature repr. a composition from the feature rule's class to the expr class

Associations

<name> = [<rule>]: assoc from feature rule's class to the expr rule's class

[<rule>]: cross-ref to an existing elem of rule's type

[<rule> | ID]: explicit that cross-ref by ID

Inheritance: rules whose <expression> has the OR syntax: <rule1> | <rule2> ... | <rule> ← inher of main rule by each OR'd rule

Cardinalities: default cardinality is [1] for <name> = <type>

<name> ?= <type>? [0..1], <name> += <type>+ [1..*]

<name> += <type>* [0..*]

2. dev a code generation templ using the model lang API
   xtext allows code generator to be developed w/ Xtend
3. dev an app model using modeling lang and run it through code generation templ → appl code (UML)

```
x=c; y=0
while (x>0){
  x--;
  y++;
}
```

Loop entry: x=c, y=0

After iteration 1: x=c-1, y=1

After iteration 2: x=c-2, y=2

Inductive Hypothesis (Loop Invariant): x+y=c

1) J holds initially $P \Rightarrow J$
2) J is maintained $(J \land B)\, S\, (J)$
3) J is sufficient: $(J \land \neg B) \Rightarrow Q$
4) vf is bounded: $(J \land B) \Rightarrow vf \geq 0$
5) vf decreases: $(J \land B \land vf=VF)\, S\, (vf<VF)$

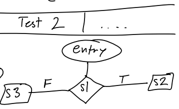
Geometric Series: $\sum_{i=1}^{N} x^i = \frac{x(x^N-1)}{x-1}$

Mutant Kill: Whether test errors on 1 variant introduced

Weakest precondition e.g., a,b != null, a>b, b≥0, ...

CFG Edge Cover Matrix (given CFG G)

∀(x,y) ∈ G	Test 1	Test 2	...
(entry, s1)	1		
(s1, s2)	0		
(s1, s3)	1		
⋮			

⇒ 

Software Testing: observed vs. expected behavior
driven by white-box testing / black-box testing

fault of omission: missing funct, incomplete impl

fault of commission: extra/unexpected func, side eff

Unit testing (dev): indiv. modules of source code working

Regression (dev + testers): prev tested software works after change

Integration (test): interface between 2 or more unit tested mod

System (testers): fully integrated systems using E2E scenarios

Acceptance (users): user req. on release candidate of the system

Isolating the Item-Under-Test (IUT): reduce uncertainties

Test stub: simulate behavior of comp IUT depends on (called)

Test driver: test IUT when callers not available yet (test from above)

Test Driven Development: writing tests first before code
- forces you to write testable code, output compared to expected result (gold stand / test oracle)

Software Test Automation: human testers expensive / inconsist
- provides rapid feedback / auto exec of software tests

JUnit: Test Suite [(Test Case 1: Test Fixture (setup), Test Method..., Test Fixt (tear down)]

@BeforeClass: runs before all test cases in the class @AfterClass

@Before: executes before each indiv. test method @After

@Test: marks a method as a test case

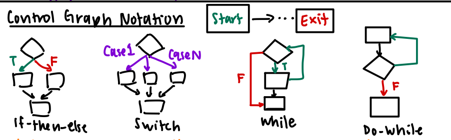
```
public class DummyTest { private static List<String> list;
  @BeforeClass public static void bC() { list = new ArrayList<String>();}
@SuiteClasses({JUnitTestCase1.class, etc.}) public class JUnitSuite{}
```

Assertions: void

assertEquals(bool exp, bool act), assertTrue(bool cond) / False

assertNotNull/Null(Object obj), assertSame/NotSame(1,2) — 2 obj references point to same loc in memory

Control Graph Notation [Start] → ... [Exit]



statement coverage: % of statements exercised by tests

branch coverage: % of branches (cond evaluations)

path coverage: % of control flow paths exerc. by tests

Input	Exercised Statements	Exercised Branches	Exercised Paths
T1(x<1)	s1, s2, s3	b1, b3, b5	[b1, b3, b5]
Coverage	70%	50%	12.5%
T2(x>2)			
Coverage			

Estimating # of paths (bounded programs)

Loop unroll → # of paths = $2^{(\#\text{ of non-det. decisions})}$

`for (int j=1; j<= Math.pow(3,i); j++)` # of feas. paths

$T(n) = 3 + 3^2 + 3^3 + \dots + 3^n$

$3T(n) = 3^2 + 3^3 + \dots + 3^n + 3^{n+1}$

$2T(n) = 3^{n+1} - 3$, $T(n) = \frac{3^{n+1}-3}{2}$

Math.pow(2,k)%2==0 is deterministic $2^{\left(\frac{3^{n+1}-3}{2}\right)}$

Symbolic Execution

For each decision, propagate constraints for T/F branches

feasible paths: Path 1: (TT) x>3 AND 2x+1>x^2 (NOT FEAS) (repeat)

Test Generation

Find concrete input for each path cond after symbolic exec.

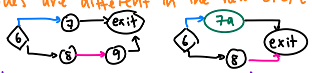
Regression Test Selection (RTS): P (old ver) P' (new) T: test suite

Assume all tests in T ran on P ⇒ generate coverage mat. C

Given Δ between P and P' and C, identify subset of T that can identify all regression faults

Harold & Rothermel's RTS

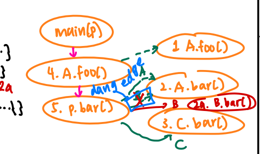
dangerous edges: edges in the old CFG whose target nodes are different in the new CFG (as effec as running all test cases)



Edge	T1	T2
(entry,1)	0	1
(6,7) (8,9)	1	0

polymorphism allows memb function call to be resolved at runtime

```
class A {
1 public static void foo(){...}
2 public void bar(){...}}
2a class B extends A { bar }
3 class C extends B { bar(){...}}
void main (A p) {
4 A.foo();
5 p.bar();}
```



Data flow testing: exploring the paths between data def + use

Randomized testing: choosing samples from the input space (rand)

model-based (black-box): testing based on how should work, don't look at code

mutation: change code and sees if it fails
- value mutation: changes to value of constants, params, loop bounds
- decision: changes to conditions to reflect common slips + errors
- statement: delete lines, swap orders, change math operations

adequacy of T: mutant killing ratio $\frac{k}{n}$ — kills k out of n mutants

Open Problems — refine for test gen & fault det.

Prompt Engin: reduce test flakiness, fault revelation, test bal, realistic inputs, corner cases, production behavior

Augmenting Existing Tests: can LLMs gen. add assert for hist. faults/edge cas

Test Correctness: lack of automated oracle — aut checks if output is correct

Hallucination & Test Correctness: LLM's prioritize likelihood > correctness, risking inaccurate test oracles
- reveal mismatches between prog semantics and expect.

Mutation Testing: Train LLMs on fault models to generate mutants coupled to real-world faults

LLM-based Repairs: sign. acc. improv in APR tasks, effect. across diverse bugs, cost-effective + scalable solutions

LLMs Extend Opt. Techniques: diverse appl (compiler instr)

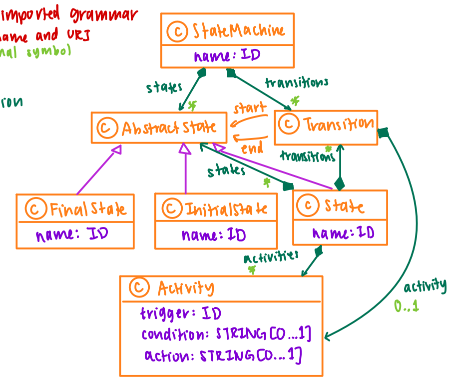
Challenges: Correctness, Scalability

grammar
grammar grammar name with imported grammar

generate statemachine metamodel name and URI

nonterm = class feat — terminal symbol

```
StateMachine:
  'state' 'machine' name=ID ('{'
      states += AbstractState*
      transitions += Transition*        composition
  '}')?;
AbstractState:
  State | InitialState | FinalState;
State:
  'state' name=ID ('{'
      activities += Activity*
      states += AbstractState*
      transitions += Transition*
  '}')?;
InitialState:
  'initial' name=ID;
FinalState
  'final' name=ID;
Transition:
  start=[AbstractState] '->' end=[AbstractState] (':' activity=Activity)?;
Activity:
  trigger=ID ('[' condition=STRING ']')? ('/' action=STRING)?;
```




```
state machine GumballMachine {
  initial start
  state S1
  state S2
    entry ["x<10"] / "x++"
    exit / "x--"
    initial I2
    state S21
    ⋮
  }
}
final
I→S1
S1→S2 }
```

Hoare Logic
For any predicate P and Q any programs S, the {P} S {Q} Hoare triple says that if S is started in (a state satisfying) P, then it terminates in (a state satisfying) Q
- If {P} S {Q} and {P} S {R}, then {P} S {Q and R}
- strongest postcondition of S w/ respect to P: the most precise Q such that {P} S {Q}
- weakest precondition of S w/ respect to R: the most general P such that {P} S {R}, written wp(S, R)
- In fact {P} S {Q} holds iff P ⇒ wp(S, Q)

- wp(skip, R) ≡ R
- wp(assert P, R) ≡ P and R
- wp(w := E, R) ≡ R[w := E]
- wp(S; T, R) ≡ wp(S, wp(T, R))
- wp(if B then S else T end, R) ≡ (B and wp(S,R)) or (!B and wp(T,R))
- wp(if B then S end, R) ≡ (B and wp(S,R)) or (!B and R)

{P} while B do S end {Q}

Loop variant J and a variant function vf s.t.:
1. J holds initially: P ⇒ J
2. J is maintained: {J and B} S {J}
3. J is sufficient: (J and !B) ⇒ Q
4. vf is bounded: (J and B) ⇒ (vf >= 0)
5. vf decreases: {J and B and vf=VF} S {vf<VF}

```
public static int power(int x, int n){
  int p=1, i=0;
  while (i<n){
    p=p*x;
    i=i+1;
  } return p }
```

P: n>=0, Q: $p=x^n$, J: $p=x^i$ and i≤n, vf: n-i

Step 1: P ⇒ J

{0 ≤ n} ⇒

≡ {0 ≤ n}

{1=1 and 0≤n}

p:=1

{$p=x^0$ and 0≤n}

i:=0

{$p=x^i$ and i≤n}

Step 2: {J and B} S {J}

≡ {$p=x^i$ and i≤n and i<n}

{$p \cdot x = x^i x$ and i<n}

p := p*x

≡ {$p=x^{(i+1)}$ and i+1 ≤ n}

i := i+1

{$p=x^i$ and i≤n}

Step 3: {J and !B} ⇒ Q

{$p=x^i$ and i≤n and i≥n}

≡ {$p=x^i$ and i==n}

≡ {$p=x^n$}

⇒ {$p=x^n$}

Step 4: {J and B} ⇒ {vf≥0}

{$p=x^i$ and i≤n and i<n}

≡ {$p=x^i$ and i<n} ⇒

≡ {i<=n}

{n-i>=0}

Step 5: {J and B and vf=VF} S {vf<VF}

{$p=x^i$ and i≤n and i<n and n-i=VF}

≡ {$p=x^i$ and i<n and n-i-1 <VF}

⇒ {n-i-1 < VF}

p:=p*x;

≡ {n-(i+1) < VF}

i:=i+1

{n-i < VF}

LLM Categories:

Encoder-only: focus on learning input data representations

Encoder-decoder: translation and text summarization

Decoder-only: generative tasks (text/code generation) (e.g. GPT)

code gen/completes, bug detect, opt, aut testing, doc creation

Open Problem: Req Eng & Design w/ LLMs

Limited research: few studies on LLM-based req. eng & des.
- engineers show reluctance to rely on LLMs for high-level

Traceability challenges: linking req. to other art (code, tests)

Opportunities: Natural fit (req. often written in nat. lang)

Traceability Sol: automate traceability link creat betw. req. and art.

Potential Use Cases: req. validation, prioritize, compl. checks

Hybrid of LLMs & Other Tech.
- enhance accuracy: combine strengths of mult. approaches
- adds reliability: external tools verify LLM outputs
- expands capabilities: supports complex tasks w/ specialized roles

Code Gen & Completion w/ LLMs

Hallucination Mitigation: code compl. acts as rec system, developers filter hallucinated outputs b4 integ

Improved Productivity: faster task completion

Shift in Focus: dev. review code more ext. rather than writing code

Open-Problem in Code-Gen & Comp.

Correctness (bug-free), Test Integ. (comb. LLMs w/ aut. test to val. out), Fine-tune (eff. tun for dom-spec scen), Explan. Engin. (Optimiz. expl. qual.)

Scientific Evaluation

Correctness: relies heavily on benchmarks like HumanEval

Robustness: consist. in outputs for similar prompts

Explainability: builds confidence and aids understanding for users, valuable for doc. and debug., crit. for user trust

Determinism: ensure reliability in LLM-based system

Security: gen. code often falls below minimal secure coding stand., ensures gen. code is safe for prod

Human Eval

Tests func. impl. in Python. Model must compl. impl given func. signature and docstring

Alg, Simple Math, Reasoning, Lang Comp

Generating New Tests w/ LLMs
- many tests are not executable
- hybrid approach impr cov. & fault detec
- conf: need for better eval frame for reliab.

Test Output Prediction: predicting a program's exec. trace to simulate real-world program behavior during test ⇒ imitate actual exec. w/o running program (faster & scal test eval)

Test Flakiness: a test is flaky if it produces inconsist. results

Test Minimization: remove redund. test cases ⇒ reduce exec time w/o compromising effectiveness

LLM's enable data-driven test minimization, balancing fault detection w/ reduced runtime

Clone detection, Refactoring, Maintenance, Evolution, Doc. Gen (rich code sum)

Method Refactoring

Extract Method: when you have a code fragment that can be reused, move this code to a separate new method and replace the code w/ a call to the method

```
void finishPayment(){
  Invoice invoice = getInvoice();
  print(invoice.getName());
  print(invoice.getDate());}
=>
void finishPayment(){
  Invoice invoice = getIn
  print(invoice); }
void printInvoice(Invoice){
  print(...)
  print(...)
```

Move Method: when a method is used more in another class than its own class, move the method to the class that uses it the most. Turn the code of the original method into a call to the new method

```
class A {
  public void foo(){...}}
class B{
  private A a;
  public void bar(){...
    a.foo() ...}}
=>
class A{}
class B{
  public void foo(){...}
  public void bar(){...
    foo()
  ...}
}
```

Form Template Method: when subclasses have methods that contain similar statements in the same order, move the steps to a new method on a common super class, and override their details in the subclasses

```
abstract class Site{}
class ASite extends Site{
  float getAmount(){
    float base = unit * rate;
    float tax = base * taxRate;
    return base + tax }};
class BSite extends Site{
  float getAmount(){
    float base = unit * rate * 0.5;
    float tax = base * taxRate * 0.2;
    return base + tax }};
=>
abstract class Site{
  float getAmount(){
    return getBase() +
      getTax();}
  float getBase();
  float getTax();
class ASite extends Site{
  float getBase(){
    return unit * rate;}
}
  :
```

Parameter Refactorings

Parameterize Method: when multiple methods perform similar actions that are different only in some aspects, combine these methods by using a parameter that will pass the necessary special aspects

```
class Employee{
  void raiseFivePercent();
  void raiseTenPercent();}
=>
class Employee{
  void raise(int percent)
}
```

Replace Parameter w/ Method Call: when a method makes a query call and sends the result to another method as a parameter, try placing the query call inside the method that uses it directly

```
void calculate(){
  int bP = quant * price;
  int fees = this.getFees();
  int fP = disPrice(bP, fees)}
double disPrice(int b, int f){
  return base * 2 + fees; }
=>
void calculate(){
  int b = quant * price;
  int fP = disPrice(bp)}
double disPrice(int b){
  int fee = this.getFees();
  return base * 2 + fees; }
```

Preserve Whole Object: when you get several values from an object and then pass them as parameters to a method. Instead, pass the whole object

```
int low = temp.getLow();
int high = temp.getHigh();
boolean withinPlan =
plan.withinRange(low, high)
=>
boolean withinPlan =
plan.withinRange(temp);
```

Introduce Parameter Objects: when your methods contain a repeating group of params, replace these params w/ an object

```
class Customer{
amtInvoiced(Dt start, Dt end);
amtReceived(Dt start, Dt end);
amtOverview(Dt start, Dt end);}
=>
class Customer{
  amtInvoiced(DtRg dt)
  amtReceived(DtRg dt)
  amtOverview(DtRg dt)}
class DtRg{ public Dt st, end}
```

Fields Refactorings

Encapsulate Field: when you have a public field, make the field private and create access methods for it

```
class Person{
  public String name;}
=>
class Person{
  private String name;
  public String getName();
  public void setName()}
```

Encapsulate Collection: when a class contains a collection field and a simple getter and setter for working with the collection, make the getter return read-only collection and add methods for adding/deleting elements of the collection

```
class Person{
  priv List<Course> c;
  pub List<Course> getC();
  pub void setC(List ...)};
=>
getCourses(){
  return c
  Collections.unmodifiableList(c)}
publ void addC(Course c)
publ void remC(Course c)
```

Replace Primitive Value w/ Object: when a class contains a primitive field w/ its own behaviors, create a new class, move the old field and its behaviors to it, and replace the field w/ one typed by the new class

```
class Order{
  priv String customer;
  priv String getCName(){...}
  publ void foo(){
    String name = getCName();}
=>
class Order{
  priv Customer customer;
  public void foo(){
    String name =
    customer.getName()
  }}
class Customer{
  private String getName
}
```

Change Bi-dir Association to Unidir.: when you have a bi-dir. assoc. between classes, but one of the classes doesn't use the other's features, remove the unused assoc.

```
class Customer{
  List<Order> orders;
  pub purch(Order o){
    o.customer = this;
    orders.add(o);}
class Order{ Customer c}
=>
class Customer{
  List<Order> orders;
  pub purchase(Order o){
    orders.add(o)}}
class Order{ }
```

Class refactorings

Extract subclass: when a class has features only used in certain cases, create a subclass and use it in these classes

```
class Animal{
  void eat();
  void drink();
  void fly();}
=>
class Animal{
  void eat();
  void drink();}
class FlyingAnimal extends Animal{
  void fly()}
```

Extract superclass: when you have 2 classes w/ common fields and methods, create a shared superclass for them and move all the identical fields and methods to it

```
class Department{
  getName()
  getHeadCount()
  getAnnualCost()}
class Employee{
  getName()
  getId()
  getAnnualCost()}
=>
class Party(){
  getName()
  getAnnualCost()}
class Department extends Party{
  getHeadCount()
  getAnnualCost()}
class Employee extends Party{
  getId()
  getAnnualCost()}
```

Collapse Hierarchy: when you have a class hierarchy in which a subclass is practically the same as its superclass, merge super & sub

```
class Employee{
  String getID()}
class Worker extends Employee{
  String getName()}
=>
class Employee{
  String getId(){...}
  String getName()}
```

Extract class: when one class does the work of two, create a new class and place the fields & methods resp. for the revelant func. in it

```
class Person{
  String name;
  String officeAreaCode;
  String office#;
  String getOffice#(){...}}
=>
class Person{
  String name;
  Telephone# office#;
  String getOffice#(){...}}
class Telephone#{
  String officeAreaCode;
  String office#}
```

Replace Enum. w/ Inheritance: when you have behavior that is affected by an enum, create an inheritance hierarchy, allocate behaviors to it, and call those behaviors polymorphically

```
class Employee{
  public void foo(){
    switch(workerType)
      case ENGINEER: ...
      case SCIENTIST: ...
}
=>
class Employee{
  void print();
  void foo(){print();}
class Engineer extends Emp{
  void print(){...}}
```

Replace Enum w/ Composition: when you have a behavior that is affected by an enum, and you can't use inheritance to mitigate this, replace enum w/ composition

```
class Employee{
  public void foo(){
    switch(workerType)
      case ENGINEER: ...
      case SCIENTIST: ...
}
=>
class Employee{
  EmployeeBehavior behavior
  void foo(){type.print();}}
class EmpBeh{void print();}
class EngBeh extends EmpBeh
  void print(){...}}
```

Replace Delegation w/ Inheritance: when a class contains many simple methods that delegate to all methods of another class, make the class a subclass instead, which makes delegating methods unnec.

```
class Person{
  String getName()}
class Employee{
  Person person;
  String getName(){...}}
=>
class Person{
  String getName(){...}}
class Employee extends
  Person{ }
```

Replace Inheritan Subclass that its superclass, object in it, de get rid of inhe

Delegation: when you have a portion of the methods of ield and put a superclass hods to the superclass, and

```
class Vector{
  boolean isEmpty(){...}}
class Stack extends Vector{}
=>
class Vector{
  boolean isEmpty(){...}
class Stack
  Vector vector;
  boolean isEmpty(){
    return vector.isEmpty
```

Bloaters

Replace Prim Value w/ Object

Primitive Obsession: when prim types are used inst. of real data types

- invalid values could be set, values of diff types could be use interchangeably, validation needs to be done in multiple places

```
int price = 5, String telephone = "818-118-..."
float temp = 32, String password = "xxxx"
```

Long Parameter List: when a long list of params are used for a method

- Intr. Param Object (if params are always coupled)
- Replace param w/ method call (if p can be queried w/ method)
- public void setAddress(String country, String state, String city, String street, int house)
- public float calculatePayment(float price, int quantity, int discount, int fees, ...)

Data Clumps: when diff parts of the code contain identical groups of variables

- Extract Class (vars are mem. of class)
- Preserve Whole Object (derived from an existing obj)

```
String databaseURL = "xxxx"
String username = "yyyy"
String password = "zzzz"
```

Change Preventers

Divergent Change: when one class is changed in diff ways for diff reasons (Single Resp. not followed)

Refactoring: Extract Class / Superclass / Subclass

- class Customer is changed when a new trans. type is supported or when product price changes

Shotgun Surgery: when one class is impl by changing mult classes (Single Resp not followed)

Move Method / Field (to existing common class)

Extract class (move related fields & methods to it)

e.g. To create a new "SupperAdmin" user role, you found yourself editing some methods in Profile, Product and Employee classes

Parallel Inheritance Hierarchy: whenever you create a subclass for a class, you find yourself needing to create a subclass for another clss (spec. case of shotgun S)

Move Method / Field (to common class)

e.g. you have a class hierarchy for Department, and a class hierarchy for the Privilege for each Dep.

Object-Oriented Abusers

Alternative Class w/ Diff Interfaces: when two classes perform identical functions but have diff method signatures (usually due to lack of comm.)

Rename Method (to a common name), [Move Method, Add Param, Param Method] (to unify the interface), Extract Superclass (to unify part of interface / impl)

e.g. "class QuickSorter" has a method "void sort()" and "class BubbleSorter" has method "void getSorted(int[] data)"

Refused Bequest: when a subclass uses only some of the methods and properties inherited from its parents (a sign they are not proper subclass / superclass)

Extract Superclass (move re-used parts to it and make both classes inherit it), Replace Inheritance by Delegation (re-use)

- "class Person" inherits from "class Building" to reuse its attributes "name" and "address"
- "class Order" inherits from "class List" to reuse operations "add" and "remove"

Switch Statements: big switch/if statements scattered in many places. Replace Enum w/ Inheritance / Comp

```
int area = switch(type){
  case "square": calcSquareArea(4);
  case "circle": calcCircleArea(5);
```

Dispensables

Extract/Move Method, Inline Method (to unify code)

Duplicate Code: 2 fragments that look identical

- Form Template Method (when code is similar but not identical)

```
void calcRecArea(Rec r)
  int width = r.width;
  int height = r.height;
  return width * height;
}
void getAreaOf Rec(Rec r){
  return r.width * r.height;
}
```

Speculative Generality: when there is an unused class, method, field, or parameter. Collapse Hierarchy (for abstract classes)

- Inline Class / Method, Remove Param (to move used class/meth/p)

e.g. "class BestUtils" has no attr or methods (could be a feature that did not pan out)

Lazy class: class does not do enough to justify its existence

- Inline Class (near-useless classes), Collapse Hierarchy (useless class is a subclass)

e.g. "class ControllerUtils" only has a single method "void static synchronize(View view, Model model)"

Data Class: when a class contains only fields and crude methods for accessing them (get & set) and not real behavior. Move/Extract Method (mov relevant beh to data class)

Encapsulate Field / Collection (hide impl details)

- "class Rectangle" w/ only "int width" and "int height" attributes, while method "calcArea(Rect)" is on another class
- "class Account" w/ a public attribute "ArrayList<Order> orders", while method "add(Account a, Order o)" is in an. cls

Couplers

Feature Envy: method access to data of another obj. more than its own (could be after data has moved to a data class)

Move/Extract Method (move method to the class w/ data)

e.g. "method 2DUtils.calculateArea" accesses attribute "int width" and "int height" on "class Rectangle"

Inappropriate Intimacy: when one class uses the internal fields and methods of another class

- Move Method / Field (using class if they do not belong to used cls)
- Replace Del. w/ Inher (if classes are subclass / superclass)

e.g. "class Account" has a "method onChange()" that loops on its "List<AccountObserver> observers" calling not only the "notify()" method from the inherited Observer interface, but also method "dispatchEmail()" on "class AccountObserver"

Non-det path est = (# condition evals)^(# of non-det paths)

```
e.g. for i in range(N) -> N iters
       if 2%2==0:  -> deterministic
       ...
       x = math.rand()  -> non-det
       if x%2==0:
       ...
       else if x%3==0     } 3 cond evals
       ...
       else
       ...
```

$\Rightarrow 3^N$ paths

Regression Testing - select tests that activate dangerous edges (paths changed)

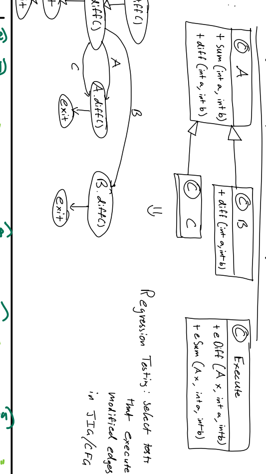
```
e.g. def squon(x,-1)
       s=0, i=0
       while (i<x):
         s = s+2^i+1
         i++
```

$\Rightarrow$ J: $i \le x$ && $s = i^2$; vf: $x - i$

1) At init $i=0, s=0, x \ge 0$ ∴ $0 \le i \le x$ && $s = i^2$
2) during sps. B: $i != x$ && J: $i \le x$ && $s = i^2$, show J after each iter
3) sps. !B (end) => $i == x$, show still J = true
4) sps. B => $i != x$, show vf ≥ 0
5) show vf decreases: either init to init+1 or init -> end

Java Interclass Diag (JIG)

Given:



Regression Testing: select tests that execute modified edges in JIG/CFG

- Composition: src ◆—— target; when src deleted => target deleted (cascades)
- No count => default 1
- association => "pointer", when src del => target not deleted
- attribute => primitive, default, pass default
- operation => member func, +/-/# => visibility
- syntax: name: Type [multiplicity] {modifiers}; Type ∈ {primitive, Enum}; mod ∈ {id, ordered, unique}; multiplicity: int, range, bounded/unbounded(*); id: glob unique; ordered: list; unique: set
- inheritance => "is-a"; assoc/comp => "has-a"

• Association: ——> (bi-dir = no arrowhead); b/w use case & actor/external system

• Inheritance: sub ——▷ super

• Include: src --include--> target; src always incls target on exec

• Extend: src --extend--> target; target sometimes incls src on exec