Model Driven Development: software development method that focuses on creating & exploiting models in software development activities ↔ reverse engineering
Forward engineering: translating models to code

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
mapping
semantic? dev writes? see
- 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 to gether 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>: <literal>='<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, no quotes
- STRING is quoted string ("hello") but ID is unquoted
- <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> ... | <ruleN> ← 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)

Software Testing: observed vs. expected behavior
driven by white-box testing; black-box testing
fault of omission: missing funct, incomplete (imp)
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

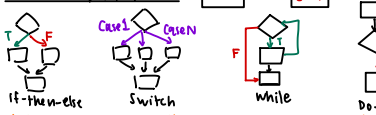
Isolating the Item-Under-Test (IUT): reduce uncertainties
Test stub: simulate behavior of comp IUT depends on (called)
Test driver: test IUT when callers not available yet (test from above)

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 via auto exec of software tests

JUnit: Test Suite [(Test Case 1: Test Fixture, Test Method..., near dump Text Fixt)]
@Before Class: runs before all test cases in the class @After Class
@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 (2 obj references point to same loc in memory)
assertEquals(bool exp, bool act), assertTrue(bool cond) False
assertNotNull/Null(Object obj), assert Same/Not Same(1,2)

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^(# of non-det. decisions)
# of feas. paths

for (int j=1; j<=Math.pow(3,i); j++)
T(n) = 3 + 3^2 + 3^3 + ... + 3^n
3T(n) = 3^2 + 3^3 + ... + 3^n + 3^(n+1)
2T(n) = 3^(n+1) − 3
T(n) = (3^(n+1) − 3)/2
Math.pow(2, t)%2 == 0 is deterministic
2^((3^(n+1) − 3)/2)

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

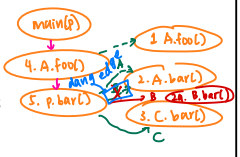
Test Generation
Find concrete input for each path cond after symbolic exec.

Regression Test Selection (RTS): P (old ver) P' (new) T: test suite
Assume all tests in T ran on P ⇒ generate coverage mnt. 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)

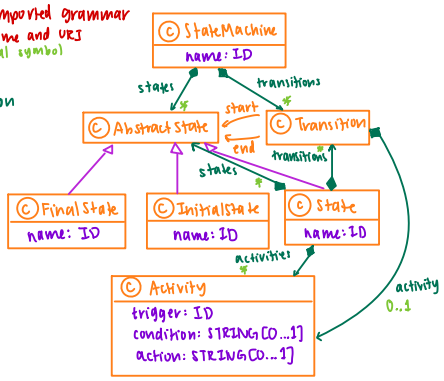
polymorphism allows memb function call to be resolved at runtime

```
class A {
1  public static void foo(){...}
2  public void bar(){...}}
class B extends A {} bar 2a
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 k/n ← kills k out of n mutants

grammar grammar name with imported grammar
generate statemachine metamodel name and URI
StateMachine: nonterm = class terminal symbol
'state' 'machine' name=ID '{'
states+=AbstractState* transitions+=Transition*
'}';
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 Gumball Machine {
  initial start
  state S1
  state S2
    entry ["x<10"] / "x++"
    exit / "x--"
    initial I2
    state S2.1 {
      :
    }
  }
  final
  I->S1
  S1->S2.1
}
```

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 post condition of S w/ respect to P: the most precise Q such that {P} S {Q}
- weakest pre condition 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·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}

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

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

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

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

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

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

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

Fields Refactorings

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

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

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

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 refactorings

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

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

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

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

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

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

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.

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

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)

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)

Change Preventers

Divergent Change: when one class is changed in diff ways for diff reasons (Single Resp. not followed)
Refactoring: Extract Class / Superclass / Subclass

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

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

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)

Refused Request: 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 (if no use)

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

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)

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)

Lazy class: class does not do enough to justify its existence
- Inline Class (near-useless classes), Collapse Hierarchy (useless class is a subclass)

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)

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)

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)